

Если вы видите что-то необычное, просто сообщите мне.

# Purescript Part 4: Web Requests and Navigation

Welcome to the conclusion of our series on Purescript! We've spent a lot of time now learning to use functional languages for frontend web. In part 3, we saw how to build a basic UI with Purescript. We made a simple counter and then a todo list application, as we did with Elm. This week, we'll explore two more crucial pieces of functionality. We'll see how to send web requests and how to provide different routes for our application.

There are two resources you can look at if you want more details on how this code works. First, you can look at our Github repository. You can also explore the Halogen Github repository. Take a look at the driver-routing and effects-ajax example.

## WEB REQUESTS

For almost any web application, you're going to need to retrieve some data from a backend server. We'll use the `purescript-affjax` library to make requests from our Halogen components. The process is going to be a little simpler than it was with Elm.

In Elm, we had to hook web requests into our architecture using the concept of commands. But Purescript's syntax uses monads by nature. This makes it easier to work effects into our `eval` function.

In this first part of the article, we're going to build a simple web UI that will be able to send a couple requests. As with all our Halogen components, let's start by defining our state, message, and query types:

```

type State =
  { getResponse :: String
  , postInfo :: String
  }

initialState :: State
initialState =
  { getResponse: "Nothing Yet"
  , postInfo: ""
  }

data Query a =
  SendGet a |
  SendPost a |
  UpdatedPostInfo String a

data Message = ReceivedFromPost String

```

We'll store two pieces of information in the state. First, we'll store a "response" we get from calling a get request, which we'll initialize to a default string. Then we'll store a string that the user will enter in a text field. We'll send this string through a post request. We'll make query constructors for each of the requests we'll send. Then, our message type will allow us to update our application with the result of the post request.

We'll initialize our component as we usually do, except with one difference. In previous situations, we used an unnamed `m` monad for our component stack. This time, we'll specify the `Aff` monad, enabling our asynchronous messages. This monad also gets applied to our `eval` function.

```

webSender :: H.Component HH.HTML Query Unit Message Aff
webSender = H.component
  { initialState: const initialState
  , render
  , eval
  , receiver: const Nothing
  }

render :: State -> H.ComponentHTML Query
...

```

```
eval :: Query ~> H.ComponentDSL State Query Message Aff
```

```
...
```

Our UI will have four elements. We'll have a `p` field storing the response from our get request, as well as a button for triggering that request. Then we'll have an input field where the user can enter a string. There will also be a button to send that string in a post request. These all follow the patterns we saw in part 3 of this series, so we won't dwell on the specifics:

```
render :: State -> H.ComponentHTML Query
render st = HH.div [] [progressText, getButton, inputText, postButton]
  where
    progressText = HH.p [] [HH.text st.getResponse]
    getButton = HH.button
      [ HP.title "Send Get", HE.onClick (HE.input_ SendGet) ]
      [ HH.text "Send Get" ]
    inputText = HH.input
      [ HP.type_ HP.InputText
      , HP.placeholder "Form Data"
      , HP.value st.postInfo
      , HE.onValueChange (HE.input UpdatedPostInfo)
      ]
    postButton = HH.button
      [ HP.title "Send Post", HE.onClick (HE.input_ SendPost) ]
      [ HH.text "Send Post" ]
```

Our `eval` function will assess each of the different queries we can receive, as always. When updating the post request info (the text field), we update our state with the new value.

```
eval :: Query ~> H.ComponentDSL State Query Message Aff
eval = case _ of
  SendGet next -> ...
  SendPost next -> ...
  UpdatedPostInfo newInfo next -> do
    st <- H.get
    H.put (st { postInfo = newInfo })
    pure next
```

Now let's specify our get request. The `get` function from the `Affjax` library takes two parameters. First we need a "deserializer", which tells us how to convert the response into some desired type.

We'll imagine we're getting a String back from the server, so we'll use the string deserializer. The our second parameter is the URL. This will be a localhost address. We call liftAff to get this Aff call into our component monad.

```
import Affjax as AX
import Affjax.ResponseFormat as AXR

eval :: Query ~> H.ComponentDSL State Query Message Aff
eval = case _ of
  SendGet next -> do
    response <- H.liftAff $ AX.get AXR.string "http://localhost:8081/api/hello"
    ...
  SendPost next -> ...
  UpdatedPostInfo newInfo next -> ...
```

The response contains a lot of information, including things like the status code. But our main concern is the response body. This is an Either value giving us a success or error value. In either case, we'll put a reasonable value into our state, and call the next action!

```
eval :: Query ~> H.ComponentDSL State Query Message Aff
eval = case _ of
  SendGet next -> do
    response <- H.liftAff $ AX.get AXR.string "http://localhost:8081/api/hello"
    st <- H.get
    case response.body of
      Right success -> H.put (st { getResponse = success })
      Left _ -> H.put (st { getResponse = "Error!" })
    pure next
  SendPost next -> ...
  UpdatedPostInfo newInfo next -> ...
```

Then we can go to our UI, click the button, and it will update the field with an appropriate value!

# POST REQUESTS

Sending a post request will be similar. The main change is that we'll need to create a body for our post request. We'll do this using the "Argonaut" library for Purescript. The fromString function gives

us a JSON object. We wrap this into a `RequestBody` with the `json` function:

```
import Affjax.RequestBody as AXRB
import Data.Argonaut.Core as JSON

...

eval :: Query ~> H.ComponentDSL State Query Message Aff
eval = case _ of
  SendGet next -> ...
  SendPost next -> do
    st <- H.get
    let body = AXRB.json (JSON.fromString st.postInfo)
    ...
  UpdatedPostInfo newInfo next -> ...
```

Aside from adding this body parameter, the post function works as the get function does. We'll break the response body into `Right` and `Left` cases to determine the result. Instead of updating our state, we'll send a message about the result.

```
eval :: Query ~> H.ComponentDSL State Query Message Aff
eval = case _ of
  SendGet next -> ...
  SendPost next -> do
    st <- H.get
    let body = AXRB.json (JSON.fromString st.postInfo)
    response <- H.liftAff $ AX.post AXR.string "http://localhost:8081/api/post" body
    case response.body of
      Right success -> H.raise (ReceivedFromPost success)
      Left _ -> H.raise (ReceivedFromPost "There was an error!")
    pure next
  UpdatedPostInfo newInfo next -> ...
```

And that's the basics of web requests!

# ROUTING BASICS

Now let's change gears and consider how we can navigate among different pages. For the sake of example, let's say we've got 4 different types of pages in our app.

1. A home page
2. A login page
3. A user profile page
4. A page for each article Each user profile will have an integer user ID attached to it. Each article will have a string identifier attached to it as well as a user ID for the author. Here's a traditional router representation of this:

```
/home  
/login  
/profile/:userid  
/blog/articles/:userid/:articleid
```

With the Purescript Routing library, our first step is to represent our set of routes with a data type. Each route will represent a page on our site, so we'll call our type `Page`. Here's how we do that:

```
data Page =  
  HomePage |  
  LoginPage |  
  ProfilePage Int |  
  ArticlePage Int String
```

By using a data structure, we'll be able to ensure two things. First, all the routes in our application have some means of handling them. If we're missing a case, the compiler will let us know. Second, we'll ensure that our application logic cannot route the user to an unknown page. We will need to use one of the routes within our data structure.

## BUILDING A PARSER

That said, the user could still enter any URL they want in the address bar. So we have to know how to parse URLs into our different pages. For this, we have to build a parser on our route type. This will have the type `Match Page`. This will follow an applicative parsing structure. For more background on this, check out this article from our parsing series!

But even if you've never seen this kind of parsing before, the patterns aren't too hard. The first thing to know is that the `lit` function (meaning literal) matches a string path component. So we feed it the string element we want, and it will match our route.

For our home page route, we'll want to first match the URL component "home".

```
import Routing.Match (Match, lit, int, str)

matchHome = lit "home"
```

But this will actually give us a `Match` that outputs a `String`. We want to ignore the string we parsed, and give a constructor of our `Page` type. Here's what that looks like:

```
matchHome :: Match Page
matchHome = HomePage <$ lit "home"
```

The `<$ data-preserve-html-node="true"` operator tells us we want to perform a functor wrap. Except we want to ignore the resulting value from the second part. This gives our first match!

The login page will have a very similar matcher:

```
matchLogin :: Match Page
matchLogin = LoginPage <$ lit "login"
```

But then for the profile page, we'll actually want to use the result from one of our matchers! We want to use `int` to read the integer out of the URL component and plug it into our data structure. For this, we need the applicative operator `<*>`. Except once again, we'll have a string part that we ignore, so we'll actually use `*>`. Here's what it looks like:

```
matchProfile :: Match Page
matchProfile = ProfilePage <$> (lit "profile" *> int)
```

Now for our final matcher, we'll keep using these same ideas! We'll use the full applicative operator `<*>` since we want both the user ID and the article ID.

```
matchArticle :: Match Page
matchArticle = ArticlePage <$>
  (lit "blog" *> lit "articles" *> int) <*> string
```

Now we combine our different matchers into a router by using the `<|>` operator from Alternative:

```
router :: Match Page
router = matchHome <|> matchLogin <|> matchProfile <|> matchArticle
```

And we're done! Notice how similar Purescript and Haskell are in this situation! Pretty much all the code from this section could work in Haskell. (As long as we used the corresponding libraries).

# INCORPORATING OUR ROUTER

Now to use this routing mechanism, we're going to need to set up our application in a special way. It will have one single parent component and several child components. We will make it so that our application can listen to changes in the URL. We'll use our router to match those changes to our URL scheme. Our parent component will, as always, respond to queries. We won't go through the details of our child components. You can take a look at `src/NavComponents.purs` in our Github repo for details there.

We'll use some special mechanisms to send a query on each route change event. Then our parent component will handle updating the view. An important thing to know is that all the child components have the same query and message type. We won't use these much in this article, but these are how you would customize app-wide behavior.

```
type ChildState = Int
data ChildQuery a = ChildQuery a
data ChildMessage = ChildMessage
```

Each child component will have a link to the "next page" in the sequence. This way, we can show how these links work once we render it. We'll need access to these component definitions in our parent module:

```
homeComponent :: forall m.
  H.Component HH.HTML ChildQuery Unit ChildMessage m
```



```
loginComponent :: forall m.
  H.Component HH.HTML ChildQuery Unit ChildMessage m

profileComponent :: forall m. Int ->
  H.Component HH.HTML ChildQuery Unit ChildMessage m

articleComponent :: forall m. Int -> String ->
  H.Component HH.HTML ChildQuery Unit ChildMessage m
```

# THE PARENT COMPONENT

Now let's start out by making a simple query type for our parent element. We'll have one query for changing the page, and one for processing messages from our children.

```
data ParentQuery a =
  ChangePage Page a |
  HandleAppAction Message a
```

The parent's state will include the current page. It could also include some secondary elements like the ID of the logged in user, if we wanted.

```
type ParentState = { currentPage :: Page }
```

Now we'll need slot designations for the "child" element of our page. Depending on the state of our application, our child element will be a different component. This is how we'll represent the different pages of our application.

```
data SlotId = HomeSlot | LoginSlot | ProfileSlot | ArticleSlot
```

Our eval and render functions should be pretty straightforward. When we evaluate the "change page" query, we'll update our state. Then we won't do anything when processing a ChildMessage:

```
eval :: forall m. ParentQuery ~>
  H.ParentDSL ParentState ParentQuery ChildQuery SlotId Void m

eval = case _ of
  ChangePage pg next -> do
```

```
H.put {currentPage: pg}  
pure next  
HandleAppAction _ next -> do  
  pure next
```

For our render function, we first need a couple helpers. The first goes from the page to the slot ID. The second gives a mapping from our page data structure to the proper component.

```
slotForPage :: Page -> SlotId  
slotForPage HomePage = HomeSlot  
slotForPage LoginPage = LoginSlot  
slotForPage (ProfilePage _) = ProfileSlot  
slotForPage (ArticlePage _ _) = ArticleSlot  
  
componentForPage :: forall m. Page ->  
  H.Component HH.HTML ChildQuery Unit Message m  
componentForPage HomePage = homeComponent  
componentForPage LoginPage = loginComponent  
componentForPage (ProfilePage uid) = profileComponent uid  
componentForPage (ArticlePage uid aid) = articleComponent uid aid
```

Now we can construct our render function. We'll access the page from our state, and then create an appropriate slot for it:

```
render :: forall m. ParentState ->  
  H.ParentHTML ParentQuery ChildQuery SlotId m  
render st = HH.div_  
  [ HH.slot sl comp unit (HE.input HandleAppAction)  
  ]  
  where  
    sl = slotForPage st.currentPage  
    comp = componentForPage st.currentPage
```

# ADDING ROUTING

Now to actually apply the routing in our application, we'll update our Main module. This process will be a little complicated. There are a lot of different libraries involved in reading event changes. We

won't dwell too much on the details, but here's the high level overview.

Every time the user changes the URL or clicks a link, this produces a HashChangeEvent. We want to create our own Producer that will listen for these events so we can send them to our application. Here's what that looks like:

```
import Control.Coroutine as CR
import Control.Coroutine.Aff as CRA
import Web.HTML (window) as DOM
import Web.HTML.Event.HashChangeEvent as HCE
import Web.HTML.Event.HashChangeEvent.EventTypes as HCET

hashChangeProducer :: CR.Producer HCE.HashChangeEvent Aff Unit
hashChangeProducer = CRA.produce \emitter -> do
  listener <- DOM.eventListener
    (traverse_ (CRA.emit emitter) <<< HCE.fromEvent)
  liftEffect $
    DOM.window
      >>= Window.toEventTarget
      >>> DOM.addEventListener HCET.hashchange listener false
```

Now we want our application to consume these events. So we'll set up a Consumer function. It consumes the hash change events and passes them to our UI, as we'll see:

```
hashChangeConsumer
  :: (forall a. ParentQuery a -> Aff a)
  -> CR.Consumer HCE.HashChangeEvent Aff Unit
hashChangeConsumer query = CR.consumer \event -> do
  let hash = Str.drop 1 $ Str.dropWhile (_ /= '#') $ HCE.newURL event
  result = match router hash
  newPage = case result of
    Left _ -> HomePage
    Right page -> page
  void $ liftAff $ query $ H.action (ChangePage newPage)
  pure Nothing
```

There are a couple things to notice. We drop the hash up until the # to get the relevant part of our URL. Then we pass it to our router for processing. Finally, we pass an appropriate ChangePage action to our UI.

How do we do this? Well, the first argument of this consumer function (query) is actually another function. This function takes in our ParentQuery and produces an Aff event. We can access this function as a result of the runUI function.

So our final step is to run our UI. Then we run a separate process that will chain the producer and consumer together:

```
main :: Effect Unit
main = HA.runHalogenAff do
  body <- HA.awaitBody
  io <- runUI parentComponent unit body
  CR.runProcess (hashChangeProducer CR.$$ hashChangeConsumer io.query)
```

We pass the io.query property of our application UI to the consumer, so our UI can react to the events. And now our application will respond to URL changes!

# CONCLUSION

This wraps up our series on Purescript! Between this and our Elm Series , you should have a good idea on how to use functional languages to write a web UI. As a reminder, you can see more details on running Purescript code on our Github Repository. The README will walk you through the basic steps of getting this code setup.

You can also take a look at some of our other resources on web development using Haskell! Read our Haskell Web Series to see how to write a backend for your application. You can also download our Production Checklist to learn about more libraries you can use.

---

Revision #1

Created 11 March 2022 16:57:33 by gasick

Updated 11 March 2022 17:11:17 by gasick