# io.aviso/rook Documentation

*Release 0.2.0*

Howard M. Lewis Ship

Sep 27, 2017

# Contents

CHAPTER 1

# Easier Routing for Pedestal

Rook is a way of mapping Clojure namespaces and functions as the endpoints of a Pedestal application.

Using Rook, you map a namespace to a URI; Rook uses metadata to identify which functions are endpoints. It generates a Pedestal routing table that you can use as-is, or combine with more traditional routing.

With Rook, your configuration is both less dense and more dynamic, because you only have to explicitly identify your namespaces and Rook finds all the endpoints within those namespaces.

Rook is also designed to work well the Component library, though Component is not a requirement.

Rook generates a set of table routes that can then be used by the io.pedestal.http/create-server bootstrapping function.

```
(require '[io.aviso.rook :as rook]
         [io.pedestal.http :as http])

(def service-map
  {:env :prod
   ::http/routes (rook/gen-table-routes {"/widgets" 'org.example.widgets
                                          "/gizmos"  'org.example.gizmos}
                                         nil})
   ::http/resource-path "/public"
   ::http/type :jetty
   ::http/port 8080})

(defn start
  []
  (-> service-map
      http/create-server
      http/start))
```

Rook supports many more options:

- Nested namespaces
- Defining interceptors for endpoints
- Leveraging metadata at the namespace and function level

- Defining constraints for path parameters

CHAPTER 2

# License

Rook is released under the terms of the Apache Software License 2.0.

## Defining Endpoints

The io.aviso.rook/gen-table-routes function is provided with namespaces; the actual endpoints are functions within those namespaces.

Rook identifies functions with the metadata :rook-route. Functions with this metadata will be added to the generated routing table.

Here's an example namespace with just a single endpoint:

```
(ns org.example.widgets
  (:require [ring.util.response :as r])

(defn list-widgets
  {:rook-route [:get ""]}
  []
  ;; Placeholder:
  (r/response {:widgets []}))
```

Endpoint functions take some number of parameters (more on this shortly) and return a Ring response map.

The above example is a placeholder: it returns a fixed and largely empty Ring response. In a real application, the function could be provided with a database connection of some sort and could perform a query and return the results of that query.

**Note:** Pedestal route matching is very specific: the `list-widgets` endpoint above is mapped to `/widgets`, and a client that requests `/widgets/` will get a 404 NOT FOUND response.

## Rook Routes

The route meta is either two or three terms, in a vector:

- The verb to use, such as :get, :post, :head, ... or :all.
- The path to match.
- (optional) A map of path variable constraints.

For example, we might define some additional endpoints to flesh out a typical resource-oriented API:

```
(defn get-widget
  {:rook-route [:get "/:id" {:id #"\d{6}"}]}
  [^:path-param id]
  ;; Placeholder:
  (r/not-found "WIDGET NOT FOUND"))

(defn update-widget
  {:rook-route [:post "/:id" {:id #"\d{6}"}]}
  [^:path-param id ^:request body]
  ;; Placeholder:
  (r/response "OK"))
```

The URI for the `get-widget` endpoint is `/widgets/:id`, where the `:id` path parameter must match the regular expression (six numeric digits).

This is because the namespace's URI is `/widgets` and the endpoint's path is directly appended to that. Likewise, the `update-widget` endpoint is also mapped to the URI `/widgets/:id`, but with the POST verb.

Because of how Pedestal routing is designed, a URI where the `:id` variable doesn't match the regular expression will be ignored (it might match some other endpoint, but will most likely match nothing and result in a 404 NOT FOUND response).

This example also illustrates another major feature of Rook: endpoints can have any number of parameters, but use metadata on the parameters to identify what is to be supplied.

Two common parameter metadata are used in the above example:

- :path-param is used to mark function parameters that should match against a path parameter.
- :request is used to mark function parameters that should match a key stored in the Ring request map.

Rook defines additional parameter metadata, and they are extensible.

## Namespace Metadata

That repetition about the `:id` path parameter constraint is bothersome, having it multiple places just makes it more likely to have conflicts.

Fortunately, Rook merges metadata from the namespace with metadata from the endpoint, allowing such things to be just defined once:

```
(ns org.example.widgets
  {:constraints {:id #"\d{6}"}}
  (:require [ring.util.response :as r]))

(defn list-widgets
  {:rook-route [:get ""]}
  []
  ;; Placeholder:
  (r/response {:widgets []})

(defn get-widget
  {:rook-route [:get "/:id"]}
  [^:path-param id]
  ;; Placeholder:
  (r/not-found "WIDGET NOT FOUND"))

(defn update-widget
  {:rook-route [:post "/:id"]}
  [^:path-param id ^:request body]
  ;; Placeholder:
  (r/response "OK"))
```

Here, each endpoint inherits the `:id` constraint from the namespace.

### Route Names

When Rook creates an interceptor, it provides a name for the interceptor; this is the keyword version of the fully qualified endpoint name.

The interceptor name is the default route name, used by Pedestal when it create URLs.

You can override the route name using the :route-name metadata on the endpoint function.

## Nested Namespaces

Defining nested (that is, hierarchical) namespaces requires a smidgen of extra work. In the non-nested case, it is usually sufficient to just specify the namespace (as a symbol), but with nested namespaces, a map is used; this is the full namespace definition.

```
(rook/gen-table-routes {"/hotels" {:ns 'org.example.hotels
                                   :nested {"/:hotel-id/rooms" 'org.example.rooms}}}
                       nil)
```

In this example, the outer namespace is mapped to `/hotels` and the nested rooms namespace is mapped to `/hotels/:hotel-id/rooms` ... in other words, whenever we access a specific room, we must also provide the hotel's id in the URI.

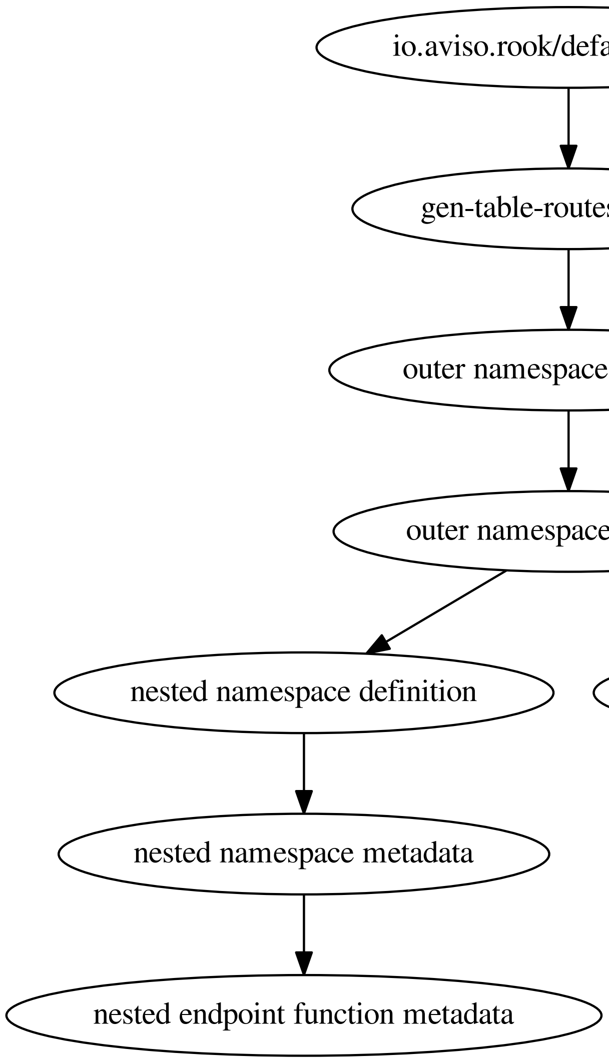
:nested is just a new mapping of paths to namespaces under `/hotels`; the map keys are extensions to the path, and the values can be namespace symbols or nested namespace definitions.

### Namespace Inheritance

Nested namespaces may inherit some data from their containing namespace:

- :arg-resolvers - a map from keyword to *argument resolver factory*
- :interceptors - a vector of *Pedestal interceptors*
- :constraints - a map from keyword to regular expression, for path parameter constraints

These options flow as follows:


io.aviso.rook/default-options
gen-table-routes options
outer namespace definition
outer namespace metadata
nested namespace definition
endpoint function metadata
nested namespace metadata
nested endpoint function metadata

Metadata on an endpoint function is handled slightly differently, :constraints overrides come from the third value in the :rook-route metadata.

In all cases, a deep merge takes place:

- nested maps are merged together, later overriding earlier
- sequences are concatenated together (using `concat` for sequences, or `into` for vectors)

This inheritance is quite useful: for example, the org.example.hotels namespace may define a `:hotel-id` constraint that will be inherited by the org.example.rooms namespace endpoints.

## Interceptors

Pedestal is all about interceptors, they are integral to how Pedestal applications are constructed and composed.

Each endpoint function may define a set of interceptors specific to that function, using :interceptors metadata.

Ultimately, Rook wraps endpoints functions so that they are Pedestal interceptors.

Interceptors may be *inherited* from the namespace and elsewhere.

## Interceptor Values

The interceptor values can be any of the values that Pedestal accepts as an interceptor:

- An interceptor, as by io.pedestal.interceptor/interceptor.
- A map, which is converted to an interceptor
- A bunch of other things that make sense in terms of Pedestal's deprecated *terse* routing syntax

Rook adds one additional option, a keyword.

The keyword references the :interceptor-defs option (passed to io.aviso.rook/gen-table-routes).

A matching value must exist (otherwise, an exception is thrown).

The value is typically a configured interceptor value.

**Pedestal Conflict?**

Normally, a function in the intererceptor chain is interpreted as a Ring handler. However, those are *only* allowed in the final position of an interceptor chain. That's never the case with Rook, because a Rook endpoint function is at the end of the interceptor chain.

Alternately, the value might be a function, which acts as an interceptor generator.

## Interceptor Generators

An interceptor generator is a function that creates an interceptor customized to the particular endpoint.

It is passed a map that describes the endpoint, and returns an interceptor.

The endpoint description has the following keys:

**:var** The Clojure Var containing the endpoint function.

**:meta** The metadata map for the endpoint function.

**:endpoint-name** A string version of the fully qualified name of the endpoint function

In this way, the interceptor can use the details of the particular endpoint to generate a custom interceptor.

For example, an interceptor that did some special validation, or authentication, might use metadata on the endpoint function to determine what validations and authentications are necessary for that particular endpoint.

Here's a more concrete example, part of Rook's test suite:

```
(ns sample.dynamic-interceptors
  (:require [io.pedestal.interceptor :refer [interceptor]]))

(defn endpoint-labeler
  [endpoint]
  (let [{:keys [endpoint-name]} endpoint]
    (interceptor {:name ::endpoint-labeler
                  :leave (fn [context]
                           (assoc-in context
                                     [:response :headers "Endpoint"]
                                     endpoint-name))})))
```

This generator returns an interceptor that operates during the leave phase, when there's a response map. It adds the `Endpoint` header to the response. This same interceptor generator could be added to any number of endpoints; a unique interceptor instance will be generated for each endpoint.

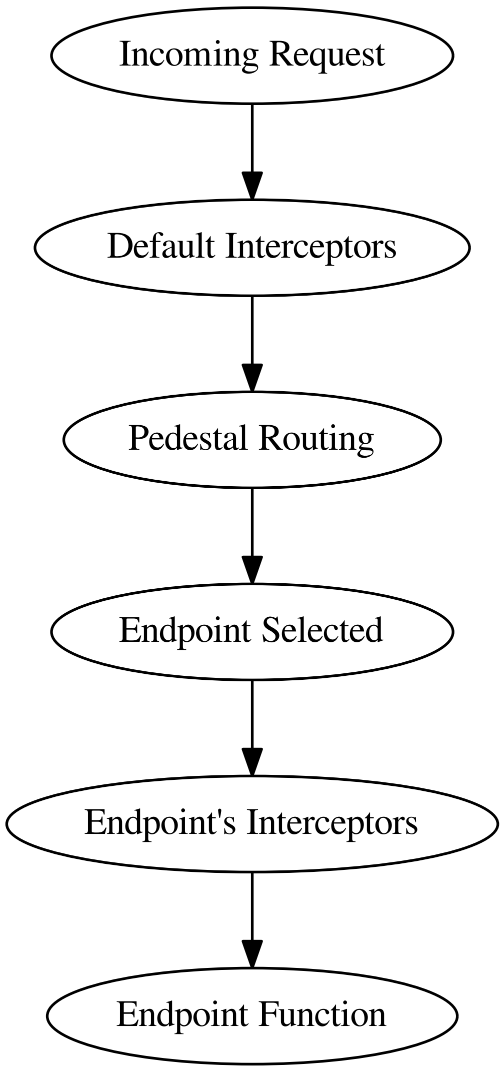
## Applying Interceptors

When a namespace provides :interceptor metadata, that's a list of interceptors to add to every endpoint in the namespace, and in any nested namespaces.

**Tip:** This can cast a wider net than is desirable; for example, including the io.aviso.rook.interceptors/keywordized-form interceptor at the namespace level will add it to *all* endpoints in the namespace, even those that do not include a POSTed form.

However, each individual endpoint will ultimately end up with its own individual interceptor list.

Further, none of those interceptors will actually execute in a request unless routing selects that particular endpoint to handle the request.



The default interceptors are usually provided by io.pedestal.http/create-server. These cover a number of cases such as handling requests for unmapped URIs, logging, preventing cross-site scripting forgery, and so forth.

The :interceptor metadata in namespaces and elsewhere simply builds up the Endpoint's Interceptors stage.

## Asynchronous Endpoints

There isn't much to say here: an endpoint can be asynchronous by returning a Clojure core.async channel instead of a Ring response map.

The channel must convey a Ring response map.

Really, Pedestal takes care of the rest.

## Argument Resolvers

In Rook, endpoint functions may *have many parameters*, in contrast to a traditional Ring handler (or middleware), or a Pedestal interceptor.

Argument resolvers are the bridge between the Pedestal context and the individual parameters of an endpoint function.

Compared to a typical Ring request handler, this saves your code from the work of destructuring the Ring request map. Beyond that, it is possible to expose other Pedestal context values, or (via custom argument resolver generators) entirely other business logic.

From a unit testing perspective, it is likely easier to pass a number of parameters than it is to construct the necessary Ring response map.

The :arg-resolvers option is a map from keyword to argument resolver *generator*.

When a parameter of an endpoint function has metadata with the matching key, the generator is invoked.

The generator is passed the parameter symbol, and returns the argument resolver function; the argument resolver function is used during request processing.

An argument resolver function is passed the Pedestal context and returns the argument value. There will be an individual argument resolver function created for each endpoint function parameter.

By convention, when the metadata value is the literal value `true`, the symbol is converted to a keyword.

So, if an endpoint function has a parameter `^:request body`, the :request argument resolver generator will return an argument resolver function equivalent to:

```
(fn [context]
  (let [v (get-in context [:request :body])]
    (if (some? v)
      v
      (throw (ex-info ...)))))
```

### Predefined Argument Resolvers

:request

Key in the Ring request map (:request). Value may not be nil.

:path-param

A path parameter, value may not be nil.

:query-param

A query parameter, as an HTTP decoded string.

:form-param

A form parameter. Endpoints using this should include the io.aviso.rook.interceptors/keywordized-form interceptor.

Further details about these are in the API documentation.

### Defining New Argument Resolvers

It is completely reasonable to provide your own :arg-resolvers (as the :arg-resolvers key in the options passed to io.aviso.rook/gen-table-routes).

```
(require '[io.aviso.rook :as rook])

(defn inject-arg-resolver [injections]
  (fn [sym]
    (let [k (rook/meta-value-for-parameter sym :inject)
          injection (get injections k)]
        (fn [_] injection))))

...

    (rook/gen-table-routes {...}
                           {:arg-resolvers {:inject (inject-arg-resolver␣
↪dependencies)}})
```

In the above example code, when the table routes are created, the dependencies symbol contains values that an endpoint function might need; for example, a database connection or the like.

When :inject is seen on a parameter symbol, the symbol is converted to a keyword via the function meta-value-for-parameter. This is then used to find the specific dependency value; finally, an argument resolver is returned that ignores the provided context and supplies the value.

## Argument Resolver Errors

Each endpoint parameter must apply to one and only one argument resolver. If you apply more than one (say `^:inject ^::request body`), you'll see an exception thrown from gen-table-routes.