# Clojure Web Development

Philipp Schirmacher

groups.google.com/group/clojure-dus

## Agenda

- Clojure Basics
- Web Development
  - Libraries
  - Micro Framework
- Demo

## Generic Data Types

```clojure
{:name "Clojure"
 :features [:functional :jvm :parens]
 :creator "Rich Hickey"
 :stable-version {:number "1.4"
                  :release "2012/04/18"}}
```

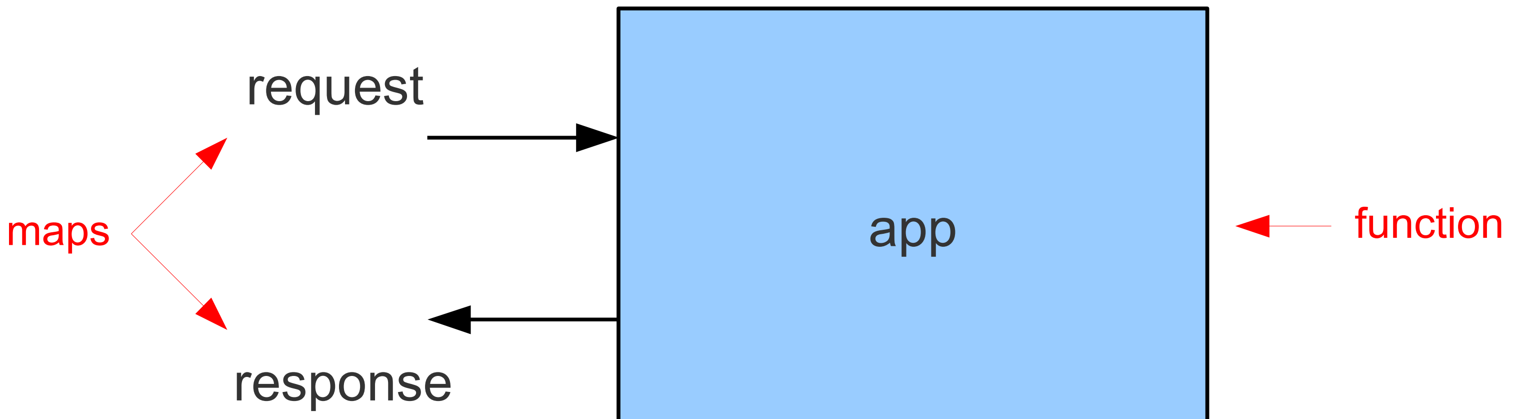
## Functions

```
(+ 1 2)

> 3


(:city {:name "innoQ"
        :city "Monheim"})

> "Monheim"


(map inc [1 2 3])

> (2 3 4)
```

```clojure
(defn activity [weather]
  (if (nice? weather)
    :surfing
    :playstation))
```

```
(defn make-adder [x]
  (fn [y]
    (+ x y)))

(def add-two (make-adder 2))

(add-two 3)
> 5
```

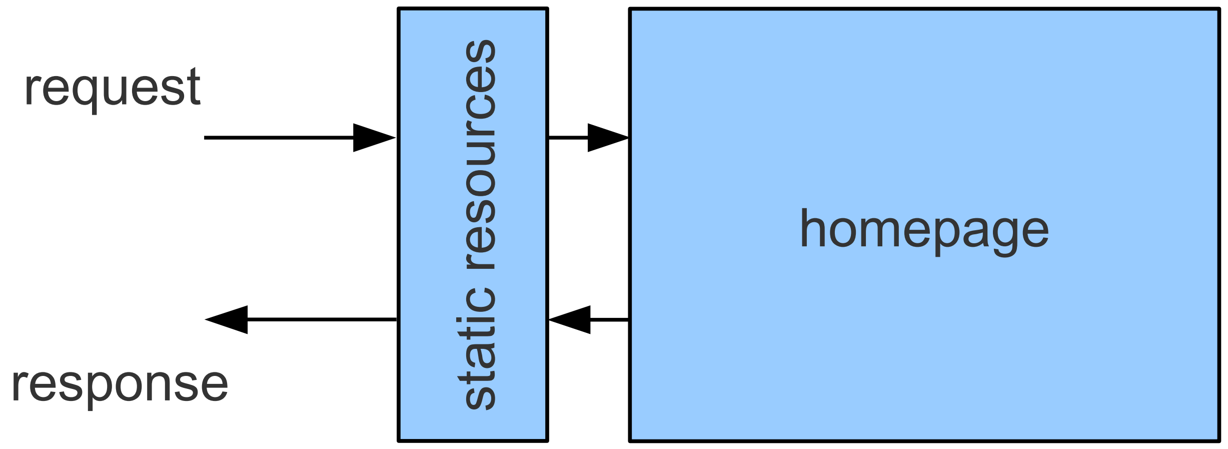
## Web Development?


request
maps
app
function
response

## Ring

```clojure
(defn hello-world-app [req]
  {:status 200
   :headers {"Content-Type" "text/plain"}
   :body "Hello, World!"})

(hello-world-app {:uri "/foo"
                  :request-method :get})
> {...}

(run-jetty hello-world-app {:port 8080})
```

```clojure
(defn my-first-homepage [req]
  {:status 200
   :headers {"Content-Type" "text/html"}
   :body (str "<html><head>"
              "<link href=\"/pretty.css\" ...>"
              "</head><body>"
              "<h1>Welcome to my Homepage</h1>"
              (java.util.Date.)
              "</body></html>")})
```


request
response
static resources
homepage

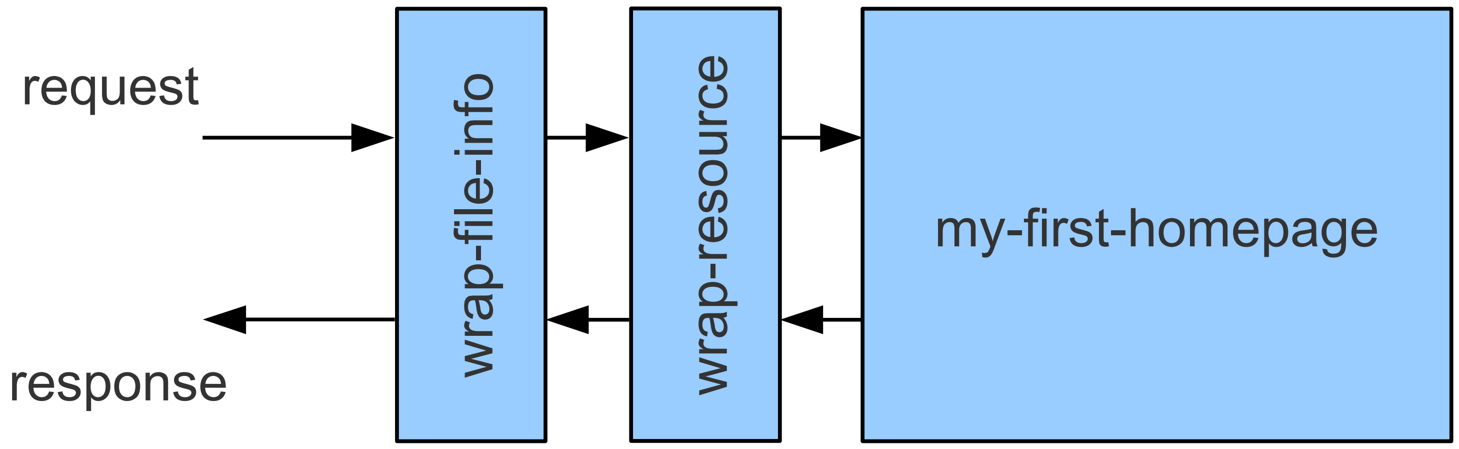
```clojure
(defn decorate [webapp]
  (fn [req]
    ...before-webapp...
    (webapp req)
    ...after-webapp...))
```

```clojure
(defn decorate [webapp]
  (fn [req]
    (if (static-resource? req)
      (return-resource req)
      (webapp req))))
```

```clojure
(defn wrap-resource [handler root-path]
  (fn [request]
    (if-not (= :get (:request-method request))
      (handler request)
      (let [path (extract-path request)]
        (or (resource-response path {:root root-path})
            (handler request))))))
```

```
(defn my-first-homepage [req] ...)

(def webapp
  (wrap-resource my-first-homepage "public"))

(run-jetty webapp {:port 8080})
```

```
(webapp {:uri "/pretty.css"
         :request-method :get
         :headers {}})
> {:status 200
   :headers {}
   :body #<File ...resources/public/pretty.css>}
```


request
response
wrap-file-info
wrap-resource
my-first-homepage

```clojure
(defn homepage [req] ...)

(def webapp
  (-> homepage
      (wrap-resource "public")
      wrap-file-info))

(run-jetty webapp {:port 8080})
```

```clojure
(wrap-file-info
  (wrap-resource
    homepage
    "public"))
```

```
(webapp {:uri "/pretty.css"
         :request-method :get
         :headers {}})
> {:status 200
   :headers {"Content-Length" "16"
             "Last-Modified" "Thu, 14 Jun ..."
             "Content-Type" "text/css"}
   :body #<File ...resources/public/pretty.css>}
```

wrap-resource

wrap-file

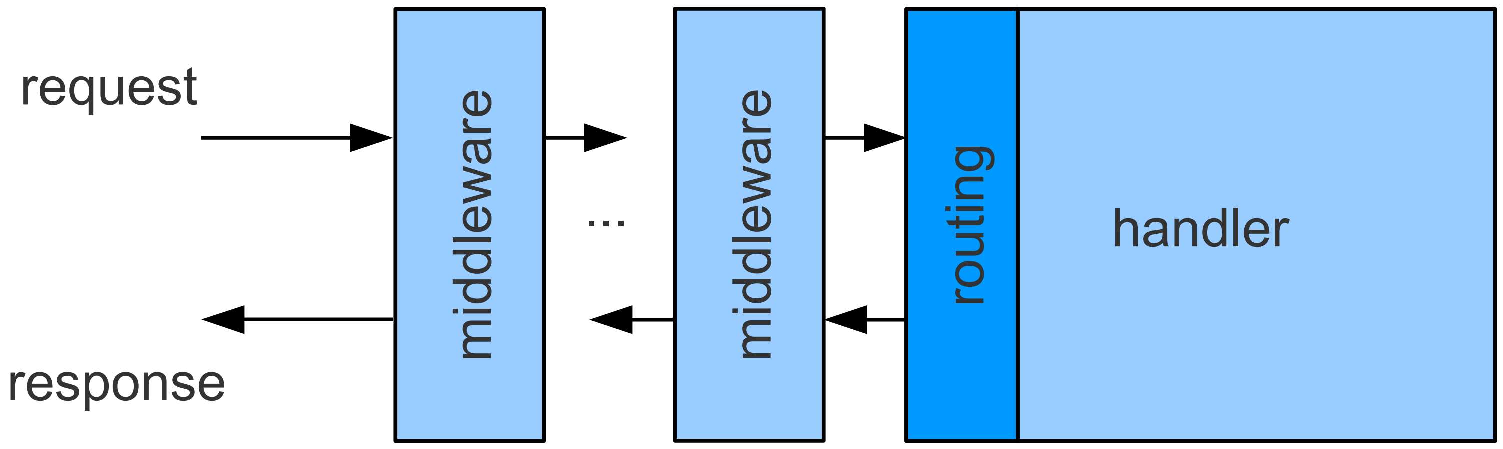
wrap-params

wrap-session

wrap-flash

wrap-etag

wrap-basic-authentication

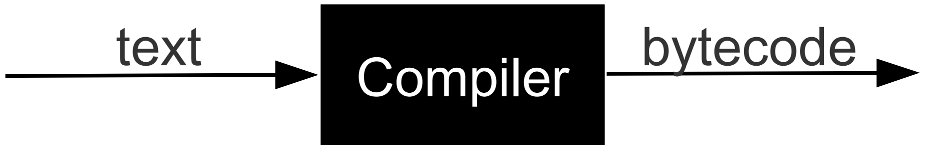

request
response
middleware
...
middleware
routing
handler

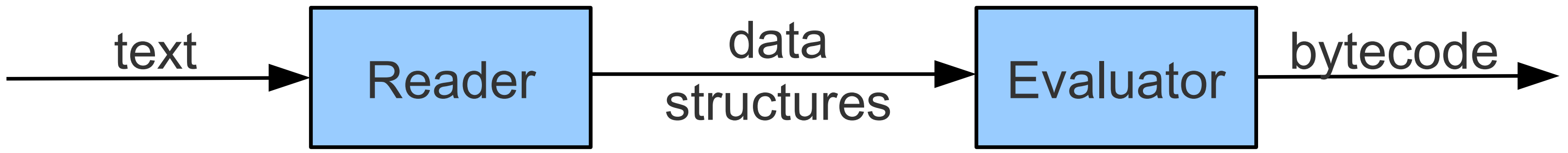
## Compojure

```
(def get-handler
  (GET "/hello" []
    "Hello, World!"))

(get-handler {:request-method :get
              :uri "/hello"})
> {:body "Hello, World!" ...}

(get-handler {:request-method :post
              :uri "/hello"})
> nil
```

```clojure
(def get-handler
  (GET "/hello/:name" [name]
    (str "Hello, " name "!")))

(def post-handler
  (POST "/names" [name]
    (remember name)
    (redirect (str "/hello/" name))))
```

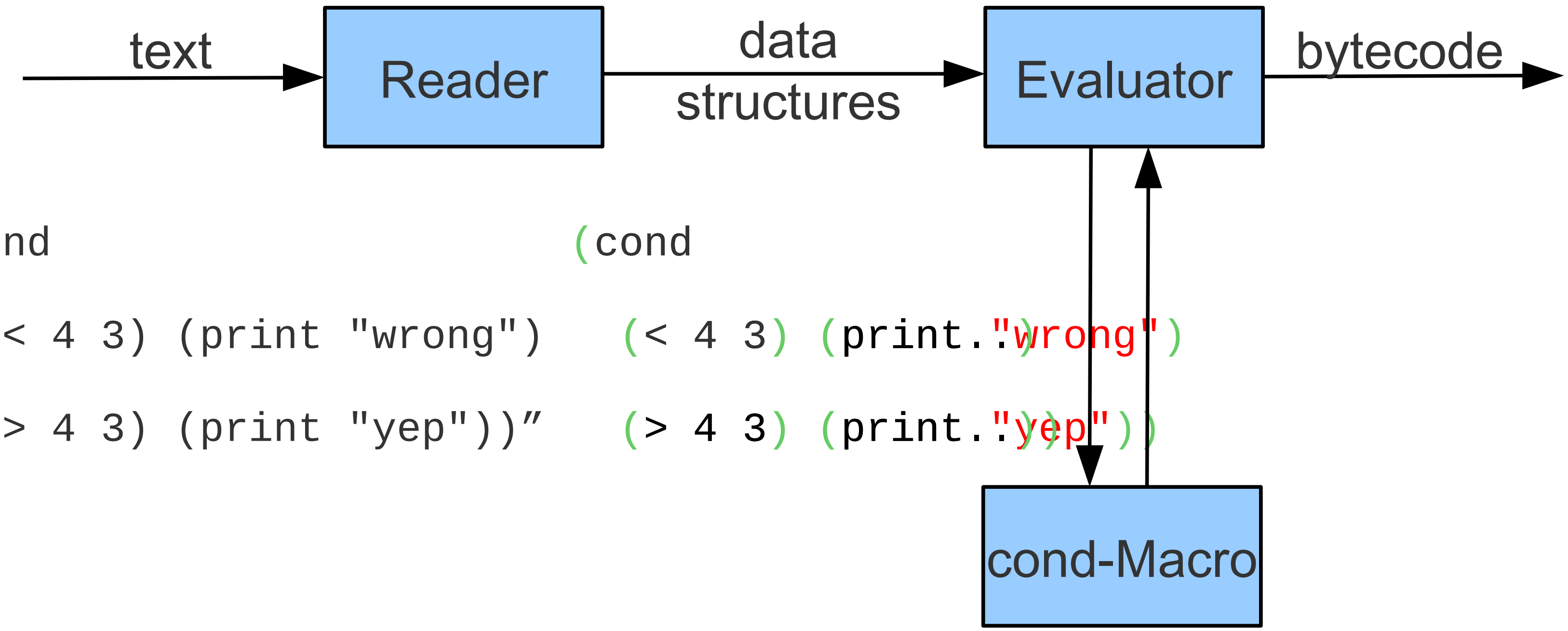
## Digression: Macros


text
Compiler
bytecode



```
“(if true
   "this is true"
   "this is false")”
```

```
(if true
  "this is true"
  "this is false")
```

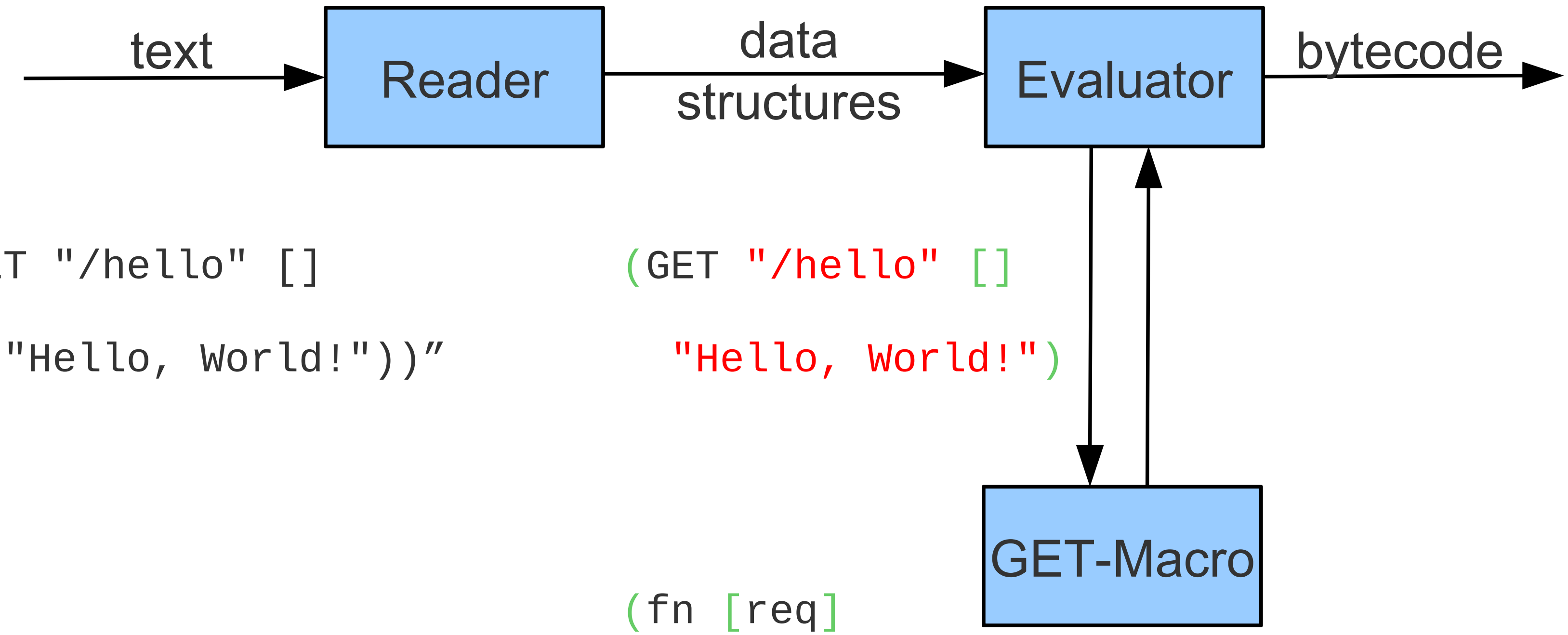


```
“(cond
   (< 4 3) (print "wrong")
   (> 4 3) (print "yep"))”
```

```
(cond
   (< 4 3) (print "wrong")
   (> 4 3) (print "yep"))
```

```
(if (< 4 3)
  (print "wrong")
  (if (> 4 3)
    (print "yep")
    nil))
```

```
(defmacro my-cond [c1 e1 c2 e2]
  (list 'if c1
        e1
        (list 'if c2
              e2
              nil)))


(my-cond
  false (println "won't see this")
  true (println "it works!"))
it works!
```



```
“(GET "/hello" []
    "Hello, World!"))”
```

```
(GET "/hello" []
  "Hello, World!")
```

```
(fn [req]
  (if (and (match (:uri req) "/hello")
           (= (:request-method req) :get))
    {:body "Hello, World!" …}
    nil))
```

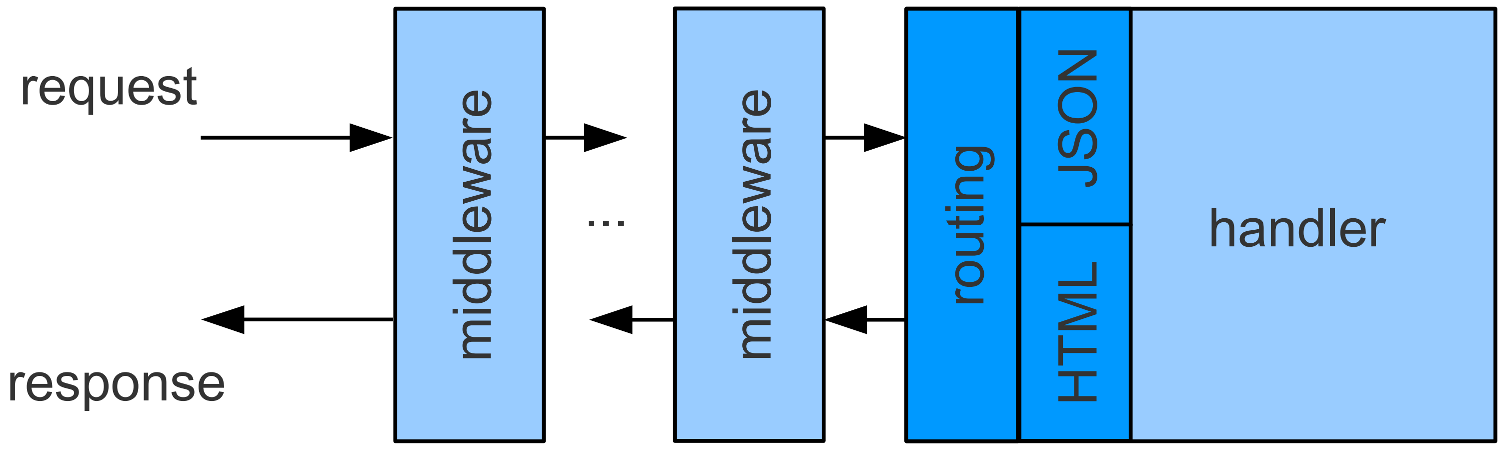
## Back to Compojure…

```
(def get-handler
  (GET "/hello/:name" [name]
    (str "Hello, " name "!")))

(def post-handler
  (POST "/names" [name]
    (remember name)
    (redirect (str "/hello/" name))))
```

```
(defroutes todo-app
  (GET "/todos" []
    (render (load-all-todos)))
  (GET "/todos/:id" [id]
    (render (load-todo id)))
  (POST "/todos" {json-stream :body}
    (create-todo (read-json (slurp json-stream)))
    (redirect "/todos")))
```

```
(defroutes more-routes
  (context "/todos/:id" [id]
    (DELETE "/" []
      (delete-todo id)
      (redirect "/todos"))
    (PUT "/" {json-stream :body}
      (update-todo (read-json (slurp json-stream)))
      (redirect (str "/todos/" id)))))
```

```
(defroutes complete-app
  todo-app
  more-routes
  (not-found "Oops."))

(def secure-app
  (wrap-basic-authentication complete-app allowed?))

(run-jetty (api secure-app) {:port 8080})
```


request
response
middleware
...
middleware
routing
JSON
HTML
handler

## Hiccup

```
<element attribute="foo">
  <nested>bar</nested>
</element>
```

```
[:element]
```

```
<element attribute="foo">
  <nested>bar</nested>
</element>
```

```
[:element {:attribute "foo"}]
```

```
<element attribute="foo">
  <nested>bar</nested>
</element>
```

```
[:element {:attribute "foo"}
  [:nested]]
```

```
<element attribute="foo">
  <nested>bar</nested>
</element>
```

```
[:element {:attribute "foo"}
  [:nested "bar"]]
```

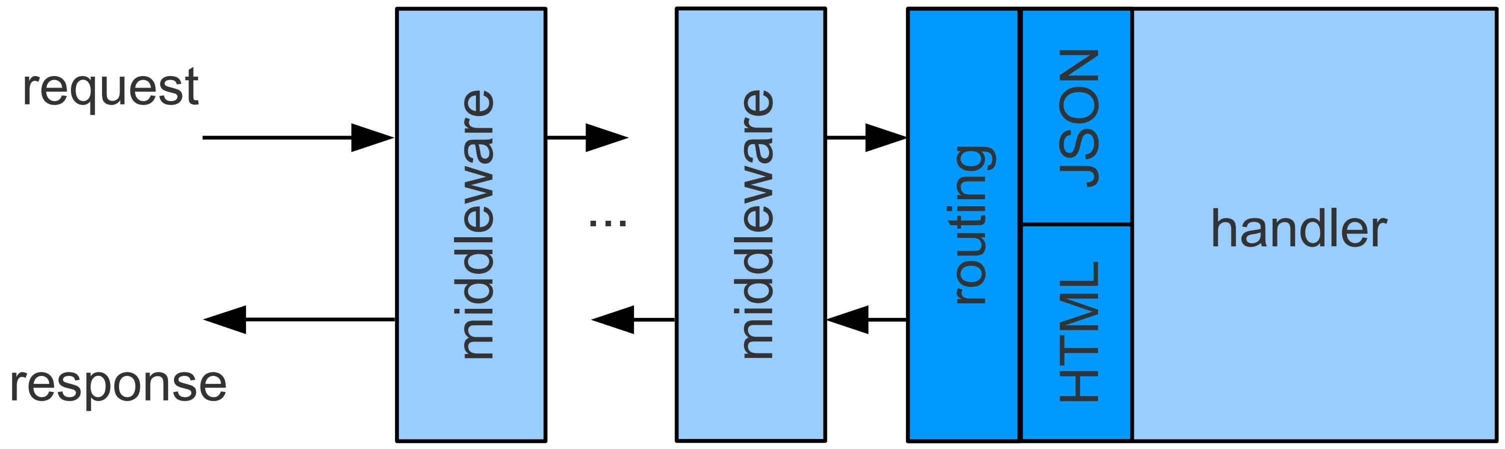
```
<html>
  <head><title>Foo</title></head>
  <body><p>Bar</p></body>
</html>


(def hiccup-example
  [:html
    [:head [:title "Foo"]]
    [:body [:p "Bar"]]]])


(html hiccup-example)
> "<html>...</html>"
```

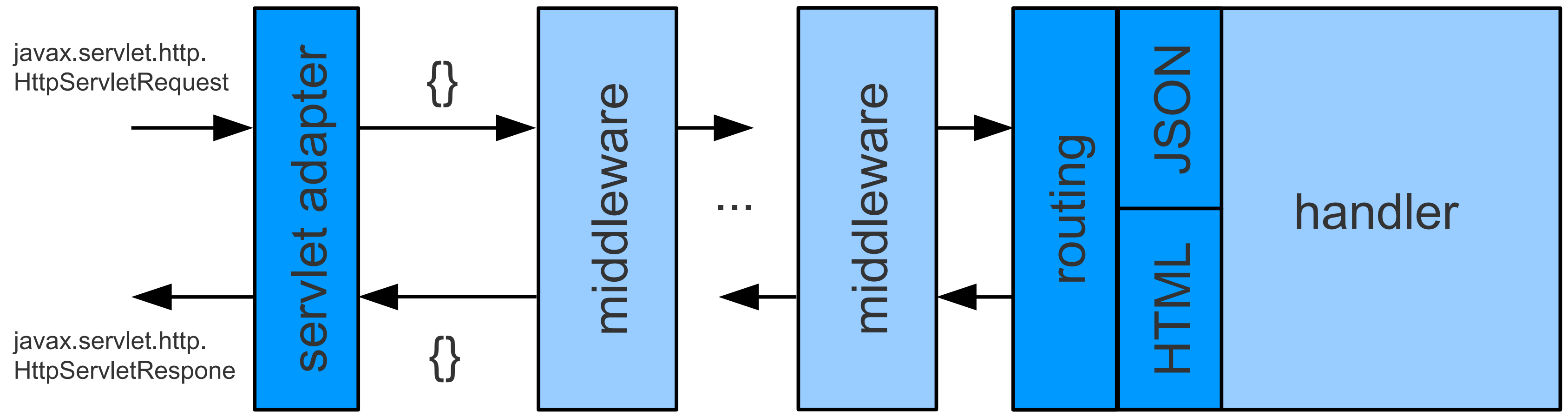
```
(def paul {:name "Paul" :age 45})

(defn render-person [person]
  [:dl
    [:dt "Name"] [:dd (:name person)]
    [:dt "Age"] [:dd (:age person)]])

(html (render-person paul))
> "<dl><dt>Name</dt><dd>Paul</dd>...</dl>"
```

```
(link-to "http://www.innoq.com" "click here")
> [:a {:href "http://www.innoq.com"} "click here"]


(form-to [:post "/login"]
  (text-field "Username")
  (password-field "Password")
  (submit-button "Login"))
> [:form {:action "POST" ...} [:input ...] ...]
```

```
<div id="my-id" class="class1 class2">
  foo
</div>
```

```
[:div#my-id.class1.class2 "foo"]
```


request
response
middleware
...
middleware
routing
JSON
HTML
handler


javax.servlet.http.
HttpServletRequest
javax.servlet.http.
HttpServletRespone
servlet adapter
{}
{}
middleware
...
middleware
routing
JSON
HTML
handler

## Noir

# www.webnoir.org

- Integration of Ring/Compojure/Hiccup
- `defpage` for defining routes
- Some middleware preconfigured
  - Parameter parsing, static resources, 404 page etc.
- Helpers for form validation etc.
- Auto reload in dev mode

innoQ

## Conclusion

## :-)

- Simple basic concepts
- Easy to use
- Little code (also in libraries)
- Helpful community
- Mature eco system

## Thank you!

Philipp Schirmacher
philipp.schirmacher@innoq.com
http://www.innoq.com
Phone: +49 151 4673 2018

**We'll take care of it. Personally.**

| **Task** | **Libraries** |
|---|---|
| HTTP Basics | Ring |
| Routing | Compojure<br>Moustache |
| HTML | Hiccup<br>Enlive |
| Persistence | clojure.java.jdbc<br>Korma<br>CongoMongo |
| Asynchronous Server | Aleph |
| JavaScript | ClojureScript |
| Micro Framework | Ringfinger<br>Noir |