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

Purescript Part 3: Simple Web UI's

In part 2 of this series, we continued learning the basic elements of Purescript. We examined how typeclasses and monads work and the slight differences from Haskell. Now it's finally time to use Purescript for its main purpose: frontend web development. We'll accomplish this using the Halogen framework, built on React.js.

In this part, we'll learn about the basic concepts of Halogen/React. We'll build a couple simple components to show how these work. In the final part of this series, we'll conclude our look at Purescript by making a more complete application. We'll see how to handle routing and sending web requests.

If you're building a frontend, you'll also need a backend at some point. Check out our Haskell Web Series to learn how to do that in Haskell!

Also, getting Purescript to work can be tricky business! Take a look at our Github repository for some more setup instructions!

HALOGEN CRASH COURSE

The Halogen framework uses React.js under the hood, and the code applies similar ideas. If you don't do a lot of web development, you might not be too familiar with the details of React. Luckily, there are a few simple principles we'll apply that will remind us of Elm!

With Halogen, our UI consists of different "components". A component is a UI element that maintains its own state and properties. It also responds to queries, and sends messages. For any component, we'll start by defining a a state type, a query type, and a message type.

```
data CState = ...

data CQuery = ...

data CMessage = ...
```

Our component receives queries from within itself or from other components. It can then send messages to other components, provided they have queries to handle them. With these types in place, we'll use the component function to define a component with 3 main elements. As a note, we'll be maintaining these import prefixes throughout the article.

```
import Halogen as H
import Halogen.HTML as HH
import Halogen.Events as HE
import Halogen.Properties as HP

myComponent :: forall m.
  H.Component HH.HTML CQuery Unit CMessage m
myComponent = H.component
  { initialState: ...
  , render: ...
  , eval: ...
  , receiver: const Nothing
  }

  where

    render ::
      CState ->
      H.ComponentHTML CQuery

    eval ::
      CQuery ~>
      H.ComponentDSL CState CQuery CMessage m
```

The initialState is self explanatory. The render function will be a lot like our view function from Elm. It takes a state and returns HTML components that can send queries. The eval function acts like our update function in Elm. Its type signature looks a little strange. But it takes queries as inputs and

can update our state using State monad function. It can also emit messages to send to other components.

BUILDING A COUNTER

For our first example of a component, we'll make a simple counter. We'll have an increment button, a decrement button and a display of the current count. Our state will be a simple integer. Our queries will involve events from incrementing and decrementing. We'll also send a message each time we update our number.

```
type State = Int

data Query a =
  Increment a |
  Decrement a

data Message = Updated Int
```

Notice we have an extra parameter on our query type. This represents the "next" action that will happen in our UI. We'll see how this works when we write our eval function. But first, let's write out our render function. It has three different HTML elements: two buttons and a p label. We'll stick them in a div element.

```
render :: State -> H.ComponentHTML Query
render state =
  let incButton = HH.button
                    [ HP.title "Inc"
                    , HE.onClick (HE.input_ Increment)
                    ]
                    [ HH.text "Inc" ]
      decButton = HH.button
                    [ HP.title "Dec"
                    , HE.onClick (HE.input_ Decrement)
                    ]
                    [ HH.text "Dec" ]
      pElement  = HH.p [] [HH.text (show state)]
```

```
  in  HH.div [] [incButton, decButton, pElement]
```

Each of our elements takes two list parameters. The first list includes properties as well as event handlers. Notice our buttons send query messages on their click events using the input_ function. Then the second list is "child" HTML elements, including the inner text of a button.

Now, to write our eval function, we use a case statement. This might seem a little weird, but all we're doing is breaking it down into our query cases:

```
eval :: Query ~> H.ComponentDSL State Query Message m
eval = case _ of
      Increment next -> ...
      Decrement next -> ...
```

Within each case, we can use State monad-like functions to manipulate our state. Our cases are identical except for the sign. We'll also use the raise function to send an update message. Nothing listens for that message right now, but it illustrates the concept.

```
eval :: Query ~> H.ComponentDSL State Query Message m
eval = case _ of
      Increment next -> do
        state <- H.get
        let nextState = state + 1
        H.put nextState
        H.raise $ Updated nextState
        pure next
      Decrement next -> do
        state <- H.get
        let nextState = state - 1
        H.put nextState
        H.raise $ Updated nextState
        pure next
```

As a last note, we would use const 0 as the initialState in our component function.

INSTALLING OUR COMPONENT

Now to display this component in our UI, we write a short Main module like so. We get our body element with awaitBody and then use runUI to install our counter component.

```
module Main where

import Prelude
import Effect (Effect)
import Halogen.Aff as HA
import Halogen.VDom.Driver (runUI)
import Counter (counter)

main :: Effect Unit
main = HA.runHalogenAff do
  body <- HA.awaitBody
  runUI counter unit body
```

And our counter component will now work! (See Github for more details on you could run this code).

BUILDING OUR TODO LIST

Now that we've got the basics down, let's see how to write a more complicated set of components. We'll write a Todo list like we had in the Elm series. To start, let's make a Todo wrapper type and derive some instances for it:

```
newtype Todo = Todo
  { todoName :: String }

derive instance eqTodo :: Eq Todo
derive instance ordTodo :: Ord Todo
```

Our first component will be the entry form, where the user can add a new task. This form will use the text input string as its state. It will respond to queries for updating the name as well as pressing the "Add" button. When we create a new Todo, we'll send a message for that.

```
type AddTodoFormState = String


data AddTodoFormMessage = NewTodo Todo


data AddTodoFormQuery a =
  AddedTodo a |
  UpdatedName String a
```

When we render this component, we'll have two main pieces. First, we need the text field to input the name. Then, there's the button to add the task. Each of these has an event attached to it sending the relevant query. In the case of updating the name, notice we use input instead of input_. This allows us to send the text field's value as an argument of the UpdatedName query. Otherwise, the properties are pretty straightforward translations of HTML properties you might see.

```
render ::
  AddTodoFormState ->
  H.ComponentHTML AddTodoFormQuery
render currentName =
  let nameInput = HH.input
                    [ HP.type_ HP.InputText
                    , HP.placeholder "Task Name"
                    , HP.value currentName
                    , HE.onValueChange (HE.input UpdatedName)
                    ]
      addButton = HH.button
                    [ HP.title "Add Task"
                    , HP.disabled (length currentName == 0)
                    , HE.onClick (HE.input_ AddedTodo)
                    ]
                    [ HH.text "Add Task" ]
  in  HH.div [] [nameInput, addButton]
```

Evaluating our queries is pretty simple. When updating the name, all we do is update the state and trigger the next action. When we add a new Todo item, we save the empty string as the state and

raise our message. In the next part, we'll see how our list will respond to this message.

```
eval ::
  AddTodoFormQuery ~>
  H.ComponentDSL
    AddTodoFormState AddTodoFormQuery AddTodoFormMessage m
eval = case _ of
  AddedTodo next -> do
    currentName <- H.get
    H.put ""
    H.raise $ NewTodo (Todo {todoName: currentName})
    pure next
  UpdatedName newName next -> do
    H.put newName
    pure next
```

And of course, we tie this all up by using the component function:

```
addTodoForm :: forall m.
  H.Component HH.HTML AddTodoFormQuery Unit AddTodoFormMessage m
addTodoForm = H.component
  { initialState: const ""
  , render
  , eval
  , receiver: const Nothing
  }
```

FINISHING THE LIST

Now to complete our todo list, we'll need another component to store the tasks themselves. As always, let's start with our basic types. We won't bother with a message type since this component won't send any messages. We'll use Void when assigning the message type in a type signature:

```
type TodoListState = Array Todo

data TodoListQuery a =
  FinishedTodo Todo a |
```

```
  HandleNewTask AddTodoFormMessage a
```

Our state is our list of tasks. Our query type is a little more complicated. The HandleNewTask query will receive the new task messages from our form. We'll see how we make this connection below.

We'll also add a type alias for AddTodoFormSlot. Halogen uses a "slot ID" to distinguish between child elements. We only have one child element though, so we'll use a string.

```
type AddTodoFormSlot = String
```

We'll consider this component a "parent" of our "add task" form. This means the types will look a little different. We'll be making something of type ParentHTML. The type signature will include references to its own query type, the query type of its child, and the slot ID type. We'll still use most of the same functions though.

```
render ::
  TodoListState ->
  H.ParentHTML TodoListQuery AddTodoFormQuery AddTodoFormSlot m

eval ::
  TodoListQuery ~>
  H.ParentDSL TodoListState TodoListQuery AddTodoFormQuery
    AddTodoFormSlot Void m
```

To render our elements, we'll have two sub-components. First, we'll want to be able to render an individual Todo within our list. We'll give it a p label for the name and a button that completes the task:

```
renderTask ::
  Todo ->
  H.ParentHTML TodoListQuery AddTodoFormQuery AddTodoFormSlot m
renderTask (Todo t) = HH.div_
  [ HH.p [] [HH.text t.todoName]
  , HH.button
      [ HE.onClick (HE.input_ (FinishedTodo (Todo t)))]
      [HH.text "Finish"]
  ]
```

Now we need some HTML for the form slot itself. This is straightforward. We'll use the slot function and provide a string for the ID. We'll specify the component we have from the last part. Then we'll attach the HandleNewTask query to this component. The allows our list component to receive the new-task messages from the form.

```
formSlot ::
  H.ParentHTML TodoListQuery AddTodoFormQuery AddTodoFormSlot m
formSlot = HH.slot
  "Add Todo Form"
  addTodoForm
  unit
  (HE.input HandleNewTask)
```

Now we combine these elements in our render function:

```
render ::
  TodoListState ->
  H.ParentHTML TodoListQuery AddTodoFormQuery AddTodoFormSlot m
render todos =
  let taskList = HH.ul_ (map renderTask todos)
  in  HH.div_ [taskList, formSlot]
```

Writing our eval is now a simple matter of using a few array functions to update the list. When we get a new task, we add it to our list. When we finish a task, we remove it from the list.

```
eval ::
  TodoListQuery ~>
  H.ParentDSL TodoListState TodoListQuery AddTodoFormQuery
    AddTodoFormSlot Void m
eval = case _ of
  FinishedTodo todo next -> do
    currentTasks <- H.get
    H.put (filter  (_ /= todo) currentTasks)
    pure next
  HandleNewTask (NewTodo todo) next -> do
    currentTasks <- H.get
    H.put (currentTasks `snoc` todo)
    pure next
```

And that's it! We're done! Again, take a look at the Github repo for some more instructions on how you can run and interact with this code.

CONCLUSION

This wraps up our look at building simple UI's with Purescript. In part 4, we'll conclude our Purescript series. We'll look at some of the broader elements of building a web app. We'll see some basic routing as well as how to send requests to a backend server.

Elm is another great functional language you can use for Web UIs. To learn more about it, check out our Elm Series!

Revision #1
Created 2022-03-11 16:55:26 UTC by gasick
Updated 2022-03-11 17:11:16 UTC by gasick