Noir uses the (defpage) macro to define what happens when a certain URL is accessed. To do this, Noir uses Compojure, which handles matching the request to a handler as well as setting up the routes for resources and such. By using Compojure, we have access to a powerful mechanism for defining routes and their actions. Let's see what we can do.

```clojure
;; A very simple page definition that maps to the root of your site.
(defpage "/" []
  "hello")
```

view raw routes.clj This Gist brought to you by GitHub.
With this page definition you see that the first parameter is the URL to map this page to, the second parameter is a destructuring form for the params of the request (we'll get to that shortly) and the rest is the content of the response. This will respond to an HTTP GET request, but what happens when we want to handle a post?

```clojure
(defpage [:post "/"] []
  "You posted here!")
```

view raw post.clj This Gist brought to you by GitHub.
What we've done here is adjusted our first parameter to be a vector containing the keyword of the request type to respond to followed by the url. You can do this with any of the HTTP request types:

```clojure
(defpage [:get "/"] [] "This is a get") ;; same as (defpage "/" [] ..)
(defpage [:post "/"] [] "This is a post")
(defpage [:put "/"] [] "This is a put")
(defpage [:any "/"] [] "This is any request type")
;; and so on...
```

view raw types.clj This Gist brought to you by GitHub.
Now with something like a post, it's not very useful if we post a form and can't get to the posted values. This is what that second value in the routes is for. It's a destructuring form for the params of the request. This includes all of the GET and POST variables. So let's say we have a login form that has fields named username and password and we post it to /login

```clojure
(defpage [:post "/login"] {:keys [username password]}
  (str "You tried to login as " username " with the password " password))
```

view raw params.clj This Gist brought to you by GitHub.
So here we're pulling the values posted as username and password out of the params. The destructuring form used here is just like the one you'd use in a (let) form, so you can use all of clojure's powerful destructuring concepts. There's another kind of param that you have access to here as well, and that is route params, which are used for dynamic routes.

```clojure
(defpage "/user/:id" {:keys [id]}
  (str "You are user number " id))
```

view raw routeparams.clj This Gist brought to you by GitHub.
This will match any routes like /user/12 and bind the last segment to the param key :id. This allows you to define some pretty complex routes and pull the interesting parts of those out of the URL. You can also take this a step further and even define regex's for the route params to match against:

```clojure
(defpage [:get ["/user/:id" :id #"\d+"]] {:keys [id]}
  (str "You are user number " id))
```

view raw complexparams.clj This Gist brought to you by GitHub.

What happens to the content we return?

This depends on what you return. If you simply return a string like we are here, or like you would by using (defpartial) or hiccup's (html) function, that is interpreted as the body of your HTTP Response and sent off. Alternatively, you can return a map with keys that correspond to the Ring spec. Here's an example of returning a response with a different status code:

```
(defpage "/error" []
  {:status 500
   :body "Oh no! An error has occurred"})
```
view raw mapresponse.clj This Gist brought to you by GitHub.
In general, you shouldn't need to use anything more than strings and if you find yourself needing to, check out the noir.response namespace first.

Routes as filters

Noir also has the concept of a pre-route, which is a set of routes that get evaluated before you get to the routes defined by (defpage). This is usually used to filter out access to an entire set of URLs based on permissions. Say for example, that you have an admin section, instead of checking whether or not a user has permission in every single (defpage), you could just do this:

```
(pre-route "/admin/*" {}
           (when-not (users/admin?)
             (resp/redirect "/login")))
```
Noir uses [Hiccup] to generate HTML, which represents elements as vectors whose first item is a keyword of the tag name. The best way to learn is to simply walk through a series of inputs and see what Hiccup outputs:

```
(use 'hiccup.core)

;; ****************************************************************
;; The basics
;; ****************************************************************

(html [:p])
  => "<p />"

;; Any elements after that become the content of the tag
(html [:p "hey"])
  => "<p>hey</p>"

;; Tags can be nested inside of tags and everything ends up concatenated
(html [:p "Hello " [:em "world"]])
  => "<p>Hello <em>world</em></p>"

;; You can specify attributes by supplying a map after the tag name
(html [:p {:id "my-p"} "hey"])
  => "<p id=\"my-p\">hey</p>"

;; There are shortcuts for setting ID and class, if you know CSS,
;; these should look familiar
(html [:p#my-p [:span.pretty "hey"]])
```

```
  => "<p id=\"my-p\"><span class=\"pretty\">hey</span></p>"

;; You can escape a string using the (escape-html) function
(html [:p (escape-html "<script>Do something evil</script>")])
  => "<p>&lt;script&gt;Do something evil&lt;/script&gt;</p>"

;; the h function is a shortcut for (escapte-html)
(html [:p (h "<script>Do more evil</script>")])
  => "<p>&lt;script&gt;Do more evil&lt;/script&gt;</p>"

;; you can actually generate generic xml too
(html [:books
       [:book#142 {:title "Noir for beginners"}]])
  => "<books><book id=\"142\" title=\"Noir for beginners\" /></books>"

;; ****************************************************************
;; Page helpers
;; ****************************************************************

(use 'hiccup.page-helpers)

;; you can use the html5, html4, or xhtml functions to write the
;; doctype boilerplate for you
(html (html5 [:p "hey"]))
  => "<!DOCTYPE html>\n<html><p>hey</p></html>"

;; there are helpers for including css and js files
(html (include-js "/js/core.js")
      (include-css "/css/reset.css"))
  =>"<script src=\"/js/core.js\" type=\"text/javascript\"></script>
     <link href=\"/css/reset.css\" rel=\"stylesheet\" type=\"text/css\" />"

;; there are also functions for creating links and images
(html (link-to "http://www.webnoir.org" "Noir")
      (mail-to "cool@cool.com")
      (image "/img/logo.png" "Noir"))
  => "<a href=\"http://www.webnoir.org\">Noir</a>
      <a href=\"mailto:cool@cool.com\">cool@cool.com</a>
      <image alt=\"Noir\" src=\"/img/logo.png\" />"

;; these functions can take maps to add custom attributes too
(html (link-to {:class "pretty"} "http://www.webnoir.org" "Noir"))
  => "<a class=\"pretty\" href=\"http://www.webnoir.org\">Noir</a>"

;; ****************************************************************
;; Form helpers
;; ****************************************************************

(use 'hiccup.form-helpers)

;; form helpers help you write the boilerplate for creating fields.
;; There are functions for all the different html input types.
(html (form-to [:post "/login"]
               (text-field "Username")
               (password-field "Password")
               (submit-button "Login")))
  => "<form action=\"/login\" method=\"POST\">
     <input id=\"Username\" name=\"Username\" type=\"text\" />
     <input id=\"Password\" name=\"Password\" type=\"password\" />
     <input type=\"submit\" value=\"Login\" />
     </form>"

;; the fields can take initial values as well
(html (text-field "Username" "chris"))
  => "<input id=\"Username\" name=\"Username\" type=\"text\"
      value=\"chris\" />"
```

view raw hiccup.clj This Gist brought to you by GitHub.

Partials

Noir adds a simple macro to make writing functions that return html easy to identify and nice to write: (defpartial). The macro defines a function with the given name and just wraps the body in Hiccup's (html) function:

```
(defmacro defpartial
  "Create a function that returns html using hiccup. The function is
   callable with the given name."
  [fname params & body]
  `(defn ~fname ~params
     (html
       ~@body)))

(defpartial hello [person]
  [:p "Hello " person])

(hello "noir")
  =>"<p>Hello noir</p>"
```

view raw defpartial.clj This Gist brought to you by GitHub.
One of the greatest advantages of hiccup is the ability to use these html generating functions with higher-order functional concepts like map. As an example, here is the idiomatic way to generate a list of blog posts in Noir:

```
(defpartial post-item [{:keys [perma-link title md-body date tme]}]
            [:li.post
             [:h2 (link-to perma-link title)]
             [:ul.datetime
              [:li date]
              [:li tme]]
             [:div.content md-body]])

(defpartial posts-list [items]
            [:ul.posts
             (map post-item items)])
```

view raw map.clj This Gist brought to you by GitHub.

Other templating options

Noir intentionally doesn't lock you into a specific templating solution, since it is highly dependent on the make up of your team and personal preference. As such, (defpartial) is the only real convenience method included in Noir for generating HTML. This means you can use Enlive, StringTemplate, or anything else you can think of inside of your page definitions.

Getting some input

The first thing we need to do is create a form. We'll do that using hiccup, so if you haven't read the Generating HTML doc, go do so real quick. In terms of organization, I've found it best to define forms as a partial of their fields like so:

```
(defpartial layout [& content]
  (html5
    [:head
     [:title "Forms"]]
    [:body
     content]))

(defpartial user-fields [{:keys [firstname lastname]}]
  (label "firstname" "First name: ")
  (text-field "firstname" firstname)
  (label "lastname" "Last name: ")
  (text-field "lastname" lastname))

(defpage "/user/add" {:as user}
  (layout
    (form-to [:post "/user/add"]
            (user-fields user)
            (submit-button "Add user"))))
```
view raw partials.clj This Gist brought to you by GitHub.
This allows you to use that partial for all the various incarnations of the form (add, edit, etc). Another good practice is to have forms post to the same URL, this will remove the need for extraneous redirects in the case of an error. Right now, though, our form posts to a URL that we haven't handled. So let's define a page for the POST to /user/add. Basically what we'll want it to do is check if our input is valid and either display a success message or show the form again:

```
(require '[noir.response :as resp])

(defn valid? [{:keys [firstname lastname]}]
  true)

(defpage [:post "/user/add"] {:as user}
  (if (valid? user)
    (layout
      [:p "User added!"])
    (render "/user/add" user)))
```
view raw post.clj This Gist brought to you by GitHub.
This is probably the first time you've seen the (render) function. It allows you to call a page by its url as if it were just a normal function. In this case, it's how we'll show the form again without having to redirect. Note that we're also passing our parameters through so that whatever we inputted shows up when we render the form again. So now we need to fill in that (valid?) function, which we'll do by using noir's validation functions.

```
(require '[noir.validation :as vali])

(defn valid? [{:keys [firstname lastname]}]
  (vali/rule (vali/min-length? firstname 5)
             [:firstname "Your first name must have more than 5 letters."])
  (vali/rule (vali/has-value? lastname)
             [:lastname "You must have a last name"])
  (not (vali/errors? :lastname :firstname)))
```
view raw validation.clj This Gist brought to you by GitHub.
What you see here is a set of rules that have the form (rule passed? [error-key error-text]) where if value of passed? is false, the error text is added to the error key. There are several ways to then retrieve those errors, one of which is to ask if there are any errors for a list of fields. That's done via the (errors?) function we use at the end to determine if all our rules passed. Note that there's nothing special about the value you pass to (rule) and you can use anything that returns truthiness as your test.
Alright, we have errors and now we need to show them on our form. The last thing we'll use is the (on-error) function to show a partial when we have an error in our form.

```
(defpartial error-item [[first-error]]
  [:p.error first-error])

(defpartial user-fields [{:keys [firstname lastname]}]
  (vali/on-error :firstname error-item)
  (label "firstname" "First name: ")
  (text-field "firstname" firstname)
  (vali/on-error :lastname error-item)
  (label "lastname" "Last name: ")
  (text-field "lastname" lastname))
```
view raw errors.clj This Gist brought to you by GitHub.
One thing to note here is that errors for a field are provided as a collection since multiple rules can add errors to a field. In this case, we only care about the first error, so our partial destructures it down to one value.

And that's it! Here's the complete code:

```
(use 'noir.core 'hiccup.page-helpers 'hiccup.form-helpers)
(require '[noir.validation :as vali])
(require '[noir.response :as resp])

(defpartial layout [& content]
  (html5
    [:head
     [:title "Forms"]]
    [:body
     content]))

(defpartial error-item [[first-error]]
  [:p.error first-error])

(defpartial user-fields [{:keys [firstname lastname]}]
  (vali/on-error :firstname error-item)
  (label "firstname" "First name: ")
  (text-field "firstname" firstname)
  (vali/on-error :lastname error-item)
  (label "lastname" "Last name: ")
  (text-field "lastname" lastname))

(defn valid? [{:keys [firstname lastname]}]
  (vali/rule (vali/min-length? firstname 5)
             [:firstname "Your first name must have more than 5 letters."])
  (vali/rule (vali/has-value? lastname)
            [:lastname "You must have a last name"])
  (not (vali/errors? :lastname :firstname)))

(defpage "/user/add" {:as user}
  (common/layout
    (form-to [:post "/user/add"]
            (user-fields user)
            (submit-button "Add user"))))

(defpage [:post "/user/add"] {:as user}
  (if (valid? user)
    (layout
      [:p "User added!"])
    (render "/user/add" user)))
```

Sessions

Noir uses Ring's session handling to store per-user sessions and provides a simple API to get and update values in a stateful way. These functions are found in the noir.session namespace and behave like you'd expect:

```
(require '[noir.session :as session])

(session/put! :username "chris")
(session/get :username)
;;  => "chris"

(session/remove! :username)
(session/get :username)
;; => nil

(session/get :username "anon")
;; => "anon"

(session/put! :username "chris")
(session/put! :user-id 2)
(session/clear!)
```

view raw session.clj This Gist brought to you by GitHub.
As of 1.1.0, Noir also has flashing, which is typically used to store a simple message across redirects. For example, if you redirect after a user is created, you would show the message "User added" on the user listings page. Note that flashes in Noir have the lifetime of one retrieval, meaning that after the first (flash-get) the value will be nil.

```
(session/flash-put! "User added!")
(session/flash-get)
;; => "User added!"
;;
(session/flash-get)
;; => nil
```

view raw flash.clj This Gist brought to you by GitHub.

Cookies

Cookies also have a simple stateful API that operates on the name of the cookie (which can either be a keyword or string).

```
(require '[noir.cookies :as cookies])

(cookies/put! :user-id 2)
(cookies/get :user-id)
;; => 2

(cookies/get :username "anon")
;; => "anon"

;; you can also pass a map that has all the cookie's attributes
(cookies/put! :tracker {:value 29649 :path "/" :expires 1})
```

view raw cookies.clj This Gist brought to you by GitHub.

A note on testing

One thing you have to keep in mind when using these functions is that they are only valid within the context of a request, i.e. inside of a (defpage). Trying to call these outside of the noir stack will throw an exception, since the values for the session and cookies come from modified request objects. In order to test these, you can use the (with-noir) macro in noir.util.test.

```
(use 'noir.util.test)

(deftest cookies-get
         (with-noir
           (is (nil? (cookies/get :noir)))
           (is (= "noir" (cookies/get :noir "noir")))))
```

The man in the middle

What is middleware? Well, here's a quote from the Ring Concepts doc:

> Middleware are higher-level functions that add additional functionality to handlers. The first argument of a middleware function should be a handler, and its return value should be a new handler function.

So basically, it's a function that takes in a handler and returns a new, more powerful handler. That's great, but that doesn't tell us why we should care about them. You can think of middleware as a series of transformations that gets applied to requests before they get routed and after they a response is generated.

Let's look at a practical example: enabling utf-8 characters in all of our responses. Noir includes a middleware function to do exactly that, and it looks like this:

```
(defn wrap-utf-8
  "Adds the 'charset=utf-8' clause onto the content type declaration,
  allowing pages to display all utf-8 characters."
  [handler]
  (fn [request]
    (let [resp (handler request)
          ct (get-in resp [:headers "Content-Type"])
          neue-ct (str ct "; charset=utf-8")]
```

```
      (assoc-in resp [:headers "Content-Type"] neue-ct))))
```
view raw utf8.clj This Gist brought to you by GitHub.
So as you can see, we're returning a new handler function that takes in the request map and then calls the next handler to get a response. Once it has the response it gets the "Content-Type" header and appends "; charset=utf-8" onto it and associates that into the response map. These functions are usually fairly simple, like this one, and always follow the form of returning an anonymous function that takes in a request and returns a response. If you look through the Noir source, a great deal of its functionality is implemented in middleware. Another great example is ring itself.

To add a custom piece of middleware to noir, all you have to do is:

```
(server/add-middleware my-middleware)
```
view raw add.clj This Gist brought to you by GitHub.

So when do I use it?

Anytime you need to modify requests or responses at a level beyond simply modifying status codes or bodies, middleware is likely your best option. Keep in mind, however, that every single request will then be routed through that function. Performance is important here, and middleware's use should be judicious. A good rule of thumb is if every single request needs to have something done to it or if you need access to the http request/response information directly, then middleware is an option. Otherwise, see if there's another solution.

Using Noir with other Ring-based libraries

As of Noir 1.1.0 you can use the (noir.server/gen-handler) function to create a standard Ring-handler for your noir application. The only catch is to make sure that you do so after you've defined your pages, past that, you simply pass your server options like normal and you're on your way:

```
(require '[noir.server :as server])

(server/load-views "src/noir-example/views")

(def handler (server/gen-handler {:mode :dev
                                  :ns 'noir-example}))
```
view raw genhandler.clj This Gist brought to you by GitHub.
Now you could use this in lein ring or lein beanstalk, for example, by simply adding the handler to your project definition:

```
(defproject noir-example "0.1.0"
            :description "An example of a noir project"
            :dependencies [[org.clojure/clojure "1.2.1"]
                           [noir "1.1.0"]]
            :dev-dependencies [[lein-ring "0.4.3"]]
            :ring {:handler noir-example.server/handler}
            :main noir-example.server)
```