kaiv for XML Users

A Cookbook

The kaiv Project

for kaiv 0.4.0

1 Introduction

You work with XML — for interchange, for B2B documents, for regulatory filings, for configuration that outlived three rewrites. This cookbook maps XML and XSD onto **kaiv**: a format with XML's discipline (schemas, constraints, explicit structure) and none of its weight — no angle brackets, no escaping, no attribute-versus-element doctrine, and validation simple enough for a deterministic finite automaton.

kaiv covers data-interchange XML completely: elements, attributes, repeated siblings, empty elements, namespace prefixes, and the XSD constraint vocabulary all map. Conversion is a hub, not a migration — what comes in from XML can leave as JSON, YAML, CBOR, or TOML, and XML remains your wire format for as long as the other side requires it.

Every example 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 document exercising the XML fundamentals — attributes, nested elements, repeated siblings, an empty element:

```
<?xml version="1.0" encoding="UTF-8"?>
<library name="Central">
  <book isbn="978-0261103252">
    <title>The Fellowship of the Ring</title>
    <pages>531</pages>
  </book>
  <book isbn="978-0261102361">
```

```
    <title>The Two Towers</title>
    <pages>352</pages>
  </book>
  <closed/>
</library>
```

Build it to canonical form:

```
$ kaiv import library.xml | kaiv build
.!kaiv 1
!str'/library::"@name"=Central
!str'/library/@book/0::"@isbn"=978-0261103252
!str'/library/@book/0::title=The Fellowship of the Ring
!str'/library/@book/0::pages=531
!str'/library/@book/1::"@isbn"=978-0261102361
!str'/library/@book/1::title=The Two Towers
!str'/library/@book/1::pages=352
!null'/library::closed=
```

One line per leaf: `!type ' address = value`. Reading the mapping off the transcript: the root element is the top namespace; *attributes and elements land in the same field space* — an attribute keeps an `@` marker in its quoted name (`"@isbn"`) and needs no doctrine debate; repeated `<book>` siblings became the indexed array `@book`; the empty element became an explicit `!null`. Element text stays untyped (`pages=531` is a string) — XML carries no type information, and kaiv does not guess; the schema, converted below, is what types it.

Note what is absent: no escaping. `&amp;`, `&lt;`, CDATA — none of it exists in kaiv; values are verbatim bytes to the end of the line.

3 The Mapping

XML / XSD	kaiv	
root element	top namespace	`/library...`
nested element	path segment	`/library/@book/0::title`
attribute	`"@name"` field	same field space as elements
repeated siblings	`@`-array	`/@book/0`, `/@book/1`
empty element	`!null`	`!null` then `closed=`
text content	`str` field	untyped without a schema
mixed content	typed embed channel	`&xml` + base64url
namespace prefixes	quoted verbatim	`/"soap:Envelope"...`
`xs:sequence`	ordered fields	kaiv fields are ordered
`xs:choice`	optional fields + note	exclusivity noted
`xs:restriction` facets	constraints	`[1,9999]`, `{a,b}`, `/re/`
`xs:dateTime` etc.	`std/time` types	`&datetime`, `&date`, `&time`

Two of those deserve their own transcripts. Namespace prefixes stay verbatim — `soap:Body` is a literal name, quoted:

```
$ printf '<soap:Envelope><soap:Body>ok</soap:Body></soap:
   ↪  Envelope>' | kaiv import --xml
.!kaiv 1

/"soap:Envelope"::"soap:Body"=ok
```

And mixed content — markup interleaved with text, the one XML shape with no flat equivalent — rides the typed embed channel, lossless and declared:

```
$ printf '<note>Call <b>Ada</b> today</note>' | kaiv
   ↪  import --xml
.!kaiv 1
.!types std/enc

&xml
note=PG5vdGU-Q2FsbCA8Yj5BZGE8L2I-IHRvZGF5PC9ub3RlPg
```

The value is the original XML fragment, base64url-armored and typed `&xml` from `std/enc`; it splices back verbatim on export. Where deep mixed content is the document — DocBook, XHTML prose — XML remains the better home.

4 The Round Trip

Back out to XML, attributes and structure intact:

```
$ kaiv import library.xml | kaiv build > library.daiv
$ kaiv export --xml library.daiv
<?xml version="1.0" encoding="UTF-8"?>
<library name="Central">
  <book isbn="978-0261103252">
    <title>The Fellowship of the Ring</title>
    <pages>531</pages>
  </book>
  <book isbn="978-0261102361">
    <title>The Two Towers</title>
    <pages>352</pages>
  </book>
  <closed/>
</library>
```

The `"@name"` convention is what makes this possible: attribute-ness survives the flat form, so the exporter knows exactly what to reconstruct.

5 Your XSD Comes Along

The real asset in an XML shop is the schemas. They convert — under the sound-weakening contract: every emitted constraint is implied by the source, everything else drops with a comment, nothing is invented.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="book">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="title" type="xs:string"/>
        <xs:element name="pages">
          <xs:simpleType>
            <xs:restriction base="xs:positiveInteger">
              <xs:maxInclusive value="9999"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="genre" minOccurs="0">
          <xs:simpleType>
```

```
            <xs:restriction base="xs:string">
              <xs:enumeration value="fantasy"/>
              <xs:enumeration value="scifi"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="isbn" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
$ kaiv import-schema --xsd --name acme/book book.xsd
.!kaivschema 1 acme/book

/book::"@isbn"=
/book::title=
!int[1,9999]
/book::pages=
!null|str{fantasy,scifi}
/book::genre?=
```

Thirty lines of XSD became six lines of kaiv, with nothing of substance lost. Follow one field through: an `xs:restriction` on `xs:positiveInteger` with `maxInclusive 9999` became `!int[1,9999]` — the *base type's* implicit lower bound and the facet's upper bound, merged. The enumeration became `{fantasy,scifi}`; `minOccurs="0"` became the `?=` optional marker; `use="required"` left `"@isbn"` required.

The converted schema validates XML-imported data directly — same shapes, same root:

```
$ kaiv import-schema --xsd --name acme/book book.xsd >
   ↪  book.saiv
$ printf '<book isbn="978-0261103283"><title>The
   ↪  Hobbit</title><pages>310</pages><genre>fantasy</ge⌋
   ↪  nre></book>' | kaiv import --xml | kaiv build >
   ↪  hobbit.daiv
$ kaiv validate hobbit.daiv book.saiv
pass
$ printf '<book isbn="978-0261103283"><title>The
   ↪  Hobbit</title><pages>0</pages><genre>fantasy</genr⌋
   ↪  e></book>' | kaiv import --xml | kaiv build >
   ↪  zero.daiv
```

```
$ kaiv validate zero.daiv book.saiv
kaiv: ConstraintViolationError: /book::pages=0 (type !str)
   ↪  violates /^-?[0-9]+$/ ..num [1,9999] (line 4)
```

`pages=0` fails on exactly the bound that `xs:positiveInteger` implied — field, value, compiled constraint, and line, from a validator that is a single forward scan. Exit code 1; `pass` and exit 0 otherwise.

6 What the Conversion Buys

The same discipline, a fraction of the machinery:

- *Validation in constant memory.* An XSD validator is a heap-allocating tree walker; kaiv's validator is a DFA-driven forward scan — certifiable for runtimes where no XML parser has ever been admitted.
- *One syntax for data and schema.* A `.saiv` schema is written in the same line grammar as the data it constrains — no second language, no `xs:` namespace within a namespace.
- *Facets without ceremony.* Two elements and two attributes of `xs:restriction` collapse to `[1,9999]`.
- *The hub.* The same `library.daiv` exports to JSON, YAML, CBOR, TOML — see the sibling cookbooks; your XML partners keep receiving XML.

7 When to Stay with XML

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

- the document *is* prose — deep mixed content (DocBook, XHTML, TEI) is XML's native shape and only armors in kaiv;
- your counterparties mandate it — regulatory and B2B pipelines (UBL, HL7, XBRL) specify XML on the wire, though kaiv can be the processing form behind the boundary;

- you depend on XSLT, XPath axes over document order, or DTD entities — transformation machinery kaiv does not reproduce.

kaiv's pull grows where XML is used as a *data* format: configuration, records, interchange — the places where the brackets outweigh the prose and the schema is the point.

8 Where Next

From here:

- the kaiv specification — the formal grammar and semantics;
- `kaiv help` — the full toolchain surface;
- the sibling cookbooks — kaiv for JSON, YAML, TOML, Protocol Buffers, Avro, CBOR, ASN.1, and GraphQL users.