# RAD Guards

## 1 RAD Guards

Dont't validation, authorization and permissions share the exact same problem set? Can we go more into the core of that problem? Aren't they in essence just some logic to block a mutation or read from happening? Shouldn't there be like 1 concept to sit underneath them, and let the user handle (or declare handlers) for them? Should this be some concept that can block unwanted mutations or reads?

The Guard name is a working title. This feels too imperative, like it actually does something. I'm looking for a more descriptive name, something that describes a roadblock. This is not about using it yet, but about a measurable unit upon which to derive which decisions have to be made.

### 1.1 A simple guard

```
(defguard admin-only [{:keys [db/current-user]}]
  {:query [:db/current-user]}
  (if (admin? current-user)
    [::rad.guard/ok]
    [::rad.guard/not-allowed "Current user is not admin"]))
```

### 1.2 A complex guard

```
(defguard account-mutation-guard [{:keys [db/current-user account/id
                                          account/name    account/other-name]}]
  {:query [{::db/current-user [::user/id ::user/account-id]}
           ::account/id ::account/name ::account/other-name]
   ;; Specify this to scope the guard to a specific set of environments.
   ::rad.guard/environment #{::rad.guard.environment/server}}
  (cond
    ;; An authorisation clause
    (not= id ::user/account-id current-user)
    [::rad.guard/not-allowed [::rad.guard/reason :my-app.account/ownership
                              ::rad.guard/ui-message
                              "Whoops, this account does not belong to you"
                              ;; Open map to stick in a bunch of crap you
                              ;; might need for the handler… that
                              ;; could be merged into the env.
                              ]]
    ;; Validation error… bad example but I want to show off
    ;; complex validation that would be hard to do on attribute
    ;; level. If you have only one of the two it's valid.
    (and (nil? name) (nil? other-name))
    [::rad.guard/invalid [::rad.guard/reason :my-app.account/no-name-provided
                          ;; Display a highlight in the ui on
                          ;; those keys with a single reason message
                          ::rad.guard.validation/keys [::account/name
                                                       ::account/other-name]
                          ::rad.guard.reason/ui-message
                          "Fill out at least one of these"]]
    ;; Another authorisation clause, but with different
    ;; repercussions at a different priority.
    (not (pro-user? current-user))
    [::rad.guard/not-allowed [::rad.guard/reason :reason/paywall
                              ::rad.guard.reason/ui-message
                              "This feature is only available for our Pro users."]]))
```

### 1.3 Handling

You could create a multi-method that dispatches on the guard keys. RAD would only allow a result of `::rad.guard/ok`, all the other ones should dispatch on some key for error handling. (separating handling and detection obviously)

Not sure how the in depth handling should work but this would be a way:

- A mutation or read would execute and collect all the guards that apply to them
- group them by guard error type but keeping the order of appearance
- handle all the guards of the first type.

  Example: if the the first error is a validation, merge the validation errors and show all the errors on all the affected fields. If it's not-allowed just handle the first reason or something (dispatching a state transition to logging in with an error message in the auth sm).

This way the user's system can actually decide what to do next.

- Validation? Show errors
- Logged out? Log in screen.

It can even handle different outcomes of the same problem.

- Unauthorized? Show big fat error over everything (this shouldn't happen in good UIs really, unless the user tinkers with the url or payload. A good UI should not allow users to do something they can't do).
- Unauthorized - paywall? Display pricings modal with pro subscribtion call to action.
- Unauthorized but because [specific reason] the correct role is not assumed? Show a modal to switch role.

### 1.4 Things I like

I like how the reason key and message allows a myriad of different error handling. I think this is more composable than just saying "Well there are validations and authorisations, just stick with it".

I also like how you can declare an order of importance and handling order. A logged out user first gets the "please log in" errors before the actual data validation errors, because it's annoying to battle the form validator only to be sent to a login form and realise you need to login on your phone and do it all over again. Just prompt for login first, and then check for validation. BUT for paywall, maybe the form is something you want to show off as a demo, and once the user submits the form it get's a prompt to purchase a subscription. If RAD makes the choice in this, you loose flexibility and may need to rock your own 3 different systems.

### 1.5 Declaration options

(Spitball, random possibilities but way less clear in my mind. Just a proof of concept)

So a full guard should be a datastructure, but to declare them there can be a couple of shortcuts where the needs are basic and generic. There's the macro as the example above, but since most guard definitions will be data heavy, it would be elementary to generate a guard from:

```
;; Makes a guard that uses the default RAD (or current, pluggable?)
;; authentication state machine to verify the current session's role
;; or something and throws a ::rad.guard/not-allowed with the
;; configured data for the handler
{::form/guards
 [{::rad.guard/query [::db/current-user]
   ::rad.guard/predicate (comp admin? ::db/current-user)
   ::rad.guard/handler {::rad.guard/not-allowed
                        {::rad.guard/reason ::session.validation/admin-restriction
                         ::rad.guard/ui-message
                         "You need to be admin to view this page"}}}}
```

And the user can define templates for guards too or something:

```
{::form/guards
 [{::rad.guard/template [:my-app.guards/permitted?
                         {:feature-key :my-app.feature/orders}]}]}
```

Where a multimethod for `:my-app.guards/permitted?` will return a custom templated guard, notably one that checks that the current user can access the orders feature. I see a lot of composable opportunities in this approach.

### 1.6 Final thoughts

I also feel like this should only be used by the user for complex guarding, but attribute level validation (password-length etc..) should be defined on the defattr using spec or something and done by default in every environment (both server and client). These declarations should generate pre-fab guards that use a query for that key and validate that key, set to every environment by default, but should be configurable like custom guards. You should be able to specify the environment scope if desired, like only check specific format on server).

Additionally, these kinds of guards are less present in rapid protyping cycles (except for logged-in permission), so we should be happy with the attribute level primitive validations and a rad template guard. but when the application grows these complexer guards can be added later.

When the project gets huge and the domain complexity exceeds RAD's reach, guards should be replaced with a custom system somehow, probably by not using rad at all but maybe still making it possible to make the handlers system re-usable? Being able to still dispatch the guard error keys to your existing handling system that was built during greenfield. Attribute level validation is important and nice to have at every stage of the project.

### 1.7 Questions

- Will RAD have a concept of evironment?

  Like `:rad.environment/client.web` or `:rad.environment/server`

- Will RAD have query definitions that both work on client and server

  I'm not happy with defining a query on a guard. It feels more like query-ing for account-name is form context bound, but querying for current-user is system state bound. What would be a better approach to get the right data at the right guard at the right time?

  Maybe we should differentiate between the data concerning the current mutation, and the data dat can come from the rest of the system.

## 2 TODO Distributed validation

debounced server ping for validations that can only be done on the server but used in the client? Maybe declaring it in the guard's meta that it's only run on the server, but used in the server and client? Validate in the macro at compile time that used in both but validated client only makes no sense?