A VISUAL REFERENCE · jq and yq

jq and yq, one change at a time

Read, transform and validate JSON and YAML with complete local inputs and actual command-line results. One question. A small filter. Its exact output.

The Org source owns the programs and synthetic fixtures. Each downloadable script recreates its inputs in an owned temporary directory. The final PDF is also versioned at the repository root for offline reading.

> **Last changes: 2026-11-01.** First combined edition: jq JSON filters, Mike Farah yq v4, paired queries, configuration edits, format conversion and explicit validation outcomes.

84
worked examples

14
topic families

1
canonical Org source

READING THE TOUR

From input to a useful result

Input

Read the complete local JSON or YAML fixture before choosing a path, filter or missing-value rule.

Command

Copy the focused command or download its complete standalone script, including the exact input setup.

Actual result

Compare captured output against independently authored expectations. Expected failures show their caught exit status.

The code and result gallery links to each example. Search by a command, topic or stable JYQ identifier. Inputs are shown beside the learning task; expand Complete runnable example for the owned temporary-directory setup. PDF pages retain input, command, output and parameter explanations.

Choose the intended yq implementation

This tour uses Mike Farah yq v4, the Go command-line program. Its expressions resemble jq but are not the same language. In particular, jq object shorthand and reduce syntax must not be assumed to work in yq. The separate Python yq project wraps jq and has different CLI flags. The examples here do not use that Python wrapper.

Run `jq --version` and `yq --version` before starting. This edition is verified with jq 1.7.1-apple and Mike Farah yq v4.54.1. On macOS, `brew install jq yq` installs the intended command-line programs. Other platforms can use the upstream installation instructions. The shared publisher supplies Python, batch Emacs and the HTML/PDF renderer; no Python YAML parser executes the filters.

A small shared dataset

The three observations are invented teaching data. A and C belong to north; B belongs to south. Their values are 10, 20 and null. Null means a missing measurement, not zero. The JSON and YAML catalog fixtures have equal values. Separate fixtures introduce configuration overlays, YAML documents, type tags, comments and aliases. The malformed JSON file is intentionally invalid.

Most structural results use compact JSON to make comparisons exact. YAML examples that teach presentation use YAML output. Converting YAML to JSON does not preserve comments, anchor names or all YAML type distinctions.

The jq alternative operator replaces false as well as null. Do not use it as an automatic missing-value repair when false is meaningful. Merging a configuration also requires an explicit choice between replacing and appending sequences.

Read an exit status correctly

Each focused command is executed by a fresh Bash driver. The displayed Captured run exit status belongs to that driver. In failure examples, the driver catches the processor status and prints an `exit` line in the output. The malformed-JSON example reports rejection instead of a build-specific parser code. That makes a deliberately rejected input a successful, verified demonstration. No example reads a real service configuration, contacts a network endpoint, or writes outside the temporary directory it creates and removes.

Primary references

- jq 1.7 manual: filters, arguments, streams and exit status
- Mike Farah yq documentation and installation
- yq deep merge, sequence replacement and append flags
- yq env, strenv and envsubst
- yq anchors, aliases and explode
- yq comment metadata
- The separate Python yq implementation, for identifying the difference

01 - First JSON queries

Start with a small JSON object. Change the filter or output option while keeping the input fixed.

JYQ-01-001

Keep the complete JSON value

The identity filter returns its input. Compact output removes presentation whitespace without changing the JSON value.

Input

data/intro.json

```
{
  "project": "field-notes",
  "version": 1,
  "active": true,
  "owner": null
}
```

Code

```
jq -c '.' data/intro.json
```

Actual output

Captured run · exit 0

```
{"project":"field-notes","version":1,"active":true,"owner":null}
```

Parameter	Meaning in this example
`.`	Identity returns the complete input value; -c writes one compact JSON result.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-01-001.sh

JYQ-01-002

Select one string as JSON

A selected string is still a JSON string, including its quotes. Use this form when another JSON processor consumes the result.

Input

data/intro.json

```
{
  "project": "field-notes",
  "version": 1,
  "active": true,
  "owner": null
}
```

Code

```
jq '.project' data/intro.json
```

Actual output

Captured run · exit 0

```
"field-notes"
```

Parameter	Meaning in this example
`.project`	Read the project member; ordinary jq output retains JSON string quotes.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-01-002.sh

JYQ-01-003

Print the same string without JSON quotes

Only the output representation changes. Raw string output is useful for shell arguments, filenames and reports.

Input

data/intro.json

```
{
  "project": "field-notes",
  "version": 1,
  "active": true,
  "owner": null
}
```

Compare with JYQ-01-002 · Select one string as JSON.

Code

```
jq -r '.project' data/intro.json
```

Actual output

Captured run · exit 0

```
field-notes
```

Parameter	Meaning in this example
`-r`	Write strings as text; it does not turn objects or arrays into shell-safe words.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-01-003.sh

JYQ-01-004

Build a smaller object

An object constructor selects the fields that a downstream step actually needs. jq allows this shorthand for members with unchanged names.

Input

data/intro.json

```
{
  "project": "field-notes",
  "version": 1,
  "active": true,
  "owner": null
}
```

Code

```
jq -c '{project, version}' data/intro.json
```

Actual output

Captured run · exit 0

```
{"project":"field-notes","version":1}
```

Parameter	Meaning in this example
`{project, version}`	Construct two members using their existing names and values.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-01-004.sh

JYQ-01-005

Count array elements

The pipe passes the samples array to length. It counts records, including a record whose value member is null.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq '.samples | length' data/catalog.json
```

Actual output

Captured run · exit 0

```
3
```

Parameter	Meaning in this example
`| length`	Run length against the selected array, not against the outer object.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-01-005.sh

JYQ-01-006

Read an array element

Array indexes start at zero. Follow the selected element with another object-member lookup.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -r '.samples[0].id' data/catalog.json
```

Actual output

Captured run · exit 0

```
A
```

Parameter	Meaning in this example
`[0]`	Select the first array element before reading its id.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-01-006.sh

02 - Iterate and select JSON records

Separate a stream of results from an array containing those results. Select records before projecting their fields.

JYQ-02-001

Emit one result per record

Iteration produces three separate JSON results. Compact output puts each result on its own line; it does not collect them into an array.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -c '.samples[] | {id, site}' data/catalog.json
```

Actual output

Captured run · exit 0

```
{"id":"A","site":"north"}
{"id":"B","site":"south"}
{"id":"C","site":"north"}
```

Parameter	Meaning in this example
`.samples[]`	Emit each array element as a separate input to the following filter.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-02-001.sh

JYQ-02-002

Collect IDs into one array

Array brackets collect the values produced inside them. This is one result rather than three independent outputs.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Compare with JYQ-02-001 · Emit one result per record.

Code

```
jq -c '[.samples[].id]' data/catalog.json
```

Actual output

Captured run · exit 0

```
["A","B","C"]
```

Parameter	Meaning in this example
`[filter]`	Collect every result of the inner filter into an array.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-02-002.sh

JYQ-02-003

Keep records from one site

select passes matching records through unchanged. The id lookup happens after the site condition has been tested.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -c '[.samples[] | select(.site == "north") | .id]' data/catalog.json
```

Actual output

Captured run · exit 0

```
["A","C"]
```

Parameter	Meaning in this example
`select(condition)`	Keep the current record only when its condition is true.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-02-003.sh

JYQ-02-004

Filter numbers while excluding null

State both the missing-value rule and the numeric threshold. The null observation is excluded deliberately.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -c '[.samples[] | select(.value != null and .value >= 15) | .id]' data/catalog.json
```

Actual output

Captured run · exit 0

```
["B"]
```

Parameter	Meaning in this example
`and`	Require a non-null value and a value of at least fifteen.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-02-004.sh

JYQ-02-005

Test whether a key exists

A missing key and an explicitly null key can have different meanings. has inspects membership instead of reading a possibly absent value.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -c '[.samples[] | {id, has_note: has("note")}]' data/catalog.json
```

Actual output

Captured run · exit 0

```
[{"id":"A","has_note":false},{"id":"B","has_note":true},{"id":"C","has_note":false}]
```

Parameter	Meaning in this example
`has("note")`	Test object membership for the note key.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-02-005.sh

JYQ-02-006

Flatten optional nested arrays

Iterate through each tags array and collect its members. An empty tags array contributes no values.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -c '[.samples[] | .tags[]?]' data/catalog.json
```

Actual output

Captured run · exit 0

```
["soil","dry","water"]
```

Parameter	Meaning in this example
`[]?`	Optional iteration suppresses an error for an unsuitable value; it does not invent a replacement.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-02-006.sh

03 - Transform JSON values

Construct the output shape explicitly. Keep null handling visible rather than silently discarding observations.

JYQ-03-001

Map records to IDs

map applies the filter to each array member and collects the results. Here it is another spelling of the earlier ID collection.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Compare with JYQ-02-002 · Collect IDs into one array.

Code

```
jq -c '.samples | map(.id)' data/catalog.json
```

Actual output

Captured run · exit 0

```
["A","B","C"]
```

Parameter	Meaning in this example
`map(filter)`	Transform the selected array one element at a time.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-03-001.sh

JYQ-03-002

Replace null measurements with zero

The replacement is a policy choice, not a data repair. Keep the original input unchanged and make the substitution explicit in the output.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -c '.samples | map({id, value: (.value // 0)})' data/catalog.json
```

Actual output

Captured run · exit 0

```
[{"id":"A","value":10},{"id":"B","value":20},{"id":"C","value":0}]
```

Parameter	Meaning in this example
`// 0`	Supply zero when the value is null or false; see the later boolean caveat.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-03-002.sh

JYQ-03-003

Remove null measurements from an array

Filtering and replacement answer different questions. This result contains two measurements, while replacing null with zero keeps three.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Compare with JYQ-03-002 · Replace null measurements with zero.

Code

```
jq -c '[.samples[].value | select(. != null)]' data/catalog.json
```

Actual output

Captured run · exit 0

```
[10,20]
```

Parameter	Meaning in this example
`select(. != null)`	Discard null results rather than assigning a numeric replacement.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-03-003.sh

JYQ-03-004

Update a number relative to itself

Relative update uses the existing value. The command writes an updated result to stdout; it does not edit the input file.

Input

data/intro.json

```
{
  "project": "field-notes",
  "version": 1,
  "active": true,
  "owner": null
}
```

Code

```
jq -c '.version += 1 | {project, version}' data/intro.json
```

Actual output

Captured run · exit 0

```
{"project":"field-notes","version":2}
```

Parameter	Meaning in this example
`+= 1`	Add one to the existing version member.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-03-004.sh

JYQ-03-005

Delete a member and inspect the keys

Deletion produces a new value without owner. jq keys returns object keys in sorted order.

Input

data/intro.json

```
{
  "project": "field-notes",
  "version": 1,
  "active": true,
  "owner": null
}
```

Code

```
jq -c 'del(.owner) | keys' data/intro.json
```

Actual output

Captured run · exit 0

```
["active","project","version"]
```

Parameter	Meaning in this example
`del(.owner)`	Remove the owner member from the output object.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-03-005.sh

JYQ-03-006

Label missing observations

A conditional can preserve the distinction between observed and missing data without changing the numeric values.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -c '.samples | map({id, status: (if .value == null then "missing" else "ok" end)})'
data/catalog.json
```

Actual output

Captured run · exit 0

```
[{"id":"A","status":"ok"},{"id":"B","status":"ok"},{"id":"C","status":"missing"}]
```

Parameter	Meaning in this example
`if ... then ... else ... end`	Choose a status for each record using its own value.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-03-006.sh

04 - Group, sort and reshape JSON

Change the organization of the data: totals, groups, indexes and key-value entries. Sort rules and missing observations remain explicit.

JYQ-04-001

Sum observed measurements

The total includes only observed numbers. The selection is applied before addition so the missing-value rule can be reviewed.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq '[.samples[].value | select(. != null)] | add' data/catalog.json
```

Actual output

Captured run · exit 0

```
30
```

Parameter	Meaning in this example
`add`	Combine the numbers in the collected array.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-04-001.sh

JYQ-04-002

Count records by site

Grouping yields arrays of records with the same site. Count records inside each group, including the observation with a null measurement.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -c '.samples | group_by(.site) | map({site: .[0].site, count: length})' data/
catalog.json
```

Actual output

Captured run · exit 0

```
[{"site":"north","count":2},{"site":"south","count":1}]
```

Parameter	Meaning in this example
`group_by(.site)`	Group and order records by the site value.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-04-002.sh

JYQ-04-003

Sort records by measurement

JSON sort order puts null before numbers. If null must appear elsewhere, define that policy instead of assuming it is a large number.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -c '.samples | sort_by(.value) | map(.id)' data/catalog.json
```

Actual output

Captured run · exit 0

```
["C","A","B"]
```

Parameter	Meaning in this example
`sort_by(.value)`	Order records by the selected value; null sorts before numeric values.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-04-003.sh

JYQ-04-004

Find distinct site names

unique removes duplicates and returns sorted values. It is suitable for a compact list of categories.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -c '[.samples[].site] | unique' data/catalog.json
```

Actual output

Captured run · exit 0

```
["north","south"]
```

Parameter	Meaning in this example
`unique`	Return sorted distinct values from an array.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-04-004.sh

JYQ-04-005

Build an ID lookup

INDEX builds an object keyed by each record ID. IDs must be unique when every original record must remain available.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -c '.samples | INDEX(.id) | .B | {id, site}' data/catalog.json
```

Actual output

Captured run · exit 0

```
{"id":"B","site":"south"}
```

Parameter	Meaning in this example
`INDEX(.id)`	Use each id as an object key; duplicate IDs would overwrite an earlier entry.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-04-005.sh

JYQ-04-006

Expose object entries as records

An object can become an array of explicit key-value records. This representation is convenient for subsequent filtering or reshaping.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -c '.samples[0] | {id, value} | to_entries' data/catalog.json
```

Actual output

Captured run · exit 0

```
[{"key":"id","value":"A"},{"key":"value","value":10}]
```

Parameter	Meaning in this example
`to_entries`	Represent each object member with key and value fields.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-04-006.sh

05 - Arguments, files and JSON streams

Pass values through CLI arguments rather than constructing filter source with shell interpolation. Distinguish JSON Lines from one JSON array.

JYQ-05-001

Pass a string argument safely

The argument becomes a jq variable. It is data, not a piece of filter source, so shell quoting and jq syntax stay separate.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Compare with JYQ-02-003 · Keep records from one site.

Code

```
jq -c --arg site north '[.samples[] | select(.site == $site) | .id]' data/catalog.json
```

Actual output

Captured run · exit 0

```
["A","C"]
```

Parameter	Meaning in this example
`--arg site north`	Bind the string north to the jq variable $site.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-05-001.sh

JYQ-05-002

Pass a numeric argument

The threshold is parsed as JSON and therefore remains numeric. Using --arg would bind the text string instead.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Compare with JYQ-02-004 · Filter numbers while excluding null.

Code

```
jq -c --argjson minimum 15 '[.samples[] | select(.value != null and .value >= $minimum)
| .id]' data/catalog.json
```

Actual output

Captured run · exit 0

```
["B"]
```

Parameter	Meaning in this example
`--argjson minimum 15`	Bind a JSON number rather than the string "15".
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-05-002.sh

JYQ-05-003

Load a second JSON file

slurpfile binds an array of all JSON values in the lookup file. Select its first object before looking up the current record site.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

data/lookup.json

```
{"north":{"team":"red"},"south":{"team":"blue"}}
```

Code

```
jq -c --slurpfile lookup data/lookup.json '[.samples[] | {id, team: $lookup[0]
[.site].team}]' data/catalog.json
```

Actual output

Captured run · exit 0

```
[{"id":"A","team":"red"},{"id":"B","team":"blue"},{"id":"C","team":"red"}]
```

Parameter	Meaning in this example
`--slurpfile lookup`	Read the lookup file into the array $lookup; it does not replace the main input.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-05-003.sh

JYQ-05-004

Process JSON Lines one value at a time

jq accepts a sequence of JSON values. The filter runs once per input record; no outer array is needed.

Input

data/samples.jsonl

```
{"id":"A","site":"north","value":10}
{"id":"B","site":"south","value":20}
{"id":"C","site":"north","value":null}
```

Code

```
jq -c '{id, value}' data/samples.jsonl
```

Actual output

Captured run · exit 0

```
{"id":"A","value":10}
{"id":"B","value":20}
{"id":"C","value":null}
```

Parameter	Meaning in this example
`JSON Lines`	Each line contains a separate JSON object, so the filter has three inputs.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-05-004.sh

JYQ-05-005

Collect a JSON stream before aggregation

Slurping collects every input value into one array. It is convenient for small files but requires memory for the complete stream.

Input

data/samples.jsonl

```
{"id":"A","site":"north","value":10}
{"id":"B","site":"south","value":20}
{"id":"C","site":"north","value":null}
```

Code

```
jq -s 'map(.value // 0) | add' data/samples.jsonl
```

Actual output

Captured run · exit 0

```
30
```

Parameter	Meaning in this example
`-s`	Collect all input values before running the filter once.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-05-005.sh

JYQ-05-006

Read remaining inputs from a null-input program

Null-input mode starts the program once. inputs then reads the supplied JSON stream explicitly.

Input

data/samples.jsonl

```
{"id":"A","site":"north","value":10}
{"id":"B","site":"south","value":20}
{"id":"C","site":"north","value":null}
```

Code

```
jq -n -c '[inputs | .id]' data/samples.jsonl
```

Actual output

Captured run · exit 0

```
["A","B","C"]
```

Parameter	Meaning in this example
`-n and inputs`	Start once with null, then consume the available input values.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-05-006.sh

06 - Missing values and predictable failures

Separate missing keys, explicit null, false, empty output and process exit status. Expected failures are caught and printed by the standalone example.

JYQ-06-001

Distinguish missing membership from null

Both lookups return null, but only owner is an existing member. A lookup alone cannot distinguish the two cases.

Input

data/intro.json

```
{
  "project": "field-notes",
  "version": 1,
  "active": true,
  "owner": null
}
```

Code

```
jq -c '[.owner, .absent, has("owner"), has("absent")]' data/intro.json
```

Actual output

Captured run · exit 0

```
[null,null,true,false]
```

Parameter	Meaning in this example
`has versus lookup`	Use membership when the presence of the key itself matters.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-06-001.sh

JYQ-06-002

Remember that fallback also replaces false

The alternative operator treats false and null as absent. Numeric zero and an empty string are retained.

Code

```
jq -n -c '[false // "fallback", 0 // "fallback", "" // "fallback"]'
```

Actual output

Captured run · exit 0

```
["fallback",0,""]
```

Parameter	Meaning in this example
`//`	Fallback is triggered by false or null, not by every empty-looking value.
`No input file`	The command constructs its input explicitly; the script does not read stdin.

Download: JYQ-06-002.sh

JYQ-06-003

Validate a condition across all records

The predicate is evaluated for each sample. The nonempty fixture passes because every id is a string.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -e 'all(.samples[]; .id | type == "string")' data/catalog.json
```

Actual output

Captured run · exit 0

```
true
```

Parameter	Meaning in this example
`all(stream; predicate)`	Return one boolean describing whether every emitted item passes the test.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-06-003.sh

JYQ-06-004

Inspect the exit status of a null result

jq can parse and run successfully while -e reports a false or null final result. The shell catches that expected status so the example can display it.

Input

data/intro.json

```
{
  "project": "field-notes",
  "version": 1,
  "active": true,
  "owner": null
}
```

Code

```
if jq -e '.owner' data/intro.json; then
  status=0
else
  status=$?
fi
printf 'exit=%s\n' "$status"
```

Actual output

Captured run · exit 0

```
null
exit=1
```

Parameter	Meaning in this example
`-e`	Return status 1 when the last result is false or null; the captured example itself finishes successfully.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-06-004.sh

JYQ-06-005

Catch a conversion error

A conversion can fail for one element without discarding the entire array. The catch result is an explicit null replacement.

Code

```
jq -n -c '["12", "bad"] | map(try tonumber catch null)'
```

Actual output

Captured run · exit 0

```
[12,null]
```

Parameter	Meaning in this example
`try ... catch`	Catch a filter error and emit a chosen replacement value.
`No input file`	The command constructs its input explicitly; the script does not read stdin.

Download: JYQ-06-005.sh

JYQ-06-006

Reject malformed JSON

This fixture is deliberately incomplete JSON. The diagnostic is captured separately. Compare acceptance or rejection because parser-error wording and numeric status can vary by jq build.

Input

data/broken.json

```
{"project":
```

Code

```
if jq . data/broken.json > result.json 2> diagnostic.txt; then
  printf 'parse=accepted\n'
else
  printf 'parse=rejected\n'
fi
```

Actual output

Captured run · exit 0

```
parse=rejected
```

Parameter	Meaning in this example
`Parse failure`	The input cannot be decoded; report the nonzero outcome without depending on a particular parser-error code.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-06-006.sh

07 - The same questions in YAML

Use Mike Farah yq v4 with the equivalent YAML data. Explicit JSON output makes results easy to compare with the earlier jq examples.

JYQ-07-001

Read a YAML string

yq reads the YAML mapping and prints the selected scalar without JSON quotes. The same project value was read by jq earlier.

Input

data/intro.yaml

```
project: field-notes
version: 1
active: true
owner: null
```

Compare with JYQ-01-003 · Print the same string without JSON quotes.

Code

```
yq '.project' data/intro.yaml
```

Actual output

Captured run · exit 0

```
field-notes
```

Parameter	Meaning in this example
`.project`	Select a YAML mapping member; scalar output is unwrapped by default.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-07-001.sh

JYQ-07-002

Construct a YAML-derived JSON object

Use explicit object keys in the yq expression. jq object shorthand is not assumed to be portable to this different language.

Input

data/intro.yaml

```
project: field-notes
version: 1
active: true
owner: null
```

Compare with JYQ-01-004 · Build a smaller object.

Code

```
yq -o=json -I=0 '{"project": .project, "version": .version}' data/intro.yaml
```

Actual output

Captured run · exit 0

```
{"project":"field-notes","version":1}
```

Parameter	Meaning in this example
`-o=json -I=0`	Serialize the result as compact JSON, with zero indentation.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-07-002.sh

JYQ-07-003

Index a YAML sequence

YAML sequences use the same zero-based indexing idea. The serialization format changes while the selected ID remains the same.

Input

data/catalog.yaml

```
samples:
  - {id: A, site: north, value: 10, tags: [soil, dry]}
  - {id: B, site: south, value: 20, tags: [water], note: review}
  - {id: C, site: north, value: null, tags: []}
```

Compare with JYQ-01-006 · Read an array element.

Code

```
yq '.samples[0].id' data/catalog.yaml
```

Actual output

Captured run · exit 0

```
A
```

Parameter	Meaning in this example
`[0]`	Read the first sequence element, then its id member.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-07-003.sh

JYQ-07-004

Count a YAML sequence

The sequence has three records. A null member inside one record does not remove that record from the sequence.

Input

data/catalog.yaml

```
samples:
  - {id: A, site: north, value: 10, tags: [soil, dry]}
  - {id: B, site: south, value: 20, tags: [water], note: review}
  - {id: C, site: north, value: null, tags: []}
```

Compare with JYQ-01-005 · Count array elements.

Code

```
yq '.samples | length' data/catalog.yaml
```

Actual output

Captured run · exit 0

```
3
```

Parameter	Meaning in this example
`length`	Count sequence members rather than mapping keys.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-07-004.sh

JYQ-07-005

Collect YAML IDs as JSON

Collect the emitted IDs into a sequence. JSON serialization provides an exact common representation for comparison.

Input

data/catalog.yaml

```
samples:
  - {id: A, site: north, value: 10, tags: [soil, dry]}
  - {id: B, site: south, value: 20, tags: [water], note: review}
  - {id: C, site: north, value: null, tags: []}
```

Compare with JYQ-02-002 · Collect IDs into one array.

Code

```
yq -o=json -I=0 '[.samples[].id]' data/catalog.yaml
```

Actual output

Captured run · exit 0

```
["A","B","C"]
```

Parameter	Meaning in this example
`[...]`	Collect emitted scalar values into one sequence.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-07-005.sh

JYQ-07-006

Select the same site in YAML

The site condition is intentionally identical to the earlier jq task. Both examples must produce the same IDs.

Input

data/catalog.yaml

```
samples:
  - {id: A, site: north, value: 10, tags: [soil, dry]}
  - {id: B, site: south, value: 20, tags: [water], note: review}
  - {id: C, site: north, value: null, tags: []}
```

Compare with JYQ-02-003 · Keep records from one site.

Code

```
yq -o=json -I=0 '[.samples[] | select(.site == "north") | .id]' data/catalog.yaml
```

Actual output

Captured run · exit 0

```
["A","C"]
```

Parameter	Meaning in this example
`select`	Keep matching sequence elements before reading their IDs.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-07-006.sh

08 - Edit YAML configuration values

Assignment prints an updated document unless in-place mode is requested. The original fixtures remain unchanged in every example.

JYQ-08-001

Assign a configuration value

Plain assignment uses the value on its right-hand side. The filter selects a small view of the updated configuration for inspection.

Input

data/settings.yaml

```
service:
  name: api # Public name
  image: demo:1.0
  replicas: 2
  enabled: false
  labels:
    app.kubernetes.io/name: demo
  ports: [8080, 9090]
```

Code

```
yq -o=json -I=0 '.service.replicas = 3 | .service | {"name": .name,
"replicas": .replicas}' data/settings.yaml
```

Actual output

Captured run · exit 0

```
{"name":"api","replicas":3}
```

Parameter	Meaning in this example
`= 3`	Assign an absolute replacement, then inspect the changed members.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-08-001.sh

JYQ-08-002

Increment a YAML number

Update assignment changes the context on the right to the selected old value. Here that context is the integer two.

Input

data/settings.yaml

```
service:
  name: api # Public name
  image: demo:1.0
  replicas: 2
  enabled: false
  labels:
    app.kubernetes.io/name: demo
  ports: [8080, 9090]
```

Code

```
yq '.service.replicas |= . + 1 | .service.replicas' data/settings.yaml
```

Actual output

Captured run · exit 0

```
3
```

Parameter	Meaning in this example
`|=`	Run the right-hand expression against each selected old value.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-08-002.sh

JYQ-08-003

Remove a YAML mapping member

Deleting labels removes the member from the output value. Testing membership proves that the key is absent.

Input

data/settings.yaml

```
service:
  name: api # Public name
  image: demo:1.0
  replicas: 2
  enabled: false
  labels:
    app.kubernetes.io/name: demo
  ports: [8080, 9090]
```

Code

```
yq 'del(.service.labels) | .service | has("labels")' data/settings.yaml
```

Actual output

Captured run · exit 0

```
false
```

Parameter	Meaning in this example
`del`	Remove a selected mapping member rather than assigning null to it.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-08-003.sh

JYQ-08-004

Read a key that contains dots

Dots inside a quoted key are ordinary characters. Without the quotes they would be interpreted as path separators.

Input

data/settings.yaml

```
service:
  name: api # Public name
  image: demo:1.0
  replicas: 2
  enabled: false
  labels:
    app.kubernetes.io/name: demo
  ports: [8080, 9090]
```

Code

```
yq '.service.labels."app.kubernetes.io/name"' data/settings.yaml
```

Actual output

Captured run · exit 0

```
demo
```

Parameter	Meaning in this example
`Quoted path member`	Treat the complete dotted label name as one mapping key.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-08-004.sh

JYQ-08-005

Append a port to a sequence

Sequence addition appends the new member. This preserves the two existing ports and their order.

Input

data/settings.yaml

```
service:
  name: api # Public name
  image: demo:1.0
  replicas: 2
  enabled: false
  labels:
    app.kubernetes.io/name: demo
  ports: [8080, 9090]
```

Code

```
yq -o=json -I=0 '.service.ports += [7070] | .service.ports' data/settings.yaml
```

Actual output

Captured run · exit 0

```
[8080,9090,7070]
```

Parameter	Meaning in this example
`+= [7070]`	Append a sequence rather than replacing the existing sequence.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-08-005.sh

JYQ-08-006

Update one matching record

Parentheses identify the complete assignment target. Only the record with ID C receives the replacement.

Input

data/catalog.yaml

```
samples:
  - {id: A, site: north, value: 10, tags: [soil, dry]}
  - {id: B, site: south, value: 20, tags: [water], note: review}
  - {id: C, site: north, value: null, tags: []}
```

Code

```
yq -o=json -I=0 '(.samples[] | select(.id == "C") | .value) = 0 | .samples |
map(.value)' data/catalog.yaml
```

Actual output

Captured run · exit 0

```
[10,20,0]
```

Parameter	Meaning in this example
`(path | select(...)) = value`	Locate a matching member for assignment while retaining the surrounding document.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-08-006.sh

09 - Merge files and read multiple YAML documents

Deep merge, shallow merge and sequence appending are different operations. Document boundaries are also different from sequence elements.

JYQ-09-001

Deep-merge a configuration overlay

Deep merge keeps the base name and combines label mappings. By default, the overlay replaces the complete ports sequence.

Input

data/base.yaml

```
service:
  name: api
  replicas: 1
  labels: {app: demo}
  ports: [80]
```

data/overlay.yaml

```
service:
  replicas: 3
  labels: {owner: lab}
  ports: [443]
```

Code

```
yq -o=json -I=0 '. * load("data/overlay.yaml") | .service' data/base.yaml
```

Actual output

Captured run · exit 0

```
{"name":"api","replicas":3,"labels":{"app":"demo","owner":"lab"},"ports":[443]}
```

Parameter	Meaning in this example
`* load(...)`	Recursively merge mappings; replace array values unless a merge flag changes that rule.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-09-001.sh

JYQ-09-002

Compare a shallow mapping merge

A shallow merge replaces the service mapping at the top level. Its name and earlier labels are not carried into the result.

Input

data/base.yaml

```
service:
  name: api
  replicas: 1
  labels: {app: demo}
  ports: [80]
```

data/overlay.yaml

```
service:
  replicas: 3
  labels: {owner: lab}
  ports: [443]
```

Compare with JYQ-09-001 · Deep-merge a configuration overlay.

Code

```
yq -o=json -I=0 '. + load("data/overlay.yaml") | .service' data/base.yaml
```

Actual output

Captured run · exit 0

```
{"replicas":3,"labels":{"owner":"lab"},"ports":[443]}
```

Parameter	Meaning in this example
`+`	Merge only the current mapping level; a matching nested value is replaced.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-09-002.sh

JYQ-09-003

Append sequences during a deep merge

The append flag changes sequence behavior while keeping deep mapping merge. Choose this intentionally instead of assuming all merges concatenate arrays.

Input

data/base.yaml

```
service:
  name: api
  replicas: 1
  labels: {app: demo}
  ports: [80]
```

data/overlay.yaml

```
service:
  replicas: 3
  labels: {owner: lab}
  ports: [443]
```

Compare with JYQ-09-001 · Deep-merge a configuration overlay.

Code

```
yq -o=json -I=0 '. *+ load("data/overlay.yaml") | .service.ports' data/base.yaml
```

Actual output

Captured run · exit 0

```
[80,443]
```

Parameter	Meaning in this example
`*+`	Deep-merge mappings and append sequence values.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-09-003.sh

JYQ-09-004

Merge all input documents explicitly

eval-all makes the documents available together. The accumulator starts with an empty mapping and merges each input in order.

Input

data/base.yaml

```
service:
  name: api
  replicas: 1
  labels: {app: demo}
  ports: [80]
```

data/overlay.yaml

```
service:
  replicas: 3
  labels: {owner: lab}
  ports: [443]
```

Code

```
yq eval-all '. as $item ireduce ({}; . * $item) | .service.replicas' data/base.yaml
data/overlay.yaml
```

Actual output

Captured run · exit 0

```
3
```

Parameter	Meaning in this example
`eval-all and ireduce`	Read every document before reducing them into one result; this is yq syntax, not jq reduce syntax.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-09-004.sh

JYQ-09-005

Select a YAML document by index

The separator begins a second document. documentIndex counts documents from zero, independently of sequence indexes.

Input

data/documents.yaml

```
env: dev
replicas: 1
---
env: prod
replicas: 3
```

Code

```
yq -o=json -I=0 'select(documentIndex == 1)' data/documents.yaml
```

Actual output

Captured run · exit 0

```
{"env":"prod","replicas":3}
```

Parameter	Meaning in this example
`documentIndex`	Select the second YAML document, not the second member of an array.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-09-005.sh

JYQ-09-006

Collect a YAML document stream

The documents are collected into one sequence before serialization. The result is a single JSON array.

Input

data/documents.yaml

```
env: dev
replicas: 1
---
env: prod
replicas: 3
```

Code

```
yq eval-all -o=json -I=0 '[.]' data/documents.yaml
```

Actual output

Captured run · exit 0

```
[{"env":"dev","replicas":1},{"env":"prod","replicas":3}]
```

Parameter	Meaning in this example
`eval-all [.]`	Collect document inputs together rather than emitting separate documents.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-09-006.sh

10 - YAML types, environment values and metadata

YAML carries information beyond JSON values. Check types, preserve strings intentionally and inspect anchors or comments before converting formats.

JYQ-10-001

Distinguish an integer from a quoted string

The visible digits are identical, but the YAML tags differ. A quoted numeric-looking value is a string.

Input

data/types.yaml

```
plain: hello
number: 12
quoted: "12"
enabled: true
nothing: null
word: on
```

Code

```
yq -o=json -I=0 '[(.number | tag), (.quoted | tag)]' data/types.yaml
```

Actual output

Captured run · exit 0

```
["!!int","!!str"]
```

Parameter	Meaning in this example
`tag`	Read the YAML type tag of each selected node.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-10-001.sh

JYQ-10-002

Parse an environment value as YAML

env parses the supplied value as YAML. This controlled numeric value therefore becomes a JSON number.

Code

```
PORT=8088 yq -n -o=json -I=0 '{"port": env(PORT)}'
```

Actual output

Captured run · exit 0

```
{"port":8088}
```

Parameter	Meaning in this example
`env(PORT)`	Parse the explicitly supplied environment variable as a typed YAML node.
`No input file`	The command constructs its input explicitly; the script does not read stdin.

Download: JYQ-10-002.sh

JYQ-10-003

Preserve an environment value as a string

strenv prevents numeric-looking input from changing type. It is useful for identifiers and version strings.

Compare with JYQ-10-002 · Parse an environment value as YAML.

Code

```
PORT=8088 yq -n -o=json -I=0 '{"port": strenv(PORT)}'
```

Actual output

Captured run · exit 0

```
{"port":"8088"}
```

Parameter	Meaning in this example
`strenv(PORT)`	Always construct a string, even when the characters could be parsed as a number.
`No input file`	The command constructs its input explicitly; the script does not read stdin.

Download: JYQ-10-003.sh

JYQ-10-004

Read a line comment

Comments are YAML metadata rather than JSON members. They are not represented in ordinary JSON conversion.

Input

data/settings.yaml

```
service:
  name: api # Public name
  image: demo:1.0
  replicas: 2
  enabled: false
  labels:
    app.kubernetes.io/name: demo
  ports: [8080, 9090]
```

Code

```
yq '.service.name | line_comment' data/settings.yaml
```

Actual output

Captured run · exit 0

```
Public name
```

Parameter	Meaning in this example
`line_comment`	Read the comment associated with the selected scalar node.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-10-004.sh

JYQ-10-005

Inspect a YAML anchor

An anchor gives a YAML node a reusable name. Inspect that metadata separately from its values.

Input

data/aliases.yaml

```
palette: &colors [red, blue]
chart:
  colors: *colors
```

Code

```
yq '.palette | anchor' data/aliases.yaml
```

Actual output

Captured run · exit 0

```
colors
```

Parameter	Meaning in this example
`anchor`	Return the anchor name attached to the selected node.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-10-005.sh

JYQ-10-006

Expand an alias before JSON conversion

explode dereferences aliases and removes anchor names. JSON receives the resulting values, without YAML alias metadata.

Input

data/aliases.yaml

```
palette: &colors [red, blue]
chart:
  colors: *colors
```

Code

```
yq -o=json -I=0 'explode(.) | .chart.colors' data/aliases.yaml
```

Actual output

Captured run · exit 0

```
["red","blue"]
```

Parameter	Meaning in this example
`explode(.)`	Normalize aliases in the selected document before inspecting its values.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-10-006.sh

11 - Convert formats and combine the tools

Use a common JSON representation to compare values. CSV and TSV are serialization formats with their own quoting rules.

JYQ-11-001

Convert YAML into compact JSON

The YAML mapping converts to the same JSON value as the opening example. Comments and YAML-specific presentation are outside this JSON result.

Input

data/intro.yaml

```
project: field-notes
version: 1
active: true
owner: null
```

Compare with JYQ-01-001 · Keep the complete JSON value.

Code

```
yq -o=json -I=0 '.' data/intro.yaml
```

Actual output

Captured run · exit 0

```
{"project":"field-notes","version":1,"active":true,"owner":null}
```

Parameter	Meaning in this example
`-o=json`	Write JSON values rather than YAML presentation.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-11-001.sh

JYQ-11-002

Convert JSON into readable YAML

Declare JSON input and YAML output explicitly. The two selected fields are emitted as a YAML mapping.

Input

data/intro.json

```
{
  "project": "field-notes",
  "version": 1,
  "active": true,
  "owner": null
}
```

Code

```
yq -p=json -oy '{"project": .project, "version": .version}' data/intro.json
```

Actual output

Captured run · exit 0

```
project: field-notes
version: 1
```

Parameter	Meaning in this example
`-p=json -oy`	Parse JSON and serialize YAML; these flags belong to Mike Farah yq.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-11-002.sh

JYQ-11-003

Pass a typed YAML value into jq

The pipe carries JSON, so jq receives a number rather than a YAML-tagged node. YAML metadata is no longer present.

Input

data/types.yaml

```
plain: hello
number: 12
quoted: "12"
enabled: true
nothing: null
word: on
```

Code

```
yq -o=json -I=0 '.number' data/types.yaml | jq -r 'type'
```

Actual output

Captured run · exit 0

```
number
```

Parameter	Meaning in this example
`JSON between tools`	Use explicit JSON serialization when one processor feeds another.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-11-003.sh

JYQ-11-004

Write a tab-separated report

Construct each row as an array before encoding it. The missing-value marker is chosen explicitly for the report.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -r '.samples[] | [.id, .site, (.value // "NA")] | @tsv' data/catalog.json
```

Actual output

Captured run · exit 0

```
A       north   10
B       south   20
C       north   NA
```

Parameter	Meaning in this example
`@tsv`	Encode array members as one tab-separated row; -r emits that row as text.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-11-004.sh

JYQ-11-005

Write a correctly quoted CSV report

CSV quoting is handled by the encoder. Do not imitate CSV by joining strings with commas when values may need quoting.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Compare with JYQ-11-004 · Write a tab-separated report.

Code

```
jq -r '.samples[] | [.id, .site, (.value // "")] | @csv' data/catalog.json
```

Actual output

Captured run · exit 0

```
"A","north",10
"B","south",20
"C","north",""
```

Parameter	Meaning in this example
`@csv`	Encode strings with CSV quoting while retaining numeric fields as numbers.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-11-005.sh

JYQ-11-006

Turn YAML records into a JSON lookup

yq handles YAML input, then jq reshapes the JSON sequence into an ID-to-site mapping. The boundary between processors is explicit.

Input

data/catalog.yaml

```
samples:
  - {id: A, site: north, value: 10, tags: [soil, dry]}
  - {id: B, site: south, value: 20, tags: [water], note: review}
  - {id: C, site: north, value: null, tags: []}
```

Code

```
yq -o=json -I=0 '.samples' data/catalog.yaml | jq -c 'map({key: .id, value: .site}) |
from_entries'
```

Actual output

Captured run · exit 0

```
{"A":"north","B":"south","C":"north"}
```

Parameter	Meaning in this example
`from_entries`	Build an object from records whose members are named key and value.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-11-006.sh

12 - Strings, patterns and explicit ordering

Transform strings as values, not as filter source. Sorting and interpolation should be deliberate parts of an output contract.

JYQ-12-001

Split a string into fields

The delimiter is a literal string. The result is a JSON array rather than a shell word list.

Code

```
jq -n -c '"north-lab" | split("-")'
```

Actual output

Captured run · exit 0

```
["north","lab"]
```

Parameter	Meaning in this example
`split("-")`	Separate one string at each literal hyphen.
`No input file`	The command constructs its input explicitly; the script does not read stdin.

Download: JYQ-12-001.sh

JYQ-12-002

Join an array of strings

Join produces a string value. Raw output makes it readable text instead of a JSON-quoted string.

Code

```
jq -n -r '["soil", "water"] | join(", ")'
```

Actual output

Captured run · exit 0

```
soil, water
```

Parameter	Meaning in this example
`join(", ")`	Combine string array members with a chosen separator.
`No input file`	The command constructs its input explicitly; the script does not read stdin.

Download: JYQ-12-002.sh

JYQ-12-003

Filter IDs with a regular expression

Anchors constrain the pattern to the complete ID. Only A and B match this one-character expression.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -c '[.samples[] | select(.id | test("^[AB]$")) | .id]' data/catalog.json
```

Actual output

Captured run · exit 0

```
["A","B"]
```

Parameter	Meaning in this example
`test("^[AB]$")`	Return a boolean indicating whether the ID matches the regular expression.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-12-003.sh

JYQ-12-004

Replace matching text globally

The semicolon separates function arguments in jq. gsub replaces every matching occurrence.

Code

```
jq -n -r '"north lab" | gsub(" "; "_")'
```

Actual output

Captured run · exit 0

```
north_lab
```

Parameter	Meaning in this example
`gsub(pattern; replacement)`	Replace every regular-expression match in a string.
`No input file`	The command constructs its input explicitly; the script does not read stdin.

Download: JYQ-12-004.sh

JYQ-12-005

Sort YAML mapping keys deliberately

Mapping key order should not be mistaken for record order. Sort keys explicitly when stable presentation is the goal.

Code

```
yq -n -o=json -I=0 '{"b": 2, "a": 1} | sort_keys(.)'
```

Actual output

Captured run · exit 0

```
{"a":1,"b":2}
```

Parameter	Meaning in this example
`sort_keys(.)`	Sort the keys of the selected mapping for predictable presentation.
`No input file`	The command constructs its input explicitly; the script does not read stdin.

Download: JYQ-12-005.sh

JYQ-12-006

Expand a controlled environment placeholder

The single shell quotes preserve the placeholder for yq. Only the environment value supplied on this command is used.

Code

```
GREETING='field team' yq -n '"hello ${GREETING}" | envsubst'
```

Actual output

Captured run · exit 0

```
hello field team
```

Parameter	Meaning in this example
`envsubst`	Interpolate environment placeholders in a string value.
`No input file`	The command constructs its input explicitly; the script does not read stdin.

Download: JYQ-12-006.sh

13 - Reusable jq filters, paths and reductions

Build small reusable filters and address values by explicit paths. Parser streaming is different from ordinary JSON Lines processing.

JYQ-13-001

Define a reusable jq filter

The definition gives a name to a filter, then applies it to each record. The current record remains its input. Avoid reserved words such as label when choosing filter names.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -r 'def sample_label: .id + "@" + .site; .samples[] | sample_label' data/catalog.json
```

Actual output

Captured run · exit 0

```
A@north
B@south
C@north
```

Parameter	Meaning in this example
`def sample_label: ...;`	Define a filter and terminate the definition with a semicolon.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-13-001.sh

JYQ-13-002

Find paths to string values

Paths are arrays of object keys or sequence indexes. The predicate restricts the result to string-valued nodes.

Input

data/intro.json

```
{
  "project": "field-notes",
  "version": 1,
  "active": true,
  "owner": null
}
```

Code

```
jq -c '[paths(type == "string")]' data/intro.json
```

Actual output

Captured run · exit 0

```
[["project"]]
```

Parameter	Meaning in this example
`paths(predicate)`	Emit paths whose values satisfy the predicate.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-13-002.sh

JYQ-13-003

Read a value through an explicit path

A path array can be assembled as data. Numeric path components select array positions; strings select object keys.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -r 'getpath(["samples", 1, "site"])' data/catalog.json
```

Actual output

Captured run · exit 0

```
south
```

Parameter	Meaning in this example
`getpath(path)`	Read the value addressed by a path array.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-13-003.sh

JYQ-13-004

Update a value through a path

The explicit path identifies the same missing measurement updated by the yq selection example. The result is a new JSON value.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Compare with JYQ-08-006 · Update one matching record.

Code

```
jq -c 'setpath(["samples", 2, "value"]; 0) | .samples[2] | {id, value}' data/
catalog.json
```

Actual output

Captured run · exit 0

```
{"id":"C","value":0}
```

Parameter	Meaning in this example
`setpath(path; value)`	Set one path without editing the source file.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-13-004.sh

JYQ-13-005

Accumulate totals into an object

The accumulator begins as an empty object. Each sample adds its contribution to the member named by its site.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Code

```
jq -c 'reduce .samples[] as $sample ({}; .[$sample.site] += ($sample.value // 0))' data/catalog.json
```

Actual output

Captured run · exit 0

```
{"north":10,"south":20}
```

Parameter	Meaning in this example
`reduce stream as $item (initial; update)`	Carry an accumulator through the input results; jq syntax differs from yq ireduce.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-13-005.sh

JYQ-13-006

Inspect parser-stream leaf paths

Parser streaming emits path-value events and closing markers. Selecting two-element events keeps leaf values; this is a different interface from normal JSON records.

Input

data/intro.json

```
{
  "project": "field-notes",
  "version": 1,
  "active": true,
  "owner": null
}
```

Code

```
jq --stream -c 'select(length == 2) | .[0]' data/intro.json
```

Actual output

Captured run · exit 0

```
["project"]
["version"]
["active"]
["owner"]
```

Parameter	Meaning in this example
`--stream`	Expose parser events instead of constructing the entire input tree; the shown filter emits only leaf paths.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-13-006.sh

14 - Complete reporting and configuration recipes

Combine the earlier operations into small reproducible tasks. All edits happen in a temporary directory created by the downloadable script.

JYQ-14-001

Create a grouped measurement report

The report distinguishes record count from measured total. Missing measurements contribute zero to this explicitly chosen total, while their records are still counted.

Input

data/catalog.json

```
{
  "samples": [
    {"id": "A", "site": "north", "value": 10, "tags": ["soil", "dry"]},
    {"id": "B", "site": "south", "value": 20, "tags": ["water"], "note": "review"},
    {"id": "C", "site": "north", "value": null, "tags": []}
  ]
}
```

Compare with JYQ-04-002 · Count records by site.

Code

```
jq -c '.samples | group_by(.site) | map({site: .[0].site, count: length, total:
(map(.value // 0) | add)})' data/catalog.json
```

Actual output

Captured run · exit 0

```
[{"site":"north","count":2,"total":10},{"site":"south","count":1,"total":20}]
```

Parameter	Meaning in this example
`group_by plus map`	Compute separate summaries inside each site group.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-14-001.sh

JYQ-14-002

Produce the same site counts from YAML

Repeat the earlier grouped count against the equivalent YAML input. The expected JSON value is identical.

Input

data/catalog.yaml

```
samples:
  - {id: A, site: north, value: 10, tags: [soil, dry]}
  - {id: B, site: south, value: 20, tags: [water], note: review}
  - {id: C, site: north, value: null, tags: []}
```

Compare with JYQ-04-002 · Count records by site.

Code

```
yq -o=json -I=0 '.samples | group_by(.site) | map({"site": .[0].site, "count":
length})' data/catalog.yaml
```

Actual output

Captured run · exit 0

```
[{"site":"north","count":2},{"site":"south","count":1}]
```

Parameter	Meaning in this example
`group_by and explicit keys`	Group records, then construct a named summary mapping for each group.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-14-002.sh

JYQ-14-003

Set a configuration string from the environment

The image reference is supplied as a string value. Shell interpolation is not used to build a yq expression.

Input

data/settings.yaml

```
service:
  name: api # Public name
  image: demo:1.0
  replicas: 2
  enabled: false
  labels:
    app.kubernetes.io/name: demo
  ports: [8080, 9090]
```

Code

```
IMAGE_TAG='demo:2.0' yq '.service.image = strenv(IMAGE_TAG) | .service.image' data/
settings.yaml
```

Actual output

Captured run · exit 0

```
demo:2.0
```

Parameter	Meaning in this example
`strenv(IMAGE_TAG)`	Keep the complete image reference as a string, including its colon.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-14-003.sh

JYQ-14-004

Edit an owned copy in place

Copy the fixture before requesting an in-place edit. The standalone script owns working.yaml inside its temporary directory, so the canonical fixture is untouched.

Input

data/settings.yaml

```
service:
  name: api # Public name
  image: demo:1.0
  replicas: 2
  enabled: false
  labels:
    app.kubernetes.io/name: demo
  ports: [8080, 9090]
```

Code

```
cp data/settings.yaml working.yaml
yq -i '.service.replicas = 4' working.yaml
yq '.service.replicas' working.yaml
```

Actual output

Captured run · exit 0

```
4
```

Parameter	Meaning in this example
`-i`	Write the updated document back to the specified owned copy.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-14-004.sh

JYQ-14-005

Construct JSON without shell-quoting surprises

The quoted text is passed as data through --arg. jq performs JSON escaping when it serializes the constructed object.

Code

```
jq -n -c --arg project 'north "lab"' '{project: $project, enabled: false}'
```

Actual output

Captured run · exit 0

```
{"project":"north \"lab\"","enabled":false}
```

Parameter	Meaning in this example
`--arg with -n`	Create a JSON value from explicit arguments without needing an input document.
`No input file`	The command constructs its input explicitly; the script does not read stdin.

Download: JYQ-14-005.sh

JYQ-14-006

Validate a YAML configuration with an exit status

The example configuration is disabled, so this deliberately strict condition fails. The boolean result and caught process status are both shown.

Input

data/settings.yaml

```
service:
  name: api # Public name
  image: demo:1.0
  replicas: 2
  enabled: false
  labels:
    app.kubernetes.io/name: demo
  ports: [8080, 9090]
```

Code

```
if yq -e '.service.replicas > 0 and .service.enabled == true' data/settings.yaml >
result.txt 2> diagnostic.txt; then
  status=0
else
  status=$?
fi
cat result.txt
printf 'exit=%s\n' "$status"
```

Actual output

Captured run · exit 0

```
false
exit=1
```

Parameter	Meaning in this example
`-e`	Ask yq to signal a false, null or missing match through a nonzero status.
`Local input`	The complete fixture shown above is recreated by the standalone script.

Download: JYQ-14-006.sh

Example and command index

- JYQ-01-001 · Keep the complete JSON value
- JYQ-01-002 · Select one string as JSON
- JYQ-01-003 · Print the same string without JSON quotes
- JYQ-01-004 · Build a smaller object
- JYQ-01-005 · Count array elements
- JYQ-01-006 · Read an array element
- JYQ-02-001 · Emit one result per record
- JYQ-02-002 · Collect IDs into one array
- JYQ-02-003 · Keep records from one site
- JYQ-02-004 · Filter numbers while excluding null
- JYQ-02-005 · Test whether a key exists
- JYQ-02-006 · Flatten optional nested arrays
- JYQ-03-001 · Map records to IDs
- JYQ-03-002 · Replace null measurements with zero
- JYQ-03-003 · Remove null measurements from an array
- JYQ-03-004 · Update a number relative to itself
- JYQ-03-005 · Delete a member and inspect the keys
- JYQ-03-006 · Label missing observations
- JYQ-04-001 · Sum observed measurements
- JYQ-04-002 · Count records by site
- JYQ-04-003 · Sort records by measurement
- JYQ-04-004 · Find distinct site names
- JYQ-04-005 · Build an ID lookup
- JYQ-04-006 · Expose object entries as records
- JYQ-05-001 · Pass a string argument safely
- JYQ-05-002 · Pass a numeric argument
- JYQ-05-003 · Load a second JSON file
- JYQ-05-004 · Process JSON Lines one value at a time
- JYQ-05-005 · Collect a JSON stream before aggregation
- JYQ-05-006 · Read remaining inputs from a null-input program
- JYQ-06-001 · Distinguish missing membership from null
- JYQ-06-002 · Remember that fallback also replaces false
- JYQ-06-003 · Validate a condition across all records
- JYQ-06-004 · Inspect the exit status of a null result
- JYQ-06-005 · Catch a conversion error
- JYQ-06-006 · Reject malformed JSON
- JYQ-07-001 · Read a YAML string
- JYQ-07-002 · Construct a YAML-derived JSON object
- JYQ-07-003 · Index a YAML sequence
- JYQ-07-004 · Count a YAML sequence
- JYQ-07-005 · Collect YAML IDs as JSON
- JYQ-07-006 · Select the same site in YAML
- JYQ-08-001 · Assign a configuration value
- JYQ-08-002 · Increment a YAML number
- JYQ-08-003 · Remove a YAML mapping member
- JYQ-08-004 · Read a key that contains dots
- JYQ-08-005 · Append a port to a sequence
- JYQ-08-006 · Update one matching record
- JYQ-09-001 · Deep-merge a configuration overlay
- JYQ-09-002 · Compare a shallow mapping merge
- JYQ-09-003 · Append sequences during a deep merge
- JYQ-09-004 · Merge all input documents explicitly
- JYQ-09-005 · Select a YAML document by index
- JYQ-09-006 · Collect a YAML document stream

- JYQ-10-001 · Distinguish an integer from a quoted string
- JYQ-10-002 · Parse an environment value as YAML
- JYQ-10-003 · Preserve an environment value as a string
- JYQ-10-004 · Read a line comment
- JYQ-10-005 · Inspect a YAML anchor
- JYQ-10-006 · Expand an alias before JSON conversion
- JYQ-11-001 · Convert YAML into compact JSON
- JYQ-11-002 · Convert JSON into readable YAML
- JYQ-11-003 · Pass a typed YAML value into jq
- JYQ-11-004 · Write a tab-separated report
- JYQ-11-005 · Write a correctly quoted CSV report
- JYQ-11-006 · Turn YAML records into a JSON lookup
- JYQ-12-001 · Split a string into fields
- JYQ-12-002 · Join an array of strings
- JYQ-12-003 · Filter IDs with a regular expression
- JYQ-12-004 · Replace matching text globally
- JYQ-12-005 · Sort YAML mapping keys deliberately
- JYQ-12-006 · Expand a controlled environment placeholder
- JYQ-13-001 · Define a reusable jq filter
- JYQ-13-002 · Find paths to string values
- JYQ-13-003 · Read a value through an explicit path
- JYQ-13-004 · Update a value through a path
- JYQ-13-005 · Accumulate totals into an object
- JYQ-13-006 · Inspect parser-stream leaf paths
- JYQ-14-001 · Create a grouped measurement report
- JYQ-14-002 · Produce the same site counts from YAML
- JYQ-14-003 · Set a configuration string from the environment
- JYQ-14-004 · Edit an owned copy in place
- JYQ-14-005 · Construct JSON without shell-quoting surprises
- JYQ-14-006 · Validate a YAML configuration with an exit status

---

Content: CC0 1.0. Structure and publishing framework: CC BY 4.0 by Lukas C. Bossert. Repository source · Publishing framework source · License scope and third-party notices.