Component Lifecycle

Contents

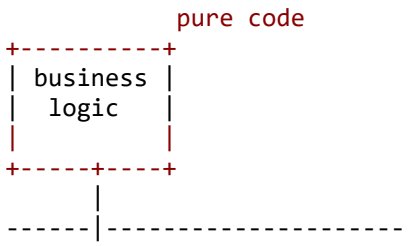
Luminus encourages using the Clean Architecture style for writing web applications.

The workflows in web applications are typically driven by the client requests. Since requests will often require interaction with a resource, such as a database, we will generally have to access that resource from the route handling the request. In order to isolate the stateful code, we should have our top level functions deal with managing the side effects.

Consider a route that facilitates user authentication. The client will supply the username and the password in the request. The route will have to pull the user credentials from the database and compare these to the ones supplied by the client. Then a decision is made whether the user logged in successfully or not, and its outcome communicated back to the client.

In this workflow, the code that deals with the external resources should be localized to the namespace that provides the route and the namespace that handles the database access.

The route handler function will be responsible for calling the function that fetches the credentials from the database. The code that determines whether the password and username match represents the core business logic. This code should be pure and accept the supplied credentials along with those found in the database explicitly. This structure can be seen in the diagram below.



```
      |     stateful code
+-----+----+   +-----------+
|  route   |   |           |
| handlers +---+  database |
|          |   |           |
+----------+   +-----------+
```

Keeping the business logic pure ensures that we can reason about it and test it without considering the external resources. Meanwhile, the code that deals with side effects is pushed to a thin outer layer, making it easy for us to manage.

Managing Component Lifecycle

The management of stateful components, such as database connections, is handled by the mount library. The library handles the lifecycle of such resources within the application ensuring that any such resources are started and stopped as necessary.

Luminus encourages keeping related domain logic close together. Therefore, in cases where we have functions that rely on an external resource the management of the state for that resource should be handled in the same namespace where the functions using it are defined.

Stateful components belong to the namespace they're declared in. To create a component we need to reference the `mount.core/defstate` macro in the namespace definition and then use it as follows:

```
(ns myapp.resource
  (:require [mount.core :refer [defstate]]))

(defn connect []
  ;;open-a-remote-connection should return the connection instance
  {:state :connected})

(defn disconnect [conn]
  (assoc conn :state :disconnected))

(defstate conn
  :start (connect)
  :stop (disconnect conn))
```

When the component is started the function bound to the `:start` key will be called. It's result will be used as the value for the state. In the example above, the `conn` will contain the map returned by the `connect` function.

When the component is shut down then the function bound to the `:stop` key is called. This function must accept the current state of the var. The function is expected to clean up any external resources before the component is shutdown.

The component dependencies are inferred from the namespace hierarchy. If namespace `a` references namespace `b` then the component specified using `defstate` in namespace `b` will be started before the one

specified in namespace `a`. When the system is shutdown the `:stop` functions for each state are called in the reverse order.

For example, we may have one namespace that loads the configuration and another that used the configuration to connect to a database. This could be expressed as follows:

```
(ns <app>.config
  (:require [cprop.core :refer [load-config]]
            [cprop.source :refer [from-resource]]
            [mount.core :refer [args defstate]]))

(defstate env :start (load-config :merge [(args)]))
```

The `<app>.config` namespace loads the config into the `env` state var. We can now access this config in a different namespace:

```
(ns app.db
  (:require [mount.core :refer [defstate]]
            [app.config :refer [env]]))

(defn connect! [config] ...)

(defn disconnect! [conn] ...)

(defstate conn :start (connect! env)
               :stop (disconnect! conn))
```

The component hierarchy is initialized by calling `mount.core/start` and stopped with `mount.core/stop`. This is done by the `<app>.core/start-app` and `<app>.core/stop-app` functions respectively.

Luminus provides an `<app>.user` namespace found in the `env/dev/clj/user.clj` file. This namespace provides convenience functions for starting and stopping the application states from the REPL:

```
(ns user
  (:require [mount.core :as mount]
            [<app>.config :refer [env]]
            <app>.core))

(defn start []
  (mount/start-without #'<<project-ns>>.core/repl-server))

(defn stop []
  (mount/stop-except #'<<project-ns>>.core/repl-server))

(defn restart []
  (stop)
  (start))
```

The states can be started selectively by explicitly providing the namespaces to be started and stopped to the `start` and `stop` functions:

```
(mount.core/start #'app.config #'app.db)

(mount.core/stop #'app.config #'app.db)
```

Alternatively, it's possible to specify the namespaces that should be omitted from the lifecycle using the `start-without` function:

```
(start-without #'app.db)
```

Finally, the states can be replaced by alternate ones such as mock states for testing using the `start-with` function:

```
(start-with #'app.db #'app.test.mock-db)
```

In the above example, the `app.db` will be replaced by the `app.test.mock-db` when the the components are loaded.

Please see the official documentation for further details on using mount.
