kaiv for JSON Users

A Cookbook

The kaiv Project

for kaiv 0.4.0

1 Introduction

You know JSON. This cookbook maps what you know onto **kaiv** — a text format in which every line carries its own type, its own address, and its own value — and shows what you gain on the way: schemas inferred from examples, validation errors that name the field and the constraint, physical units, per-value provenance, and a canonical form so regular that `grep` becomes a typed query tool.

kaiv covers JSON completely: every JSON document converts to kaiv, every kaiv document converts back, and the round trip is lossless. Your data does not migrate anywhere — kaiv is a hub. What comes in from JSON can leave as XML, CBOR, TOML, Avro, Protocol Buffers, or ASN.1 DER, and the same canonical form is what schemas validate and tools query. JSON stays your wire format for exactly as long as you want it to.

Every example in this cookbook is a verified transcript of the reference `kaiv` tool (`cargo install kaiv-cli`), run against the fixture files shipped beside this document.

2 Hello, kaiv

A JSON configuration file, the kind you have written a hundred times:

```
{
  "server": {
    "host": "api.example.com",
    "port": 8443,
    "tls": true,
```

```
    "motd": null,
    "tags": ["prod", "eu"]
  }
}
```

Import it:

```
$ kaiv import config.json
.!kaiv 1

/server::host=api.example.com
!int
/server::port=8443
!bool
/server::tls=true
!null
/server::motd=
/server/@tags;=prod;eu
```

This is *authored* kaiv (a `.kaiv` file): the form a person writes. Nesting became paths — `/server` descends into a namespace the way a directory path does, and `::` projects a field out of it, the way a filename ends a path. A metadata line like `!int` annotates the line below it. Strings need no quotes and no escaping; the default type is `str`, so only the non-strings say their type.

Now build it — kaiv's equivalent of compiling:

```
$ kaiv import config.json | kaiv build
.!kaiv 1
!str'/server::host=api.example.com
!int'/server::port=8443
!bool'/server::tls=true
!null'/server::motd=
!str'/server/@tags::0=prod
!str'/server/@tags::1=eu
```

This is *canonical* kaiv (a `.daiv` file), and it is the heart of the format. One line per field. Each line is completely self-contained:

`!int`	`'`	`/server::port`	`=`	`8443`
type	delimiter	address	operator	value

There is no nesting, no braces, no indentation — structure lives in the address, not in the syntax. A line can be read, moved, or diffed in isolation. The only structural characters on a canonical line are `'` and `=`.

3 The Mapping

Everything JSON has, kaiv has:

JSON	kaiv	
object	namespace	`/server::host=...`
array	`@`-array	`/server/@tags::0=prod`
nested object	deeper path	`/a/b/c::d=...`
string	`str` (the default)	`host=api.example.com`
number	`!int` / `!float`	`!int` then `port=8443`
boolean	`!bool`	`!bool` then `tls=true`
null	`!null`	`!null` then `motd=`
object as dictionary	`:=` map	`/limits:=alice=10|bob=20`
mixed-type array	tagged union	`!int|str`

Dynamic-key objects — JSON objects used as dictionaries — import as kaiv maps and land in canonical form as ordinary fields:

```
$ printf '{"limits":{"alice":10,"bob":20}}' | kaiv import
   ↪  --json | kaiv build
.!kaiv 1
!int'/limits::alice=10
!int'/limits::bob=20
```

4 The Round Trip

Nothing is lost in either direction. Build the canonical form once, then export:

```
$ kaiv import config.json | kaiv build > config.daiv
$ kaiv export --json config.daiv
{"server":{"host":"api.example.com","port":8443,"tls":tru⌟
   ↪  e,"motd":null,"tags":["prod","eu"]}}
```

The export is compact JSON, semantically identical to the source — objects, arrays, numbers, booleans, and null all survive. Round-tripping is a hard contract in kaiv's converter suite, held by property-based tests in the reference implementation.

5 Null Is a Type, Not a Hole

In JSON, `null` and `""` both render as "nothing there," and every consumer re-derives which one it meant. In kaiv the two are different *types* with the same empty payload:

```
$ printf '{"nickname":null,"bio":""}' | kaiv import --json
   ↪  | kaiv build
.!kaiv 1
!null'::nickname=
!str'::bio=
```

The annotation is the discriminant. A schema can require one, the other, or admit both (`!null|str` — a tagged union, kaiv's version of `["null", "string"]`).

6 Every Line Answers for Itself

Because each canonical line carries its type and full address, the humblest Unix tools become structured queries. Find every integer in the document:

```
$ grep '^!int' config.daiv
!int'/server::port=8443
```

No parser, no jq expression — the line format *is* the index. This property is what kaiv's query language (qaiv) and its certifiable constant-memory validator are built on.

7 A Schema from Your Example

JSON gives you no schema language; JSON Schema is a separate standard you write by hand. kaiv infers a schema from the document you already have:

```
$ kaiv infer --name acme/server config.json
.!kaivschema 1 acme/server
```

```
/server::host=
!int
/server::port=
!bool
/server::tls=
!null|str
/server::motd=
/server/@tags;=
```

Note `!null|str` for `motd`: the inferrer saw `null` and widened to "null or string" rather than locking the field to null forever. Schemas are written in the same line grammar as data — a schema is data about data.

Save it, then tighten what inference cannot know — that a port must fit in sixteen bits. The change is one constraint, `[1,65535]`, on the type annotation:

```
$ kaiv infer --name acme/server config.json > server.saiv
$ sed -i 's/^!int$/!int[1,65535]/' server.saiv
$ kaiv validate config.daiv server.saiv
pass
```

And when data breaks the contract — `bad.json` is the same document with `"port": 99999` — the error names the field, the value, the violated constraint, and the line:

```
$ kaiv import bad.json | kaiv build > bad.daiv
$ kaiv validate bad.daiv server.saiv
kaiv: ConstraintViolationError: /server::port=99999 (type
   ↪  !int) violates /^-?[0-9]+$/ ..num [1,65535] (line
   ↪  3)
```

The constraint shown is the *compiled* form: the integer pattern, the numeric ordering, and your range, exactly as the validator checked them. Exit code 1; `pass` and exit 0 otherwise.

8 Your JSON Schema Comes Along

Already have JSON Schema? It converts. The contract is a *sound weakening*: every constraint kaiv emits is implied by your schema, and anything kaiv cannot express is dropped with a comment — never invented, never silently ignored.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "host": { "type": "string", "format": "hostname" },
    "port": { "type": "integer", "minimum": 1, "maximum": 65535
       ↪  },
    "proto": { "enum": ["http", "https"] }
  },
  "required": ["host", "port"]
}
```

```
$ kaiv import-schema --json --name acme/endpoint
   ↪  endpoint.schema.json
.!kaivschema 1 acme/endpoint
.!types std/net

&hostname
host=
!int[1,65535]
port=
!null|str{http,https}
proto?=
```

Everything survived: the range as `[1,65535]`, the enum as an enumeration constraint — and `format: hostname` became `&hostname` from the `std/net` library, an RFC 1123 pattern with teeth where JSON Schema's `format` is, per spec, only an annotation most validators ignore. (Formats kaiv has no type for are dropped with a `// dropped:` comment you can act on — never silently.) `proto?=` marks the field optional, with a null alternative so absent instances materialize cleanly.

The converted schema validates real data immediately:

```
$ kaiv import-schema --json --name acme/endpoint
   ↪  endpoint.schema.json > endpoint.saiv
$ printf '{"host":"api.example.com","port":8443,"proto":"」
   ↪  https"}' | kaiv import --json | kaiv build >
   ↪  endpoint.daiv
$ kaiv validate endpoint.daiv endpoint.saiv
pass
$ printf '{"host":"api.example.com","port":8443,"proto":"」
   ↪  ftp"}' | kaiv import --json | kaiv build >
   ↪  badproto.daiv
```

```
$ kaiv validate badproto.daiv endpoint.saiv
kaiv: ConstraintViolationError: ::proto=ftp (type !str)
   ↪  violates !null(/^$/)|str({http,https}) (line 4)
```

9 The Hub

The canonical form you built from JSON is not a JSON artifact — it is format-neutral. The same `config.daiv` leaves as XML:

```
$ kaiv export --xml config.daiv
<?xml version="1.0" encoding="UTF-8"?>
<server>
  <host>api.example.com</host>
  <port>8443</port>
  <tls>true</tls>
  <motd/>
  <tags>prod</tags>
  <tags>eu</tags>
</server>
```

As binary CBOR — and back, closing the loop bit-for-bit:

```
$ kaiv export --cbor config.daiv | kaiv import --cbor |
   ↪  kaiv build
.!kaiv 1
!str'/server::host=api.example.com
!int'/server::port=8443
!bool'/server::tls=true
!null'/server::motd=
!str'/server/@tags::0=prod
!str'/server/@tags::1=eu
```

And when a target format *cannot* carry your data, kaiv refuses loudly instead of corrupting quietly — TOML has no null:

```
$ kaiv export --toml config.daiv
kaiv: TOML cannot represent null
```

Fidelity is never silent in kaiv: what maps, maps exactly; what cannot map is an error or a documented drop. Avro, Protocol Buffers, ASN.1 DER, and YAML travel the same roads — each has its own cookbook in this series.

10 What JSON Cannot Say

Two kaiv capabilities have no JSON equivalent at all. Values can carry *units* — part of the type, byte-compared, never silently converted:

```
$ printf '.!kaiv 1\n\n!float:km/h\n/car::speed=120.5\n' |
   ↪  kaiv build
.!kaiv 1
!float:km/h'/car::speed=120.5
```

And every value can carry *provenance* — who says so, and when — with sources declared once and referenced per-line:

```
$ printf '.!kaiv 1\n.?gps
   ↪  https://fleet.example.com/gps-17\n\n!float:km/h?gp⌋
   ↪  s@20260717T120000Z\n/car::speed=120.5\n' | kaiv
   ↪  build
.!kaiv 1
.?gps https://fleet.example.com/gps-17
!float:km/h?gps@20260717T120000Z'/car::speed=120.5
```

A schema can *require* provenance on every line — compliance-grade lineage without an external metadata system. With type, unit, source, timestamp, and address, a single canonical line carries five attributions; JSON carries one (the address, and only implicitly).

11 When to Stay with JSON

An honest map marks the roads not taken. JSON remains the right choice when:

- you need browser-native parsing — `JSON.parse` ships in every JavaScript engine; kaiv does not;
- your toolchain depends on JSON's universal ecosystem (every language, every service, every editor);
- your documents are transient API payloads with no schema, no audit requirements, and no lifetime.

kaiv's ecosystem is young. The pull toward the hub grows with what you need *around* the data: types checked at build time, schemas that travel, units, provenance, deterministic canonical form, validation certifiable for safety-critical runtimes — and conversion back out at any moment, so the door never locks behind you.

12 Where Next

The type libraries kaiv ships with are published at the registry addresses the toolchain resolves by default (the `kaiv.io` alpha hosts; the permanent `ktaiv.com` eternalinks activate at beta):

```
$ curl -s https://t.kaiv.io/std/core.taiv | head -3
.!kaivtype 1 std/core

// Integer — decimal string with numeric ordering
```

From here:

- the kaiv specification — the formal grammar and semantics;
- `kaiv help` — the full toolchain surface: `import`, `export`, `build`, `schema`, `validate`, `infer`, `import-schema`;
- the sibling cookbooks — kaiv for YAML, TOML, XML, Protocol Buffers, Avro, CBOR, ASN.1, and GraphQL users.