Schema and select

Alex Miller edited this page Dec 17, 2019 · 15 revisions

This is an overview of the new schema and select in spec 2. Everything is subject to change. For other differences from spec.alpha that are not covered here, see: https://github.com/clojure/spec-alpha2/wiki/Differences-from-spec.alpha.

Background

It is probably useful to watch "Maybe Not" by Rich Hickey to get a deeper look at some of the issues from spec.alpha we are trying to resolve.

All examples below assume spec namespaces have been loaded:

```
(require '[clojure.alpha.spec :as s] '[clojure.alpha.spec.gen :as gen])
```

Schemas

It is common to have aggregates of keys that work together to describe the attributes of an entity (User, Company, Order, etc). In spec.alpha, `s/keys` was used to define required and optional attributes. In spec 2, the optionality of these attributes comes out of the selection and *schemas* are used to aggregate attributes into groups that "travel together" to describe one thing. Schemas do not define "required" or "optional" - it's up to selections to declare this in a context.

Schemas have both a symbolic form, and an object form (same as specs). Schemas are also

named with fully-qualified keywords and schema objects are managed at runtime in the registry. Schemas are also specs and can be used as such in the API. In spec 1 terms, they are similar to an s/keys with only :opt or :opt-un keys (no :req or :req-un).

Literal schemas

The simplest form of a literal schema is a vector of qualified keywords, which should refer to specs in the registry.

```
[::address/street ::address/city ::address/state ::address/zip]
```

Additionally, you may include a map of unqualified keys to the specs or spec names that should be used for them:

```
[::a ::b {:c (s/spec int?)}]
```

Typically you will either see all qualified keywords (use the vector form) or all unqualified keywords (when spec'ing JSON, etc). For the latter case, you can just use a map:

```
{:first ::domain/fname, :last ::domain/lname}
```

Schema forms

The above literal forms can be used directly in a spec (like `s/select`), but must be wrapped in an `s/schema` form to serve as a symbolic schema (similar to the use of `s/spec` for wrapping a symbolic predicate).

For example, to define and store schemas in the registry:

```
(s/def ::street string?)
(s/def ::city string?)
(s/def ::state string?) ;; simplified
(s/def ::zip int?) ;; simplified
(s/def ::addr (s/schema [::street ::city ::state ::zip]))

(s/def ::id int?)
(s/def ::first string?)
(s/def ::last string?)
(s/def ::user (s/schema [::id ::first ::last ::addr]))
```

Unions

In addition to `s/schema`, you can use `s/union` to combine schemas, which are either schema names, literal schemas, or schema forms:

```
(s/def ::company string?)
(s/def ::suite string?)
(s/def ::company-addr (s/union ::addr [::company ::suite]))
```

Schema gen

In the case of a schema, all elements are optional, and the generator will produce any combination:

```
(gen/sample (s/gen ::user) 5)
;;=> ({}
;;    #:user{:addr #:user{:state "H"}, :last "2"}
;;    #:user{:addr #:user{:zip 0, :state "2", :street "98z", :city "f"}, :last
"j", :first "K"}
;;    #:user{:addr {}, :id -2}
;;    #:user{:last "", :id -23, :first ""})
```

Unqualified keys

Nested unqualified schemas are supported as well:

```
(s/def ::order
  (s/schema {:purchaser string?
             :due-date inst?
             :line-items (s/coll-of (s/schema {:item-id pos-int?
                                               :quantity nat-int?})
                                    :kind vector?
                                    :min-count 1
                                    :gen-max 3)}))

(gen/sample (s/gen ::order) 5)
;; ({:due-date #inst "1970-01-01T00:00:00.000-00:00"}
;;  {}
;;  {:purchaser "", :due-date #inst "1969-12-31T23:59:59.999-00:00",
;;   :line-items [{:item-id 2, :quantity 0}]}
;;  {:purchaser "0"}
;;  {:due-date #inst "1969-12-31T23:59:59.999-00:00",
;;   :line-items [{:quantity 1} {} {:item-id 2, :quantity 3}]})
```

Helper functions

Some helper functions have been added (these are largely analogous to the similar functions for symbolic specs and spec objects):

- `schema*` - takes an explicated symbolic schema and returns a schema object
- `schema?` - checks whether an object is a schema object

There is also a new protocol clojure.alpha.spec.protocols/Schema.

Select

`s/select` is a spec op that uses a schema to define the world of possible keys and a selection pattern to specify the particular keys (and sub-keys for nested maps) are required in a particular context.

General form: `(s/select schema selection)`

- schema (required) - can be a registered schema name, schema form (like s/union), or literal schema
- selection (required) - vector of
  - `*`, the wildcard symbol
  - required keys (qualified or unqualified keywords)
  - optional subselections (maps of optional keyword to a selection pattern)

get-movie-times

Continuing with the schemas above (which are the same as the ones from "Maybe Not" talk), consider the "get-movie-times" example where you need to know only a user's id and zip code for lookup. The selection pattern here requires that both ::id and ::addr exist, and if ::addr exists, it must contain ::zip.

```
;; get-movie-times
(s/def ::movie-times-user (s/select ::user [::id ::addr {::addr [::zip]}]))

(s/valid? ::movie-times-user {::id 1 ::addr {::zip 90210}})
;;=> true

(s/explain ::movie-times-user {})
;; {} - failed: (fn [m] (contains? m :user/id)) spec: :user/movie-times-user
;; {} - failed: (fn [m] (contains? m :user/addr)) spec: :user/movie-times-user
```

```
(s/explain ::movie-times-user {::id 10 ::addr {}})
;; {} - failed: (fn [m] (contains? m :user/zip)) in: [:user/addr] at:
[:user/addr] spec: :user/movie-times-user
```

And these selects can also gen examples that conform to the selection:

```
(gen/sample (s/gen ::movie-times-user) 5)
;;=> (#:user{:last "", :first "", :id -1, :addr #:user{:zip 0}}
;;    #:user{:id 0, :addr #:user{:zip -1}}
;;    #:user{:last "", :id -1, :addr #:user{:state "BV", :street "40", :city
"Vx", :zip 0}}
;;    #:user{:last "A", :first "ZXl", :id -3, :addr #:user{:state "a", :street
"7H", :zip -4}}
;;    #:user{:last "4S3O", :first "c4Qo", :id 7, :addr #:user{:zip 2}})
```

Note that in all examples, the user has ::id and ::addr, which has a ::zip. Other elements from the schema and nested schema may optionally appear.

place-order

This example is for a user placing an order, where the name and full nested address is required, but the rest is not. Note this uses the same schemas but selects different keys and sub-keys.

```
(s/def ::place-order
  (s/select ::user [::first ::last ::addr
                    {::addr [::street ::city ::state ::zip]}]))

(s/valid? ::place-order {::first "Alex" ::last "Miller"
                         ::addr {::street "123 Elm" ::city "Springfield"
                                 ::state "IL" ::zip 12345}})
;; true

(s/explain ::place-order {::first "Alex" ::last "Miller" ::addr {::state "IL"}})
;; #:user{:state "IL"} - failed: (fn [m] (contains? m :user/city)) in:
[:user/addr] at: [:user/addr] spec: :user/place-order
;; #:user{:state "IL"} - failed: (fn [m] (contains? m :user/street)) in:
[:user/addr] at: [:user/addr] spec: :user/place-order
;; #:user{:state "IL"} - failed: (fn [m] (contains? m :user/zip)) in:
[:user/addr] at: [:user/addr] spec: :user/place-order
```

And it gens as well (notice differences from previous):

```
(gen/sample (s/gen ::place-order) 3)
;;=> (#:user{:first "", :last "", :addr #:user{:city "", :street "", :state "",
:zip 0}}
;;    #:user{:first "i", :last "", :addr #:user{:city "V", :street "v", :state
"", :zip 0}}
;;    #:user{:id -1, :first "58MF", :last "", :addr #:user{:city "tQ", :street
"c5", :state "q", :zip -1}})
```

Wildcard

The wildcard selection pattern indicates that all keys are required:

```
(gen/sample (s/gen (s/select ::user [*])) 3)
;;=> (#:user{:first "", :id -1, :last "", :addr #:user{:zip 0, :state "",
:street "", :city ""}}
;;    #:user{:first "r9", :id 1, :last "4i", :addr {}}
;;    #:user{:first "t2", :id 1, :last "72", :addr #:user{:state "35M", :city
"3B"}})
```

And a nested example:

```
(gen/sample (s/gen (s/select ::user [* {::addr [*]}])) 3)
;;=>  (#:user{:first "", :id 0, :last "", :addr #:user{:city "", :street "",
:state "", :zip 0}}
;;     #:user{:first "", :id -1, :last "n", :addr #:user{:city "0", :street "N",
:state "", :zip -1}}
;;     #:user{:first "H9", :id -1, :last "", :addr #:user{:city "n", :street
"60", :state "O", :zip -1}})
```

Unqualified keys

As mentioned above in the schema selection, the schema can supply a set of unqualified keys and the specs to use with them as well:

```
(gen/sample (s/gen (s/select {:a int? :b keyword?} [:a])) 5)
;;=> ({:b :C, :a -1}
;;    {:b :f_/s!, :a -1}
;;    {:b :J8.M+/+88, :a -1}
;;    {:a -2}
;;    {:b :IEcw.l?/X, :a -1})
```

Pages 3

Find a Page…

Home

Differences from spec.alpha

Schema and select

Clone this wiki locally

https://github.com/clojure/spec-alpha2.wiki.git