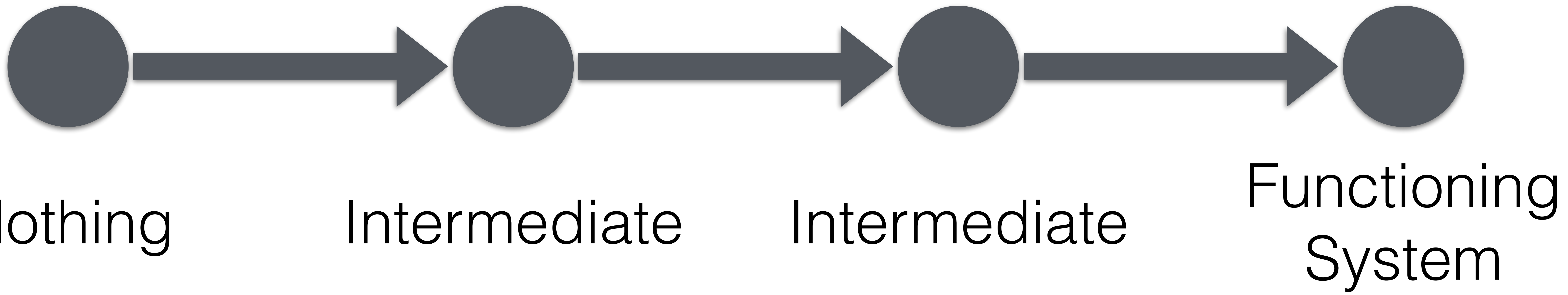
Learning through Libraries

Learning HTTP via a low-level web application

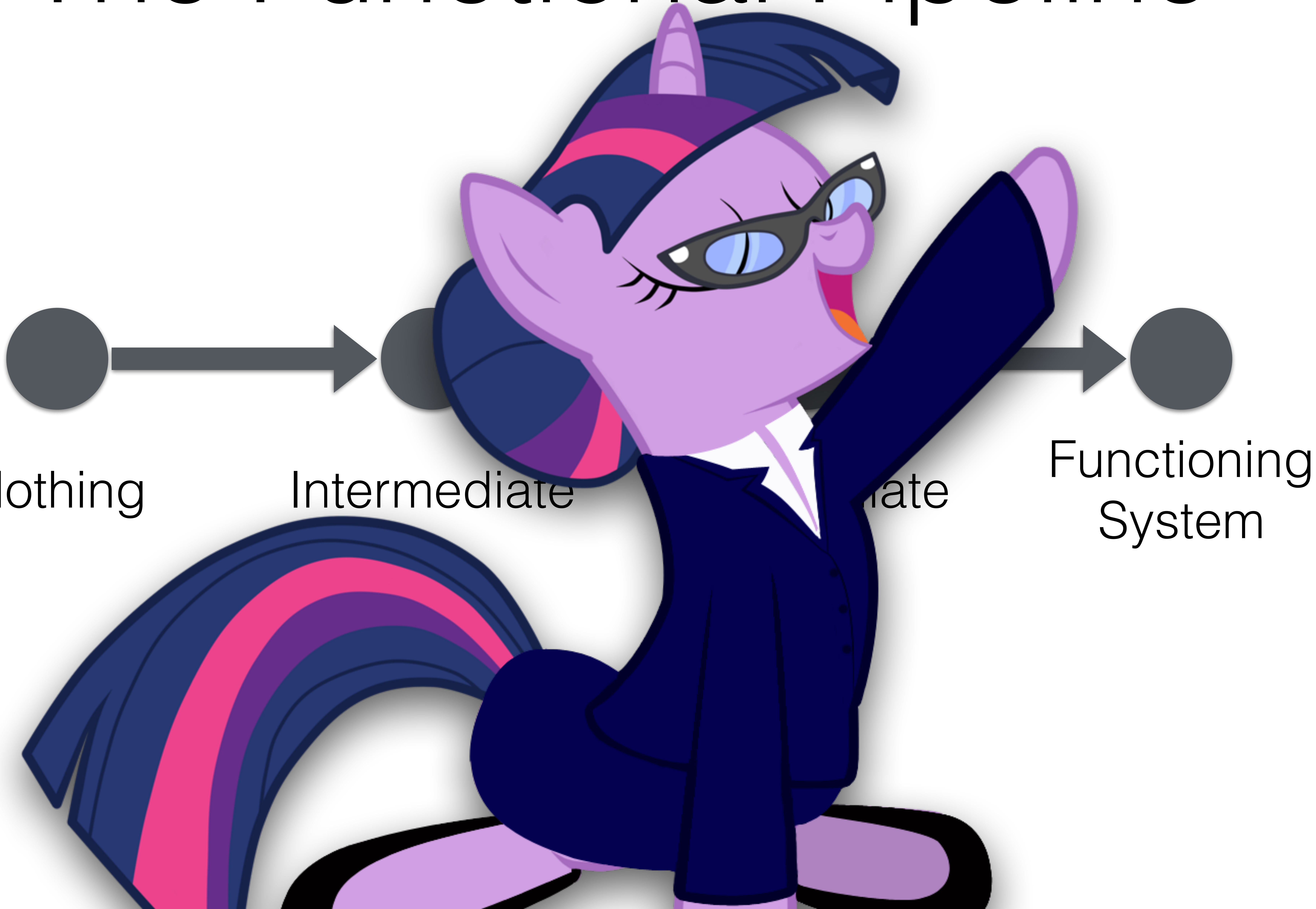
A good software library…

- **Teaches you** about the domain it models
- **Makes it easy** to build a correct system via composed higher-level functions
- **Doesn’t restrict you** from accessing lower-level functions

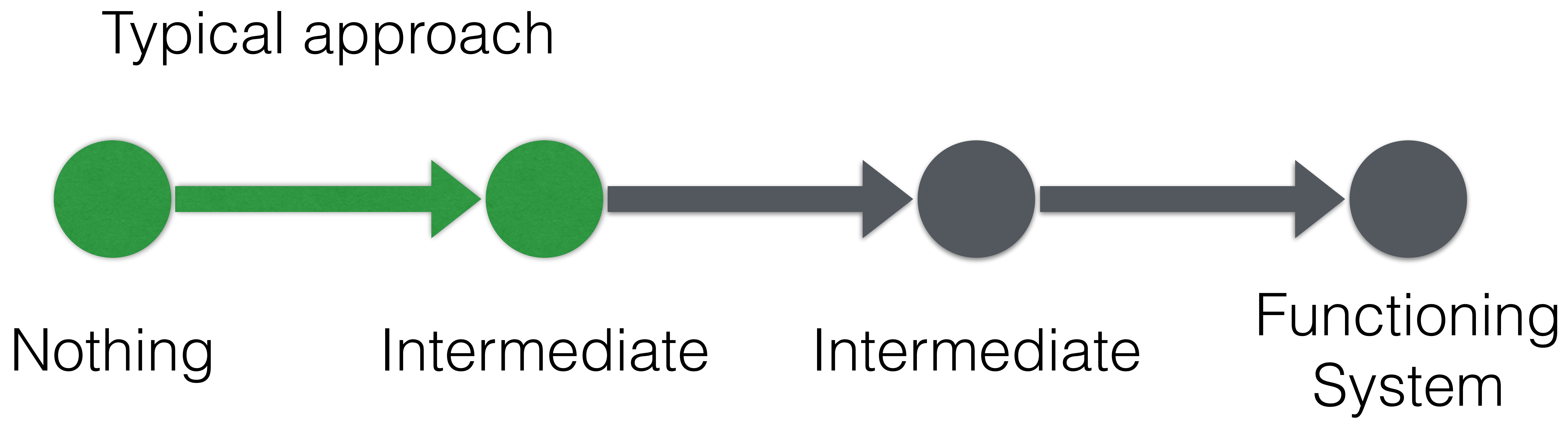
The Functional Pipeline



The Functional Pipeline

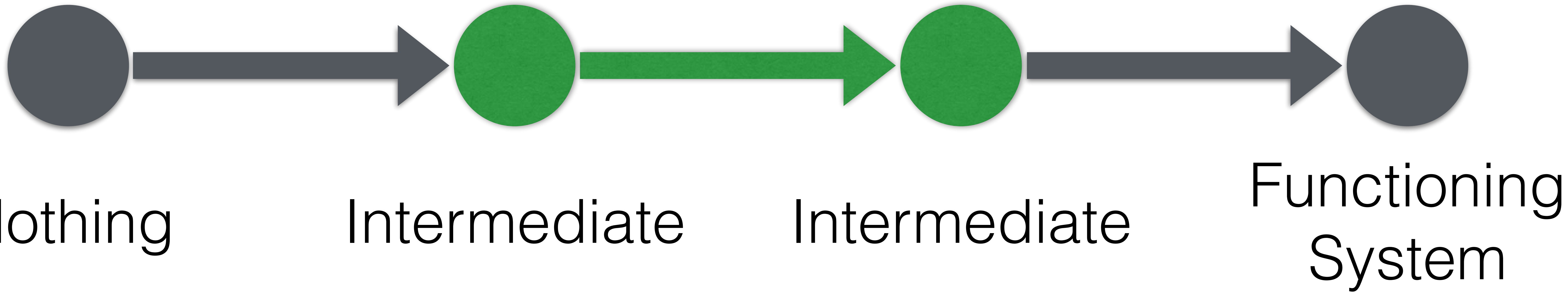


The Functional Pipeline

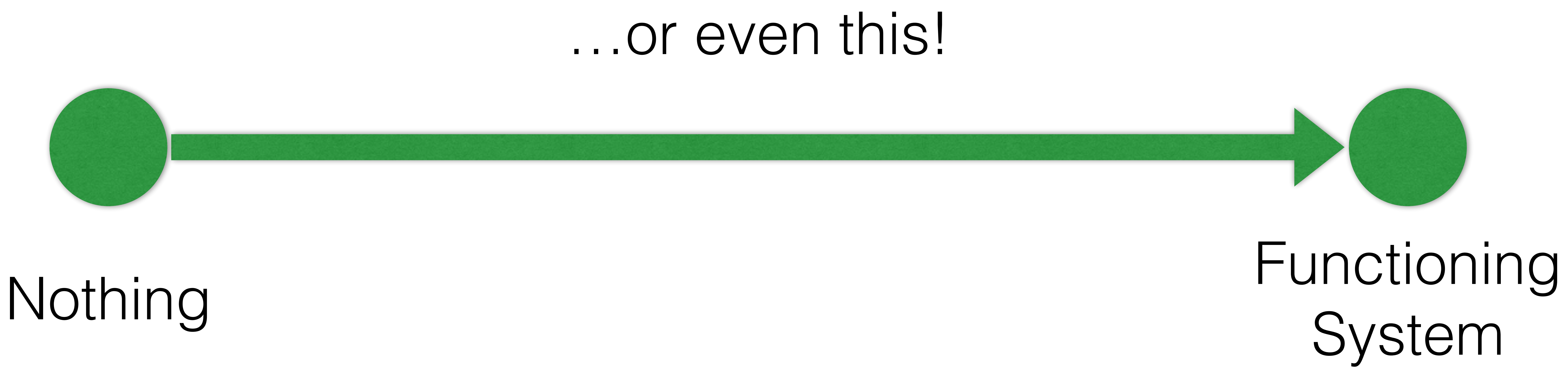


The Functional Pipeline

…but no reason we can't start with this



The Functional Pipeline



Types help us to prototype for little additional effort

Some libraries we're using:

```haskell
-- standard HTTP data types
import qualified Network.HTTP.Types as Http

-- URI parsing
import qualified Network.URI as Uri

-- the interface our application will follow
import qualified Network.Wai as Wai

-- the server for our application
import qualified Network.Wai.Handler.Warp as Warp
```

What we’re building

```haskell
type Wai.Application = Wai.Request
                     -> (Wai.Response -> IO Wai.ResponseReceived)
                     -> IO Wai.ResponseReceived

myApp :: Wai.Application
myApp request responder = undefined
```

How we’re running it

```
Warp.run :: Port -> Wai.Application -> IO ()

type Port = Int
```

Seems easy enough!

```haskell
main :: IO ()
main = Warp.run 1337 myApp

myApp :: Wai.Application
myApp request responder = undefined
```

The code compiles!

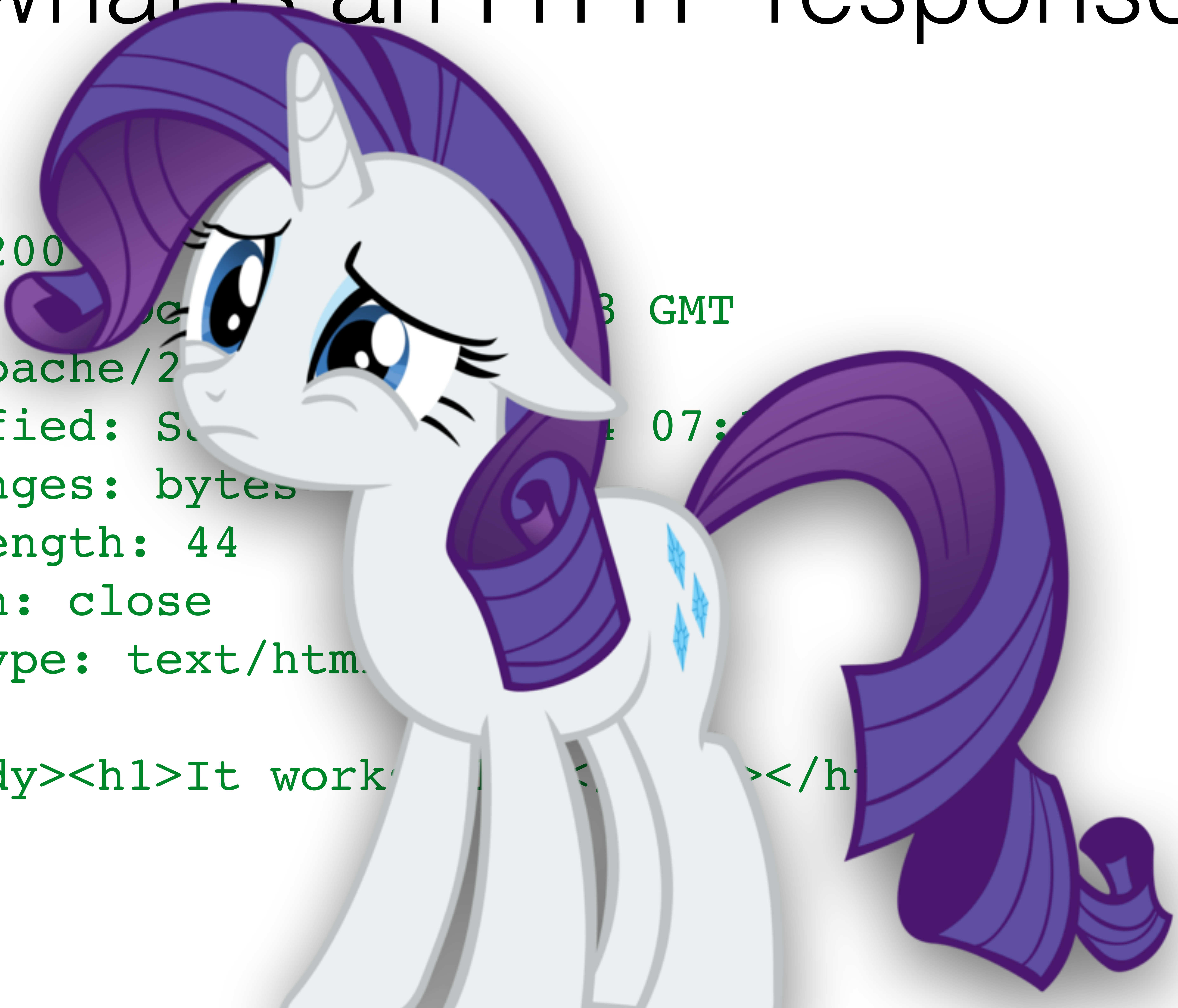
The code compiles!

Okay, but what is an HTTP response?

```
HTTP/1.1 200 OK
Date: Sun, 18 Oct 2009 08:56:53 GMT
Server: Apache/2.2.14
Last-Modified: Sat, 20 Nov 2004 07:16:26 GMT
Accept-Ranges: bytes
Content-Length: 44
Connection: close
Content-Type: text/html

<html><body><h1>It works</h1></body></html>
```

Okay, but what is an HTTP response?

```
HTTP/1.1 200
Date: Sun,                     3 GMT
Server: Apache/2
Last-Modified: S                07:
Accept-Ranges: bytes
Content-Length: 44
Connection: close
Content-Type: text/htm

<html><body><h1>It work              ></h
```



Let’s look at the types, instead…

```
Wai.responseLBS :: Status -> [Header] -> ByteString -> Response

data Status = Status { statusCode :: Int, statusMessage :: ByteString }

type Header = (HeaderName, ByteString)
type HeaderName = CI ByteString
```

We can build that!

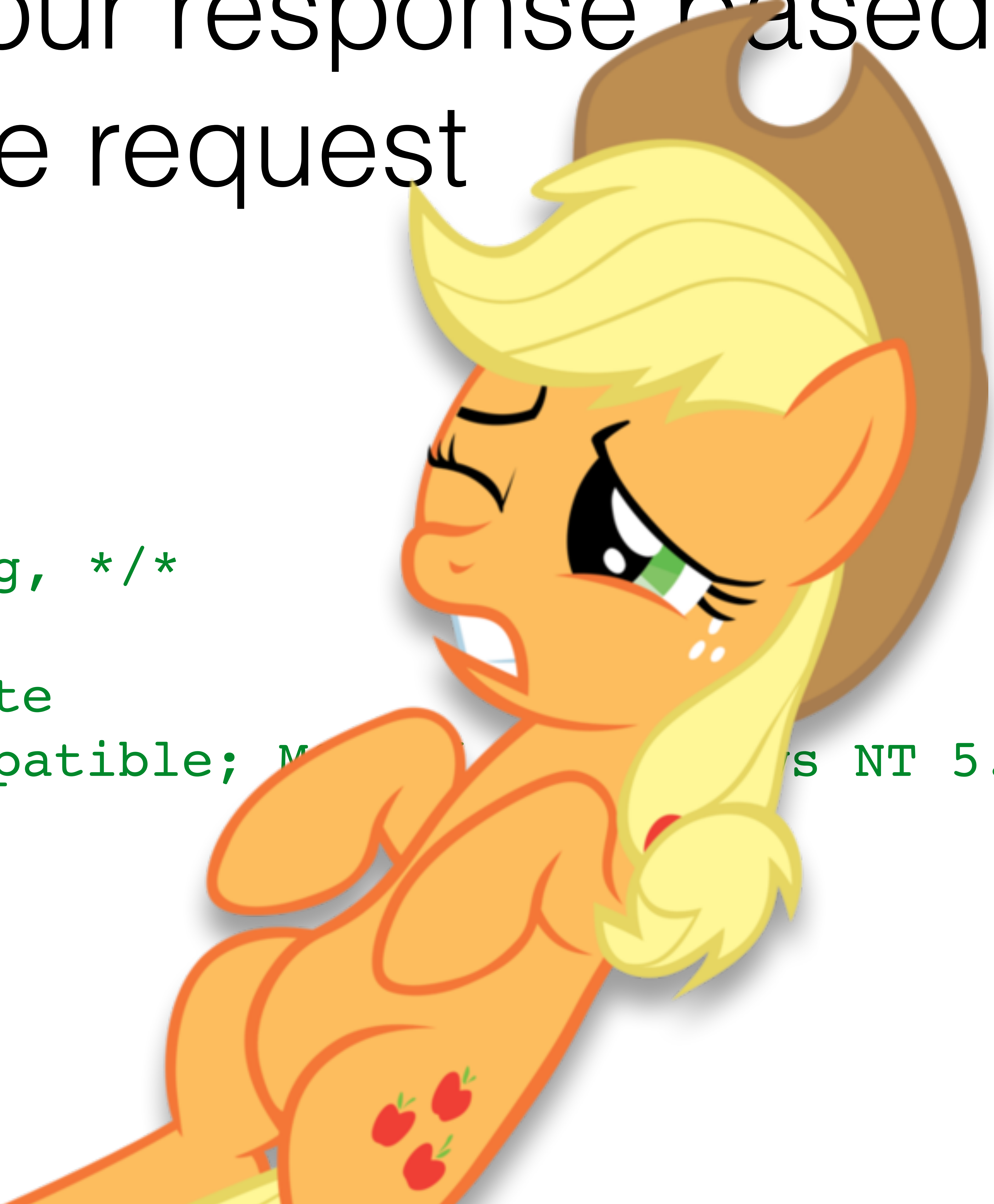
```haskell
main :: IO ()
main = Warp.run 1337 myApp

myApp :: Wai.Application
myApp request responder = responder myResponse

myResponse :: Wai.Response
myResponse = Wai.responseLBS status headers body
  where
    status  = Http.status200
    headers = [(Http.hContentType, "text/plain")]
    body    = "Woohoo it works!"
```

…but we want our response based on the request

```
GET /index.html HTTP/1.1
Host: www.rawhttprequest.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
<blank line>
```

…but we want our response based on the request

```
GET /index.html HTTP/1.1
Host: www.rawhttprequest.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; M          s NT 5.1)
<blank line>
```



That’s better!

```haskell
data Wai.Request = Wai.Request
  { requestMethod        :: Http.Method
  , httpVersion          :: Http.HttpVersion
  , requestHeaders       :: Http.RequestHeaders
  , isSecure             :: Bool
  , remoteHost           :: SockAddr
  , pathInfo             :: [Text]
  , queryString          :: Http.Query
  , requestBody          :: IO ByteString
  -- and a whole lot more...
  }
```

That’s better!

```
data Wai.Request = Wai.Request
  { requestMethod          :: Http.Method
  , httpVersion            :: Http.HttpVersion
  , requestHeaders         :: Http.RequestHeaders
  , isSecure               :: Bool
  , remoteHost             :: SockAddr
  , pathInfo               :: [Text]
  , queryString            :: Http.Query
  , requestBody            :: IO ByteString
  -- and a whole lot more...
  }
```

We can make a router!

```haskell
router :: [Text] -> Http.StdMethod -> Wai.Response
router = undefined
```

We can make a router!

```haskell
myApp :: Wai.Application
myApp request responder =
    case method of
      Right m -> responder $ router path m
      Left _   -> error "unknown request method"
  where
    path = Wai.pathInfo request
    method = Http.parseMethod $ Wai.requestMethod request

router :: [Text] -> Http.StdMethod -> Wai.Response
router = undefined
```

Let’s make routes for a RESTful resource

```
router :: Http.StdMethod -> [Text] -> Wai.Response
router Http.GET    ["resources"]      = undefined -- index
router Http.POST   ["resources"]      = undefined -- create
router Http.GET    ["resources", rid] = undefined -- show
router Http.PUT    ["resources", rid] = undefined -- update
router Http.DELETE ["resources", rid] = undefined -- delete
```

…and controller actions to call

```
indexAction :: Wai.Response
indexAction = undefined

createAction :: Wai.Response
createAction = undefined

showAction :: Int -> Wai.Response
showAction rid = undefined

updateAction :: Int -> Wai.Response
updateAction rid = undefined

deleteAction :: Int -> Wai.Response
deleteAction rid = undefined
```

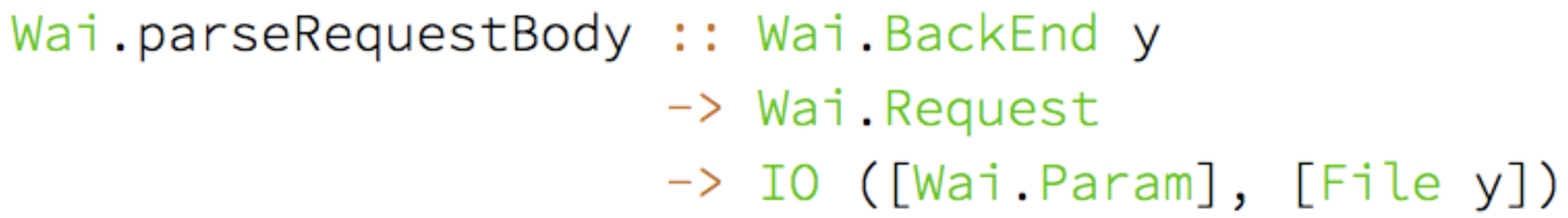
…and controller actions to call

```
      Wai.Response
      defined

      ai.Respo
creat    define

showActi
sho         ned

    Acti       Wai.Respo
updateAct   rid =     ned

deleteAc    :: In     Response
deleteAc    rid       d
```

We still need request parameters to modify resources…

```haskell
data Wai.Request = Wai.Request
  { requestMethod          :: Http.Method
  , httpVersion            :: Http.HttpVersion
  , requestHeaders         :: Http.RequestHeaders
  , isSecure               :: Bool
  , remoteHost             :: SockAddr
  , pathInfo               :: [Text]
  , queryString            :: Http.Query
  , requestBody            :: IO ByteString
  -- and a whole lot more...
  }
```

What the fu…



```
Wai.parseRequestBody :: Wai.BackEnd y
                     -> Wai.Request
                     -> IO ([Wai.Param], [File y])
```

- There’s something new! A request body also contains encodings for uploaded files. Huh!
- Also, what the hell is a “backend” ?
- We don’t care about files for now, so let’s just find a function that fits the type and move on

This seems to work with minimal hassel!

```haskell
requestParams :: Wai.Request -> IO [Param]
requestParams request = do
    (params, _) <- Wai.parseRequestBody Wai.lbsBackEnd request
    return params
```

…but let's get fancy!

```haskell
type Params = Map.Map ByteString ByteString

requestParams :: Wai.Request -> IO Params
requestParams request = do
    (params, _) <- Wai.parseRequestBody Wai.lbsBackEnd request
    return $ paramListToMap params

paramListToMap :: [Wai.Param] -> Params
paramListToMap = foldl' insert Map.empty
  where
    insert params (name, val) = Map.insert name val params
```

…but let t fancy!

```
type Params = Map.Map ByteStrin         ng

requestParams :: Wai.Request
requestParams reque
    (params, _) <      rs            Wai.lbsBackEnd request
    return $ pa

paramListToMap ::        ra        a
paramListToMap =        n   t M
  where
    insert params   ame,   ) =    .insert name val params
```

…and the resulting application

```haskell
myApp :: Wai.Application
myApp request responder =
    case method of
      Right m -> do
        params <- requestParams request
        responder $ router m path params
      Left _   -> error "unknown request method"
  where
    path = Wai.pathInfo request
    method = Http.parseMethod $ Wai.requestMethod request
```

…and the resulting application

```haskell
myApp :: Wai.Application
myApp request responder =
    case method of
      Right m -> do
        params <- requestParams request
        responder $ router m path params
      Left _   -> error "unknown request method"
  where
    path = Wai.pathInfo request
    method = Http.parseMethod $ Wai.requestMethod request
```

…and the resulting router

```haskell
router :: Http.StdMethod -> [Text.Text] -> Params -> Wai.Response
router Http.GET    ["resources"]      = indexAction
router Http.POST   ["resources"]      = createAction
router Http.GET    ["resources", rid] = showAction (fromText rid)
router Http.PUT    ["resources", rid] = updateAction (fromText rid)
router Http.DELETE ["resources", rid] = deleteAction (fromText rid)
```

(thanks, currying!)

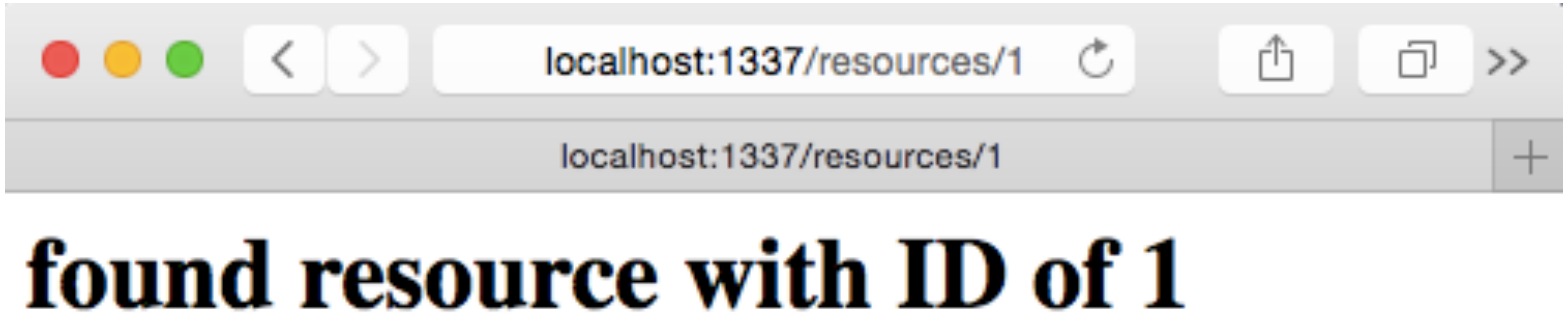
…and the resulting controller actions

```haskell
indexAction :: Params -> Wai.Response
indexAction params = undefined

createAction :: Params -> Wai.Response
createAction params = undefined

showAction :: Int -> Params -> Wai.Response
showAction rid params = undefined

updateAction :: Int -> Params -> Wai.Response
updateAction rid params = undefined

deleteAction :: Int -> Params -> Wai.Response
deleteAction rid params = undefined
```

Finally, a working action:

```haskell
showAction :: Int -> Params -> Wai.Response
showAction rid params = htmlResponse $
    "<h1>found resource with ID of " `mappend`
    (C8.pack $ show rid)              `mappend`
    "</h1>"

htmlResponse :: C8.ByteString -> Wai.Response
htmlResponse content = Wai.responseLBS status headers body
  where
    status  = Http.status200
    headers = [(Http.hContentType, "text/html")]
    body    = LazyBS.fromStrict content
```

…and the result!


localhost:1337/resources/1

localhost:1337/resources/1

found resource with ID of 1

What have we learned?

- What information we can get from an HTTP request
- How to minimally compose an HTTP response
- The data format of a POST request
- How to route based on HTTP request data

What have we learned?

- What information ar P request
- How to minimally c esponse
- What's inside of a POST re
- How on HT est data