

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

# Purescript: Haskell + Javascript

Elm is a great language, as we cover in our Elm Series. It has great, intuitive primitives for building a web UI. But it lacks a lot of important features that we as Haskell developers are used to, most notably typeclasses.

Purescript offers another alternative in the realm of functional "javascript-like" languages. Its feature set more closely resembles that of Haskell. In this series we'll explore Purescript, from the absolute basics of the language to building full web UI's.

- [Purescript Part 1: Basics of Purescript](#)
- [Purescript Part 2: Typeclasses and Monads](#)
- [Purescript Part 3: Simple Web UI's](#)
- [Purescript Part 4: Web Requests and Navigation](#)

# Purescript Part 1: Basics of Purescript

Our Haskell Web Series covers a lot of cool libraries you can use when making a web app. But frontend web development can be quite a different story! There are a number libraries and frameworks out there. Yesod and Snap come to mind. Another option is Reflex FRP, which uses GHCJS under the hood.

But in this series we'll take different approach by exploring the Purescript language. Purescript is a bit of a meld between Haskell and Javascript. Its syntax is like Haskell's, and it incorporates many elements of functional purity. But it compiles to Javascript and thus has some features that seem more at home in that language.

In this part, we'll start out by exploring the basics of Purescript. If you're already familiar with those, you can move right onto part 2 of the series! There, we'll see some of the main similarities and differences between it and Haskell. We'll culminate this series by making a web front-end with Purescript and routing between different pages.

Purescript is the tip of the iceberg when it comes to using functional languages in production! Check out our Production Checklist for some awesome Haskell libraries!

## GETTING STARTED

Since Purescript is its own language, we'll need some new tools. You can follow the instructions on the Purescript website, but here are the main points.

1. Install Node.js and NPM, the Node.js package manager
2. Run `npm install -g purescript`
3. Run `npm install -g pulp bower`
4. Create your project directory and run `pulp init`.

5. You can then build and test code with `pulp build` and `pulp test`.
6. You can also use PSCI as a console, similar to GHCi. First, we need NPM. Purescript is its own language, but we want to compile it to Javascript we can use in the browser, so we need Node.js. Then we'll globally install the Purescript libraries. We'll also install `pulp` and `bower`. `Pulp` will be our build tool like `Cabal`.

`Bower` is a package repository like `Hackage`. To get extra libraries into our program, you would use the `bower` command. For instance, we need `purescript-integers` for our solution later in the article. To get this, run the command:

```
bower install --save purescript-integers
```

## A SIMPLE EXAMPLE

Once you're set up, it's time to start dabbling with the language. While Purescript compiles to Javascript, the language itself actually looks a lot more like Haskell! We'll examine this by comparison. Suppose we want to find the all pythagorean triples whose sum is less than 100. Here's how we can write this solution in Haskell:

```
sourceList :: [Int]
sourceList = [1..100]

allTriples :: [(Int, Int, Int)]
allTriples =
  [(a, b, c) | a <- sourceList, b <- sourceList, c <- sourceList]

isPythagorean :: (Int, Int, Int) -> Bool
isPythagorean (a, b, c) = a ^ 2 + b ^ 2 == c ^ 2

isSmallEnough :: (Int, Int, Int) -> Bool
isSmallEnough (a, b, c) = a + b + c < 100

finalAnswer :: [(Int, Int, Int)]
finalAnswer = filter
  (\t -> isPythagorean t && isSmallEnough t)
  allTriples
```

Let's make a module in Purescript that will allow us to solve this same problem. We'll start by writing a module `Pythagoras.purs`. Here's the code we would write to match up with the Haskell above. We'll examine the specifics piece-by-piece below.

```
module Pythagoras where

import Data.List (List, range, filter)
import Data.Int (pow)
import Prelude

sourceList :: List Int
sourceList = range 1 100

data Triple = Triple
  { a :: Int
  , b :: Int
  , c :: Int
  }

allTriples :: List Triple
allTriples = do
  a <- sourceList
  b <- sourceList
  c <- sourceList
  pure $ Triple {a: a, b: b, c: c}

isPythagorean :: Triple -> Boolean
isPythagorean (Triple triple) =
  (pow triple.a 2) + (pow triple.b 2) == (pow triple.c 2)

isSmallEnough :: Triple -> Boolean
isSmallEnough (Triple triple) =
  (triple.a) + (triple.b) + (triple.c) < 100

finalAnswer :: List Triple
finalAnswer = filter
  (\triple -> isPythagorean triple && isSmallEnough triple)
  allTriples
```

For the most part, things are very similar! We still have expressions. These expressions have type signatures. We use a lot of similar elements like lists and filters. On the whole, Purescript looks a lot more like Haskell than Javascript. But there are some key differences. Let's explore those, starting with the higher level concepts.

# DIFFERENCES

One difference you can't see in code syntax is that Purescript is NOT lazily evaluated. Javascript is an eager language by nature. So it is much easier to compile to JS by starting with an eager language in the first place.

But now let's consider some of the differences we can see from the code. For starters, we have to import more things. Purescript does not import a Prelude by default. You must always explicitly bring it in. We also need imports for basic list functionality.

And speaking of lists, Purescript lacks a lot of the syntactic sugar Haskell has. For instance, we need to use `List Int` rather than `[Int]`. We can't use `..` to create a range, but instead resort to the `range` function.

We also cannot use list comprehensions. Instead, to generate our original list of triples, we use the list monad. As with lists, we have to use the term `Unit` instead of `()`:

```
-- Comparable to main :: IO ()
main :: Effect Unit
main = do
  log "Hello World!"
```

In the next part, we'll discuss the distinction between `Effect` in Purescript and monadic constructs like `IO` in Haskell.

One annoyance is that polymorphic type signatures are more complicated. Whereas in Haskell, we have no issue creating a type signature `[a] -> Int`, this will fail in Purescript. Instead, we must always use the `forall` keyword:

```
myListFunction :: forall a. List a -> Int
```

Another thing that doesn't come up in this example is the Number type. We can use Int in Purescript as in Haskell. But aside from that the only important numeric type is Number. This type can also represent floating point values. Both of these get translated into the number type in Javascript.

# PURESRIPT DATA TYPES

But now let's get into one of the more glaring differences between our examples. In Purescript, we need to make a separate Triple type, rather than using a simple 3-tuple. Let's look at the reasons for this by considering data types in general.

If we want, we can make Purescript data types in the same way we would in Haskell. So we could make a data type to represent a Pythagorean triple:

```
data Triple = Triple a b c
```

This works fine in Purescript. But, it forces us to use pattern matching every time we want to pull an individual value out of this element. We can fix this in Haskell by using record syntax to give ourselves accessor functions:

```
data Triple = Triple
  { a :: Int
  , b :: Int
  , c :: Int
  }
```

This syntax still works in Purescript, but it means something different. In Purescript a record is its own type, like a generic Javascript object. For instance, we could do this as a type synonym and not a full data type:

```
type Triple = { a :: Int, b :: Int, c :: Int}

oneTriple :: Triple
oneTriple = { a: 5, b: 12, c: 13}
```

Then, instead of using the field names like functions, we use "dot-syntax" like in Javascript. Here's what that looks like with our type synonym definition:

```
type Triple = { a :: Int, b :: Int, c :: Int}

oneTriple :: Triple
oneTriple = { a: 5, b: 12, c: 13}

sumAB :: Triple -> Int
sumAB triple = triple.a + triple.b
```

Here's where it gets confusing though. If we use a full data type with record syntax, Purescript no longer treats this as an item with 3 fields. Instead, we would have a data type that has one field, and that field is a record. So we would need to unwrap the record using pattern matching before using the accessor functions.

```
data Triple = Triple
  { a :: Int
  , b :: Int
  , c :: Int
  }

oneTriple :: Triple
oneTriple = Triple { a: 5, b: 12, c: 13}

sumAB :: Triple -> Int
sumAB (Triple triple) = triple.a + triple.b

-- This is wrong!
sumAB :: Triple -> Int
sumAB triple = triple.a + triple.b
```

That's a pretty major gotcha. The compiler error you get from making this mistake is a bit confusing, so be careful!

# PYTHAGORAS IN PURESCRIPT

With this understanding, the Purescript code above should make some more sense. But we'll go through it one more time and point out the little details.

To start out, let's make our source list. We don't have the range syntactic sugar, but we can still use the range function:

```
import Data.List (List, range, filter)

data Triple = Triple
  { a :: Int
  , b :: Int
  , c :: Int
  }

sourceList :: List Int
sourceList = range 1 100
```

We don't have list comprehensions. But we can instead use do-syntax with lists instead to get the same effect. Note that to use do-syntax in Purescript we have to import Prelude. In particular, we need the bind function for that to work. So let's generate all the possible triples now.

```
import Prelude

...

allTriples :: List Triple
allTriples = do
  a <- sourceList
  b <- sourceList
  c <- sourceList
  pure $ Triple {a: a, b: b, c: c}
```



Notice also we use `pure` instead of `return`. Now let's write our filtering functions. These will use the record pattern matching and accessing mentioned above.

```
isPythagorean :: Triple -> Boolean
isPythagorean (Triple triple) =
    (pow triple.a 2) + (pow triple.b 2) == (pow triple.c 2)

isSmallEnough :: Triple -> Boolean
isSmallEnough (Triple triple) =
    (triple.a) + (triple.b) + (triple.c) < 100
Finally, we can combine it all with filter in much the same way we did in Haskell:
```

```
finalAnswer :: List Triple
finalAnswer = filter
    (\triple -> isPythagorean triple && isSmallEnough triple)
    allTriples
```

And now our solution will work!

# CONCLUSION

This concludes part 1 of our Purescript series. Syntactically, Purescript is a very near cousin of Haskell. But there are a few key differences we highlighted here about the nature of the language.

In part 2, we'll look at some other important differences in the type system. We'll see how Purescript handles type-classes and monads. After that, we'll see how we can use Purescript to build a web front-end with some of the security of a solid type system.

Download our Production Checklist for some more cool ideas of libraries you can use!

# Purescript Part 2:

## Typeclasses and Monads

In part 1 of this series, we started our exploration of Purescript. Purescript seeks to bring some of the awesomeness of Haskell to the world of web development. Its syntax looks a lot like Haskell's, but it compiles to Javascript. This makes it very easy to use for web applications. And it doesn't just look like Haskell. It uses many of the important features of the language, such as a strong system and functional purity.

If you need to brush up on the basics of Purescript, make sure to check out part 1 again. In this part, we're going to explore a couple other areas where Purescript is a little different. We'll see how Purescript handles typeclasses, and we'll also look at monadic code. We'll also take a quick look at some other small details with operators. In part 3, we'll look at how we can use Purescript to write some front-end code.

For another perspective on functional web development, check out our Haskell Web Series. You can also download our Production Checklist for some more ideas!

## TYPE CLASSES

The idea of type classes remains pretty consistent from Haskell to Purescript. But there are still a few gotchas. Let's remember our Triple type from the last part.

```
data Triple = Triple
  { a :: Int
  , b :: Int
  , c :: Int
  }
```

Let's write a simple Eq instance for it. To start with, instances in Purescript must have names. So we'll assign the name tripleEq to our instance:

```
instance tripleEq :: Eq Triple where
  eq (Triple t1) (Triple t2) = t1 == t2
```

Once again, we only unwrap the one field for our type. This corresponds to the record, rather than the individual fields. We can, in fact, compare the records with each other. The name we provide helps Purescript to generate Javascript that is more readable. Take note: naming our instances does NOT allow us to have multiple instances for the same type and class. We'll get a compile error if we try to create another instance like:

```
instance otherTripleEq :: Eq Triple where
  ...
```

There's another small change when using an explicit import for classes. We have to use the class keyword in the import list:

```
import Data.Eq (class Eq)
```

You might hope we could derive the Eq typeclass for our Triple type, and we can. Since our instance needs a name though, the normal Haskell syntax doesn't work. The following will fail:

```
-- DOES NOT WORK
data Triple = Triple
  { a :: Int
  , b :: Int
  , c :: Int
  } deriving (Eq)
```

For simple typeclasses though, we CAN use standalone deriving. This allows us to provide a name to the instance:

```
derive instance eqTriple :: Eq Triple
```

As a last note, Purescript does not allow orphan instances. An orphan instance is where you define a typeclass instance in a different file from both the type definition and the class definition. You can get away with these in Haskell, though GHC will warn you about it. But Purescript is less forgiving. The way to work around this issue is to define a newtype wrapper around your type. Then you can define the instance on that wrapper.

# EFFECTS

In part 1, we looked at a small snippet of monadic code. It looked like:

```
main :: Effect Unit
main = do
  log ("The answer is " <> show answer)
```

If we're trying to draw a comparison to Haskell, it seems as though Effect is a comparable monad to IO. And it sort've is. But it's a little more complicated than that. In Purescript, we can use Effect to represent "native" effects. Before we get into exactly what this means and how we do it, let's first consider "non-native" effects.

A non-native effect is one of those monads like Maybe or List that can stand on its own. In fact, we have an example of the List monad in part 1 of this series. Here's what Maybe might look like.

```
maybeFunc :: Int -> Maybe Int

mightFail :: Int -> Maybe Int
mightFail x = do
  y <- maybeFunc x
  z <- maybeFunc y
  maybeFunc z
```

Native effects use the Effect monad. These include a lot of things we'd traditionally associate with IO in Haskell. For instance, random number generation and console output use the Effect monad:

```
randomInt :: Int -> Int -> Effect Int

log :: String -> Effect Unit
```

But there are also other "native effects" related to web development. The most important of these is anything that writes to the DOM in our Javascript application. In the next part, we'll use the Halogen library to create a basic web page. Most of its main functions are in the Effect monad. Again, we can imagine that this kind of effect would use IO in Haskell. So if you want to think of Purescript's Effect as an analogue for IO, that's a decent starting point.

What's interesting is that Purescript used to be more based on the system of free monads. Each different type of native effect would build on top of previous effects. The cool part about this is the way Purescript uses its own record syntax to track the effects in play. You can read more about how this can work in chapter 8 of the Purescript Book. However, we won't need it for our examples. We can just stick with Effect.

Besides free monads, Purescript also has the `purescript-transformers` library. If you're more familiar with Haskell, this might be a better starting spot. It allows you to use the MTL style approach that's more common in Haskell than free monads.

# SPECIAL OPERATORS

It's worth noting a couple other small differences. Some rules about operators are a little different between Haskell and Purescript. Since Purescript uses the period operator `.` for record access, it no longer refers to function composition. Instead, we would use the `<<<` operator:

```
odds :: List Int -> List Int
odds myList = filter (not <<< isEven) myList
  where
    isEven :: Int -> Boolean
    isEven x = mod x 2 == 0
```

Also, we cannot define operators in an infix way. We must first define a normal name for them. The following will NOT work:

```
(=%=) :: Int -> Int -> Int
(=%=) a b = 2 * a - b
```

Instead, we need to define a name like `addTwiceAndSubtract`. Then we can tell Purescript to apply it as an infix operator:

```
addTwiceAndSubtract :: Int -> Int -> Int
addTwiceAndSubtract a b = 2 * a - b

infixl 6 addTwiceAndSubtract as %=
```

Finally, using operators as partial functions looks a little different. This works in Haskell but not Purescript:

```
doubleAll :: List Int -> List Int
doubleAll myList = map (* 2) myList
```

Instead, we want syntax like this:

```
doubleAll :: List Int -> List Int
doubleAll myList = map (_ * 2) myList
```

# CONCLUSION

This wraps up our look at the key differences between Haskell and Purescript. Now that we understand typeclasses and monads, it's time to dive into what Purescript is best at. In part 3 we'll look at how we can write real frontend code with Purescript!

For some more ideas on using Haskell for some cool functionality, download our [Production Checklist](#)! For another look at function frontend development, check out our recent [Elm Series](#)!

# 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 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. This 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!



# 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.onChange (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 our 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.