PureScript-Milkis Documentation

Justin Woo

May 14, 2022

Contents

This is a guide for the PureScript library Milkis, a library for easily making HTTP requests by using the Fetch API and getting the results in Aff.

This library allows you to use fetch from the Browser by using the `window` implementation, and from Node by using the node-fetch library.

Note: If there is a topic you would like more help with that is not in this guide, open a issue in the Github repo for it to request it.

CHAPTER 1

Pages

1.1 Introduction

1.1.1 FetchImpl

To use this library, you'll have to use a value of FetchImpl, which is a foreign data type. This library provides two bindings via the modules Milkis.Impl.Window and Milkis.Impl.Node. You may choose to bring your own, typing a foreign import as FetchImpl.

You can partially apply the function Milkis.fetch to get a value of Fetch. For example, with Node you could do this:

```
import Milkis as M
import Milkis.Impl.Node (nodeFetch)

fetch :: M.Fetch
fetch = M.fetch nodeFetch
```

1.1.2 Fetch

Fetch is simply a type alias:

```
type Fetch
   = forall options trash
   . Union options trash Options
  => URL
  -> Record (method :: Method | options)
  -> Aff Response
```

What this signature says is given some type varaibles options and trash where there is a Union of options and trash together to form Options, we have a function that takes URL and a record with a method :: Method field and the fields specified in options to return an Aff Response. Let's look at the definition of Options:

```
type Options =
  ( method :: Method
  , body :: String
  , headers :: Headers
  , credentials :: Credentials
  )
```

So what the `Union` constraint does here is declare that `options` must have some subset of this row type, and that there exists some `trash` row type that is the complement. *For more reading about `Union`, you might want to read a post about it here: https://github.com/justinwoo/my-blog-posts#unions-for-partial-properties-in-purescript*

1.1.3 How using `Fetch` works

Let's see an example of this at work:

```
main = do
  _response <- Aff.attempt $ fetch (M.URL "https://www.google.com") M.
↪defaultFetchOptions
  case _response of
    Left e -> do
      fail $ "failed with " <> show e
    Right response -> do
      stuff <- M.text response
      let code = M.statusCode response
      code `shouldEqual` 200
      String.null stuff `shouldEqual` false
```

Let's also peek at the definition of `defaultFetchOptions`:

```
defaultFetchOptions :: { method :: Method }
defaultFetchOptions =
  { method: getMethod
  }
```

So in this case, we chose to only supply `method :: Method` and it worked. Let's see how this works with a POST request:

```
main = do
  let
    opts =
      { method: M.postMethod
      , body: "{}"
      , headers: M.makeHeaders { "Content-Type": "application/json" }
      }
  result <- attempt $ fetch (M.URL "https://www.google.com") opts
  isRight result `shouldEqual` true
```

This time, we provided a body for the post method along with some headers. If we look at the type of `makeHeaders`, we can get a better idea of what is happening:

```
makeHeaders
  :: forall r . Homogeneous r String
  => Record r
  -> Headers
```

Here, `makeHeaders` allows us to create headers from a homogeneous record of `String` types using the `Homogeneous` class from Typelevel Prelude.

By using these constraints, we are able to use `Fetch` in quite flexible ways that don't require having a large set of default options to be overridden. If you understand the content of this page, you'll be able to tackle any problems you run into with this library and understand how to use `Union`-based approaches in general.

For the browser usage, you should really only need to do `fetch = M.fetch windowFetch` and be on your way, so please look through the tests for examples of how to use this library.