kaiv for GraphQL Users

A Guide

The kaiv Project

for kaiv 0.5.0

1 Introduction

You think in SDL — types, non-null bangs, enums, a schema that fronts every byte your API serves. This guide maps GraphQL's schema language onto **kaiv**, and fills the gap SDL leaves open: GraphQL has no *data format*. Your types constrain responses in flight, and nothing else — the JSON on disk, the fixture in the repo, the export in the bucket are all untyped the moment they leave the server.

kaiv gives an SDL-shaped contract a life outside the API: the schema converts, documents validate against it at rest, and the same canonical form exports to JSON, YAML, XML, CBOR, TOML, and the binary schema formats. GraphQL keeps serving queries; kaiv keeps the data honest everywhere else.

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 Your SDL Converts

A schema with the usual shapes — non-null scalars, an enum, a nested object type, a list, a nullable field:

```
enum Status {
  DRAFT
  PUBLISHED
}

type Author {
  name: String!
```

```
  handle: String!
}

type Post {
  title: String!
  words: Int!
  status: Status!
  author: Author!
  tags: [String!]!
  summary: String
}
```

Convert it, picking the root type:

```
$ kaiv import-schema --graphql --name acme/post --message
   ↪  Post post.graphql
.!saiv 1 acme/post

title=
!int[-2147483648,2147483647]
words=
!str{DRAFT,PUBLISHED}
status=
/author::name=
/author::handle=
/@tags;=
!null|str
summary?=
```

Reading it back: GraphQL `Int` is specified as signed 32-bit, and the conversion says so *as a value constraint* — the exact range, on the line. The enum became an enumeration with teeth. Non-null `!` became required (kaiv's default); nullable `summary` became optional with an explicit `!null` alternative. The `Author` object flattened to namespace fields; the list became a vector. Nothing to unlearn — the conversion reads like the SDL, minus the braces.

3 Types for Data at Rest

Now the part GraphQL cannot do. Author a document — or receive one as API-response JSON — and hold it to the contract *off* the wire:

```
$ kaiv import-schema --graphql --name acme/post --message
   ↪  Post post.graphql > post.saiv
```

```
$ printf '{"title":"The Hub","words":900,"status":"PUBLIS⌋
   ↪  HED","author":{"name":"Ada","handle":"ada"},"tags"⌋
   ↪  :["kaiv"],"summary":null}' | kaiv import --json |
   ↪  kaiv build > post.daiv
$ kaiv validate post.daiv post.saiv
pass
$ printf '{"title":"The Hub","words":900,"status":"ARCHIV⌋
   ↪  ED","author":{"name":"Ada","handle":"ada"},"tags":⌋
   ↪  ["kaiv"],"summary":null}' | kaiv import --json |
   ↪  kaiv build > arch.daiv
$ kaiv validate arch.daiv post.saiv
kaiv: ConstraintViolationError: ::status=ARCHIVED (type
   ↪  !str) violates {DRAFT,PUBLISHED} (line 4)
```

Field, value, the violated enumeration, and the line — on a fixture file, a cached response, an export; no server in sight. Exit code 1; `pass` and exit 0 otherwise.

One equivalence worth naming: GraphQL types are *implicitly* strict — a response never carries fields the type does not declare. A kaiv `strict` schema is the same assertion made explicit, and the converted schema can adopt it by adding `strict` to its header line.

4 The Mapping

GraphQL SDL	kaiv	
`type`	namespace / document	
nested object type	namespace fields	`/author::name=`
`String!`	required `str`	requiredness is the default
`String`	`!null|str` optional	absence explicit
`Int`	`!int` + 32-bit range	per the GraphQL spec
`enum`	enum constraint	`{DRAFT,PUBLISHED}`
`[String!]!`	`;=` vector	
`union`	tagged union	over types, not object types
`interface`	schema composition	`allOf`-style
`"""docs"""`	`//` doc comments	
`@deprecated`	—	mapping documents instead

And beyond the mapping, the kaiv side adds what SDL has no syntax for: value patterns, numeric ranges of your choosing, length bounds, units, per-value provenance — see the sibling guides for the machinery.

5 When to Stay with GraphQL

An honest map marks the roads not taken. GraphQL remains the right choice — indeed, has no kaiv equivalent — for:

- query execution: resolvers, client-driven field selection, batching; kaiv validates data, it does not serve it;
- subscriptions and live data;
- introspection-driven tooling against a running endpoint.

kaiv is not a GraphQL replacement — it is where your SDL's discipline extends when the data leaves the endpoint: fixtures, seeds, caches, exports, archives, and every file a contract should cover but a resolver never sees.

6 Where Next

From here:

- the kaiv specification — the formal grammar and semantics;
- `kaiv help` — the full toolchain surface;
- the sibling guides — kaiv for JSON, YAML, TOML, XML, Protocol Buffers, Avro, CBOR, and ASN.1 users; the JSON guide shows the schema-inference and JSON Schema paths that complement this one.