Forging code with schemas

Summary

- Presentation’s goals
- What is a schema?
- Introduction to Malli
- Use cases in status-mobile
- Conclusion & Next Steps
- Questions

Presentation’s Goals

- Share experimental results using Malli in status-mobile.
- Have fun ✨

Definition

Schema (plural schemas/schemata)

“A specification or description of the types of data represented in a database, the attributes they possess, and the relationships between them.”

– Oxford in informatics
https://www.oxfordreference.com/display/10.1093/oi/authority.20110803100445389

Intro to Malli: About

- A library by Metosin, created ~4 years old
- Already funded by Clojurists Together.
- Has been steadily improving based on community feedback.
- Jank will implement a gradual type system inspired by Malli's syntax.

> *"I believe Spec and Malli can co-exist and can even be friends together. Spec is an awesome tool for describing the core language and library APIs, and Malli the world we are living in."*
>
> – https://www.metosin.fi/blog/malli

Intro to Malli: Syntax

```clojure
:int

;; Can have properties
[:int {:min 1}]

;; Composable
[:or :int :string]

;; Enumerators
[:enum :blue :orange]

;; Example: "0123"
[:re #"^\d+$"]

;; Example: {"0x1" :hello}
[:map-of string? :keyword]

;; Example: :t/translation-xyz
[:qualified-keyword {:namespace :t}]
```

```clojure
;; Example: {:x 1 :y 2 :z 3}
[:map
 [:x :int]
 [:y :int]
 [:z {:optional true} :int]]

;; Arbitrary predicates
[:and
 [:map
  [:x :int]
  [:y :int]
  [:z {:optional true} :int]]
 [:fn
  (fn [{:keys [x y]}]
    (> x y))]]
```

Intro to Malli: Syntax

```clojure
;; :json/rpc-call effect
[:vector
 [:map
  [:method [:re #"^wakuext_.+$"]]
  [:params [:vector :any]]
  [:on-success [:or ::event fn?]]
  [:on-error [:or ::event fn?]]]]]

;; Function schema of :int -> :string
[:=> [:cat :int] :string]
```

```clojure
;; Re-frame app db root :activity-center
[:map {:closed true} ①
 [:filter {:required true}
  [:map
   [:status [:enum :read :unread]]
   [:type :int]]]
 [:loading? {:optional true} :boolean]
 [:contact-requests {:optional true}
  :any]
 [:cursor {:optional true}
  :string]
 [:notifications {:optional true}
  [:sequential :schema.shell/notification]] ②
 [:seen? {:optional true}
  :boolean]
 [:unread-counts-by-type {:optional true}
  [:map-of {:min 1}
   :schema.shell/notification-type :int]]]
```

Capabilities

What can we do with schemas?

What can we do?

1. Validate
2. Instrument functions
3. Transform data
4. Generate data

What can we do?

1. Validate
2. Instrument functions
3. Transform data
4. Generate data

Validation

```clojure
(require '[malli.core :refer [validate]])

(def ?schema
  [:map
   [:x :int]
   [:y [:maybe :int]]
   [:z {:optional true} :int]])

(validate ?schema {:x 10 :y 25})
;; => true

(validate ?schema {:x 10 :y nil})
;; => true

(validate ?schema {:x 10 :y 25 :z "30"})
;; => false
```

If validation fails, how do you know the reason(s)?

```clojure
(explain ?schema {:x 10 :y 25 :z "30"})

{:schema #object [malli$27056]
 :value  {:x 10 :y 25 :z "30"}
 :errors ({:path   [:z]
           :in     [:z]
           :schema #object [malli$26962]
           :value  "30"})}
```

Validation

```
(malli.dev.pretty/explain
 ?schema
 {:x 10 :y 25 :z "30"})

-- Validation Error -----------------------
Value:
{:x ..., :y ..., :z "30"}

Errors:
{:z ["should be an integer"]}

Schema:
[:map
 [:x :int]
 [:y [:maybe :int]]
 [:z {:optional true} :int]]
------------------------------------------
```

- Malli gives us full control over how error messages are generated and reported.

Validation: Use Case

Problem: test assertions in ClojureScript can result in cryptic output when comparing nested and/or large data structures.

Solution: define a *match* function that verifies a given value matches a schema, and if it does, returns it, otherwise, prints a humanized error and fail the test.

```
(match :int 42)
;; => 42

(assert (= 42 (match :int 42)))
;; => nil (no error)

(defn find-answer-to-life
  []
  "hello")

(deftest find-answer-to-life-test
  (let [answer (find-answer-to-life)]
    (is (= 42 (match :int answer)))))
```

```
-- Validation Error ----------------------
Value:
"hello"

Errors:
["should be an integer"]

Schema:
:int
-----------------------------------------
```

Validation: Use Case

Additionally, consider changing from Clojure core predicates to schemas.

```
(deftest some-test
  (let [input {:name "Bob"
               :age  25}]
    (is (match
          [:map
           [:name [:= "Alice"]]
           [:age :int]]
          input))))

-- Validation Error -----------------------------
Value:
  {:age ..., :name "Bob"}

Errors:
  {:name ["should be Alice"]}

Schema:
  [:map [:name [:= "Alice"]] [:age :int]]
-------------------------------------------------
```

1. Remove `cljs.core/=`
2. Use the `:=` schema

Validation: Use Case

Problem: the status-mobile re-frame app db is large, complex, and in great part, modified by untested event handlers. It's easy to introduce errors during development and hard to debug what caused them.

Partial solution: Implement a sort of *schema-on-write* semantic to guarantee only valid data is written to the app db. The developer would see precise and fast feedback on every attempt to write invalid data.

- Very easy to implement on top of re-frame.
- But can be a trap to implement in the beginning.
- Enormous benefit to be able to generate data during development and test.

What can we do?

1. Validate
2. **Instrument functions**
3. Transform data
4. Generate data

Instrumentation

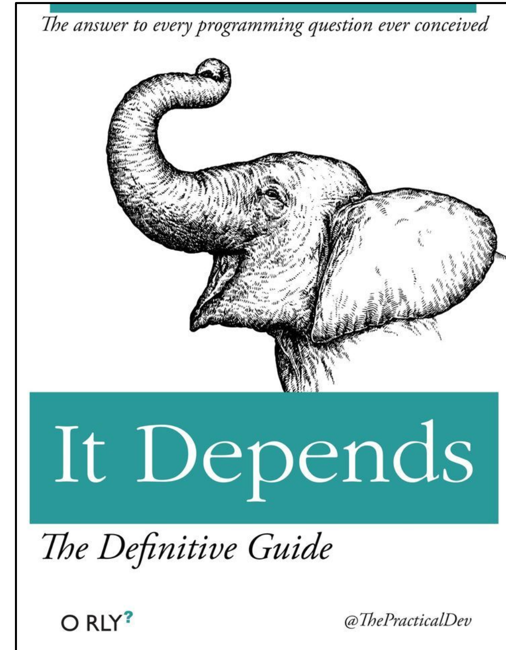
Definition: a mechanism where a function's input and/or output are automatically checked against its function schema.

```
(defn add
  [x y]
  (+ x y))

(malli.core/=> add
  [:=> [:cat :int :int]
   :int])

(add 1 2)
;; => 3

(add 1 :2)
;; => "1:2"
```

```
-- Schema Error -----------------------------------
Invalid function return value:
"1:2"

Function Var:
add

Function arguments:
[1 :2]

Output Schema:
:int

Errors:
{:in [],
:message "should be an integer",
:path [],
:schema :int,
:value "1:2"}
--------------------------------------------------
```

Instrumentation

1. When an instrumented function call receives invalid inputs, what should we do?
    a. Print/log
    b. Throw
    c. Show message on app screen
    d. It depends
    e. All of the above



Instrumentation: Use Case

Problem: quo components can have many variations and receive many arguments. Mobile devs resort to docstrings with ad-hoc “type annotations”, but they often get outdated and can’t be verified by a machine.

Solution: instrument quo components (functions), and replace type comments with schemas.

```
(defn view-internal
  [{:keys [number size blur? theme] :as props}]
  (let [size-value (get-in style/sizes [size :size])
        icon-size  (get-in style/sizes [size :icon-size])]
    [rn/view (style/container props)
     ...]))

(def view (quo.theme/with-theme view-internal))

(defn- view-internal
  ...)

(def ?schema
  [:=>
   [:cat
    [:map {:closed true} ①
     [:type [:enum :rounded :squared]]
     [:number [:re #"^\d+$"]]
     [:size [:enum :size/s-32 :size/s-24 :size/s-20
:size/s-16 :size/s-14]]
     [:theme {:optional true} :schema.common/theme] ②
     [:blur? {:optional true} :boolean]]]
   :any])

(def view
  (->> view-internal
       quo.theme/with-theme
       (instrument ::number-tag ?schema)))
```

Instrumentation: Use Case

Now when the number-tag component tests run:

```
(h/describe "number tag component test"
  (h/test "+3 render"
    (h/render [number-tag/view
               {:type   :rounded
                :number "abc"
                :size   :size/s-32
                :blur?  false}])
    (h/is-truthy (h/get-by-text "+3"))))
```

!

```
FAIL
component-spec/quo2.components.tags.number_tag.component_spe
c.js
 number tag component test
   ✕ +3 render (555 ms)
   √ +48 render (15 ms)

 ● number tag component test › +3 render
```

```
   -- Schema error -
:quo2.components.tags.number-tag.view/number-tag ----

   Invalid function arguments:
     [{:blur? false,
       :number "abc",
       :size :size/s-32,
       :type :rounded}]

   Input Schema:
     [:cat
      [:map
       [:type :keyword]
       [:number [#]]
       [:blur? {#} :boolean]
       [:size [#]]
       [:theme {#} :schema.common/theme]]]

   Errors:
     {:in [0 :number],
      :message "should match regex",
      :path [0 :number],
      :schema [:re #"^\d+$"],
      :value "abc"}

------------------------------------------------------------
```

Instrumentation: Use Case

Problem: There are 120+ quo previews. Although easy to maintain, we still have trouble maintaining preview descriptors up-to-date with the source implementation.

Solution: derive preview descriptors from instrumented functions.

Descriptor for the `number-tag` component:

```
(def descriptor
  [{:key     :type
    :type    :select
    :options [{:key :rounded}
              {:key :squared}]}
   {:key  :number
    :type :text}
   {:key     :size
    :type    :select
    :options [{:key   :size/s-32
               :value "32"}
              {:key   :size/s-24
               :value "24"}
              {:key   :size/s-20
               :value "20"}
              {:key   :size/s-16
               :value "16"}
              {:key   :size/s-14
               :value "14"}]}
   {:key  :blur?
    :type :boolean}])
```

Instrumentation: Use Case

```
(preview/generate-descriptor number-tag/?schema)

;; Returns the same descriptor (notice the sizes
;; are not "humanized").
[{:key :type, :type :select, :options [{:key
:rounded} {:key :squared}]}
 {:key :number, :type :text}
 {:key :size,
  :type :select,
  :options
  [{:key :size/s-32}
   {:key :size/s-24}
   {:key :size/s-20}
   {:key :size/s-16}
   {:key :size/s-14}]}
 {:key :theme, :type :select, :options [{:key
:light} {:key :dark}]}
 {:key :blur?, :type :boolean}]
```

We could go further, and generate entire preview namespaces on-the-fly, automatically.

```
(def ?schema
  [:=>
   [:cat
    [:map {:closed true}
     [:type [:enum :rounded :squared]]
     [:number {:preview/name    "Tag number:"
               :preview/initial "148"} 1
      [:re #"^\d+$"]]
     [:size [:enum :size/s-32 :size/s-24 ...]]
     [:theme {:optional      true
              :preview/show? false} 2
      :schema.common/theme]
     [:blur? {:optional true} :boolean]]]
   :any])
```

Instrumentation

- Can we write schemas in such a way as to only allow valid component variations?
  - Yes, with multi schemas and custom predicates.
- What should we instrument?
  - Functions at the system's borders (internal/external)
  - Highly reused functions, quo components, src/react-native/**
  - Event handlers, subscriptions
  - Data store functions
- Used in development or test environments, thus not in production!
- Gradual type checking != instrumentation.

What can we do?

1. Validate
2. Instrument functions
3. **Transform data**
4. Generate data

Transformations

- Every schema can define how it’s encoded and decoded to/from string/JSON.
- Useful to build an anti-corruption layer.
  - Decode signals or RPC responses and automatically transform them according to schemas. No more manual and error prone rename-key calls, etc.
  - Encode RPC requests before sending them to status-go.

What can we do?

1. Validate
2. Instrument functions
3. Transform data
4. Generate data

Generators

- Generators are a vast and fun topic.
- API surface is bigger, steeper learning curve, but rewarding.
- Custom generators written for Spec work with Malli and vice-versa.

```clojure
(require '[malli.generator :refer [generate]])

(generate :int)
;; => -5
;; => 197

(generate [:int {:min 10 :max 50}])
;; => 29

(generate
 [:map
  [:total {:optional true} [:int {:min 99}]]
  [:states [:enum :opened :closed :started]]])
;; => {:states :closed, :total 5546874}
;; => {:states :started}
```

```clojure
(def ?emoji-hash
  [:sequential
   {:gen/gen (gen/vector-distinct
              (gen/elements ["□" "👳" ...])
              {:num-elements 3})}
   :string])

(generate ?emoji-hash)
;; => ["□" "👲" "🤵"]
```

Generators: Use Case

Problem: in unit tests calling instrumented functions, it's necessary to pass valid arguments, but writing them by hand is a long and tedious process.

Solution: write or use an existing schema, then generate example data and assoc (hardcode) the minimum amount of data to fulfill assertions.

```
(deftest mark-as-read-test
  (testing "dispatches RPC call"
    (let [notif            (assoc (generate :schema.shell/notification)
                                  :id   "0x1"
                                  :read false                              2
                                  :type types/one-to-one-chat)
          activity-center (assoc (generate :schema.re-frame/db [:activity-center])
                                  :notifications                          3
                                  [notif])
          cofx             (assoc-in (generate :schema.re-frame/cofx)
                                     [:db :activity-center]               4
                                     activity-center)]
      (is (= {:json-rpc/call
              [{:method     "wakuext_markActivityCenterNotificationsRead"
                :params     [[(:id notif)]]
                :on-success [:activity-center.notifications/mark-as-read-success notif]
                :on-error   [:activity-center/process-notification-failure (:id notif)
                             :notification/mark-as-read]}]}
        1  (events/mark-as-read cofx (:id notif)))))))
```

Generators

- Other ideas:
  - Show a button in each preview screen that, when pressed, will randomly generate valid inputs for the descriptor and update the preview *state*.
  - Show a button in each preview screen that, when pressed, will render X number of instances with randomly generated variations.
  - Generate parts of the app db to inspect for correctness and performance levels (e.g. generate all types of notifications in all possible states, generate hundreds of messages in different states, etc).

- Malli generators stand on the shoulders of test.check.
- Test.check was inspired by QuickCheck (written in Haskell).
- Test.check is a *property-based testing* Clojure(Script) library.

Performance

- Much faster than Spec.
- Sometimes even faster than idiomatic code for data transformations.
- Validation, coercion, and other features can be used in production.
- What would be the overhead in mobile devices?
    - Instrumentation: zero cost

Conclusion & Next Steps

- Malli is flexible and schemas can be progressively leveraged.
- Complexity can go up in some areas, particularly related to data generation.
- A schema-driven approach can help us tackle a codebase with low test coverage and high churn.

- We don’t need to commit to any decision now.
- But if we do, I’d recommend we start simple:
  - Add the plumbing code to use Malli in the repo (quick, it’s mostly done).
  - Instrument a few widely used quo components, e.g. button (quick).
  - Repeat until more developers are familiar with a schema-driven approach.
  - Eventually attempt to solve other problems.

Thank you :)

Questions?