WebApi Documentation

Release 0.1

ByteAlly

Jul 25, 2018

Contents:

1 Installation **3**

2 Quick start **5**
- 2.1 Contract 5
- 2.2 Implementation 6

3 Routing **9**
- 3.1 Routes as types 9
- 3.2 More examples 10

4 Implementation **11**
- 4.1 Writing WebApiImplementation instance 11
- 4.2 Writing instances for your handlers 12
- 4.3 Doing more with your handler monad 12

5 Content Serialization / Deserealization **15**
- 5.1 Nested Types 15
- 5.2 Writing Custom instances 16
- 5.3 Content Types 16

6 Error Handling **19**

7 Building haskell client for third-party API **21**

8 Mocking Data **25**

WebApi is a Haskell library that lets you

- Write web API services
- Quickly build Haskell client for existing API services
- Generate API console interface for your web API (coming soon)
- Generate a mock server that can mock your responses and requests

WebApi is built with WAI. It makes use of the strong type system of haskell which lets to

- Create a type safe routing system.
- Enable type safe generation of links.
- Specify a contract for the APIs.
- Auto serialization and deserialization of the request and response based on api contract.
- Write handlers which respect the contract.

CHAPTER 1

Installation

We recommend using stack build tool for installation and building. If you don't have stack already, follow these instructions to install it. To setup your own project:

1) Create a project using stack:

```
stack new <Your-Project-Name>
```

2) Then add `webapi` to the `extra-deps` section in **stack.yaml** file:

```
extra-deps:
- webapi-0.2.2.0
```

3) Finally add `webapi` to the `build-depends` section of your **cabal** file.

```
build-depends:    webapi
```

CHAPTER 2

Quick start

Writing your API service comprises of two steps

- Writing a contract (definition below)
- Providing an implementation

2.1 Contract

A contract is the list of end-points in your API service and the definition of each API endpoint. We define what goes in as **request** (Query params, form params, headers etc) and what comes out as the **response** of each API endpoint.

As an example, consider a API service that lets you create, update, delete and fetch users. First step is to create a datatype for our API service. Lets call it `MyApiService`

To define your contract using the framework, you need to

- Declare a data type for your API service.

```
data MyApiService
```

- Declare your routes as types.

```
type User   = Static "user"
type UserId = "user":/Int
```

- Write a WebApi instance which declares the endpoints.

```
instance WebApi MyApiService where
  -- Route <Method  <Route Name
   type Apis MyApiService = '[ Route '[GET, POST]        User
                            , Route '[GET, PUT, DELETE] UserId
                             ]
```

- Write ApiContract instances describing what goes in an **request** and what comes out as **response** from each API endpoint. Let's write our first ApiContract instance for `POST /user`.

```
-- Our user type
data UserData = UserData { age     :: Int
                         , address :: Text
                         , name    :: Text
                         } deriving (Show, Eq, Generic)

data UserToken = UserToken { userId :: Text
                           , token :: Text
                           } deriving (Show, Eq, Generic)


-- Takes a User type in form params and returns unit.
instance ApiContract MyApiService POST User where
  type FormParam POST User = UserData
  type ApiOut    POST User = UserToken
```

In our code snippet above, the end-point `POST /user` takes user's information (**name, age** and **address**) as **post params** and gives out the user's **token** and **userId**

An equivalent curl syntax would be:

```
`curl -H "Content-Type: application/x-www-form-urlencoded" -d 'age=30&
↪address=nazareth&name=Brian' http://api.peoplefrontofjudia.com/users `
```

- Finally to complete our contract, we have to write instances for json, param serialization & deserialization for `UserData` and `UserToken` types. A definition needn't be provided since GHC.Generics provides a generic implementation.

```
instance FromJSON UserData
instance ToJSON   UserData
instance FromParam UserData 'FormParam

{--We dont need a FromParam instance since UserToken according
 to our example is not sent us form params or query params -}
instance FromJSON UserToken
instance ToJSON   UserToken
```

This completes the contract part of the API.

2.2 Implementation

First step is to create a type for the implementation and define WebApiImplementation instance for it.

```
data MyApiServiceImpl = MyApiServiceImpl

instance WebApiImplementation MyApiServiceImpl where
  type HandlerM MyApiServiceImpl = IO
  type ApiInterface MyApiServiceImpl = MyApiService
```

HandlerM is the base monad in which the handler will run. We also state that `MyApiServiceImpl` is an implementation of the ApiInterface provided by `MyApiServiceApi`.

Now let's create the ApiHandler

```
instance ApiHandler MyApiServiceImpl POST User where
  handler _ req = do
    let _userInfo = formParam req
    respond ()
```

By keeping the implementation separate from the contract, it is possible for a contract to have multiple implementations. Hypothetically, there could be a **websocket** implementation as well as a **ReST** implementation for a single contract.

The last thing that is left is to create a WAI application from all the aforementioned information. For that we use serverApp .

```
myApiApp :: Wai.Application
myApiApp = serverApp serverSettings MyApiServiceImpl

main :: IO ()
main = run 8000 myApiApp
```

That's it - now `myApiApp` could be run like any other WAI application.

There's more you could do with **WebApi** apart from building API services. You can also build haskell clients for existing API services by defining just the contract, build full-stack webapps that serve html & javascript and generate mock servers.

CHAPTER 3

Routing

WebApi supports the following HTTP verbs **GET**, **POST**, **PUT**, **DELETE**, **PATCH**, **HEAD**

In WebApi we need to first write all the routes as types and then declare the valid HTTP verbs for each route type.

3.1 Routes as types

Each route is declared as a type. For demo purpose let's consider a API service that would allow you to create and get users. We need two URIs. One to create an user another one to get the user by her ID.

`/user` **URI to create a user**

```
type User = Static "user"
```

`/user/9` **URI to get a user**

```
type UserId = "user" :/ Int
```

- Note that `/user` is declared as `Static "user"` to wrap **user** in `Static` to make all the types of the same **kind (*)**

As you could see in the above examples, routes are defined as types. The next step is to write a WebApi instance for the route types along with the HTTP verbs they support.

```
instance WebApi SocialNetworkApi where
  -- Route <Method  <Route Name
   type Apis SocialNetworkApi = '[ Route '[POST]        User
                              , Route '[GET, PUT, DELETE] UserId
                               ]
```

In the above code snippet, we are delcaring that our route type

- `User` ie (`/user`) accepts `POST`
- `UserId` accepts `GET`, `PUT`, `DELETE`.

- Let's say the user Id is 9, `GET /user/9` to get the user, `PUT /user/9` to edit the user and `DELETE user/9` to delete the user. You could read about implementation instances under the section *Implementation*

3.2 More examples

`/post/tech/why-i-like-web-api`

```
type Post = "post" :/ Text :/ Text
```

`/post/tech/why-i-like-web-api/edit`

```
type EditPost = "post" :/ Text :/ Text :/ "edit"
```

`/why-i-like-web-api/comments`

```
type Comments = Text :/ "comments"
```

- Please note that when two route format overlaps, for example `user/posts` and `user/brian` WebApi's routing system would take the first route that matches the pattern.

CHAPTER 4

Implementation

An ApiContract is just a schematic representation of your API service. We still need to implement our handlers that actually does the work. You would have already read about this in the *Quick start* section.

Implementation of a contract consists of

- Writing a `WebApiImplementation` instance.
- Writing `ApiHandler` instances for all your end-points.

4.1 Writing WebApiImplementation instance

The `WebApiImplementation` typeclass has

- **Two associated types**
  - **HandlerM** - It is the type of monad in which our handler should run (defaults to `IO`). This monad should implement `MonadCatch` and `MonadIO` classes.
  - **ApiInterface** - `ApiInterface` links the implementation with the contract. This lets us have multiple implementations for the same contract
- **One method**
  - **toIO** - It is a method which is used to convert our handler monad's action to `IO`. (defaults to `id`)

Let's define a type for our implementation and write a `WebApiImplementation` instance for the same.

```
data MyApiImpl = MyApiImpl

instance WebApiImplementation MyApiImpl where
    type HandlerM MyApiImpl     = IO
    type ApiInterface MyApiImpl = MyApp
    toIO _                      = id
```

```
Note: You can skip writing :code:`HandlerM`'s and :code:`toIO`'s definitions if
you want your :code:`HandlerM` to be :code:`IO`.
```

4.2 Writing instances for your handlers

Now we can write handler for our `User` route as

```
instance ApiHandler MyApiImpl POST User where
  handler _ req = do
    let _userInfo = formParam req
    respond (UserToken "Foo" "Bar")
```

`handler` returns a `Response`. Here we used `respond` to build a `Success` Response. You can use its counter-part `raise` as discussed in *Error Handling* to send `Failure` Response

4.3 Doing more with your handler monad

Though the above implementation can get you started, it falls short for many practical scenarios. We'll discuss some of them in the following sections.

4.3.1 Adding a config Reader

Most of the times our app would need some kind of initial setting which could come from a config file or some environment variables. To accomodate for that, we can change `MyApiImpl` to

```
data AppSettings = AppSettings

data MyApiImpl = MyApiImpl AppSettings
```

Just adding `AppSettings` to our `MyApiImpl` is useless unless our monad gives a way to access those settings. So we need a monad in which we can read these settings, anytime we require. A `ReaderT` transformer would fit perfectly for this scenario.

For those who are not familiar with `Reader` monad, it is a monad which gives you read only access to some data(say, settings) throughout a computation. You can access that data in your monad using `ask`. `ReaderT` is a monad transformer which adds capabilities of `Reader` monad on top of another monad. In our case, we'll add reading capabilities to `IO`. So the monad for our handler would look something like

```
newtype MyApiMonad a = MyApiMonad (ReaderT AppSettings IO a)
    deriving (Monad, MonadIO, MonadCatch)
```

Note: `HandlerM` is required to have `MonadIO` and `MonadCatch` instances. Thats why you see them in the `deriving` clause.

There is still one more piece left, before we can use this. We need to define `toIO` function to convert `MyApiMonad`'s actions to `IO`. We can use *runReaderT* to pass the initial `Reader`'s environment settings and execute the computation in the underlying monad(IO in this case).

```
toIO (MyApiImpl settings) (MyApiMonad r) = runReaderT r settings
```

So the `WebApiImplementation` instance for our modified `MyApiImpl` would look like:

```
instance WebApiImplementation MyApiImpl where
    type HandlerM MyApiImpl = MyApiMonad
    type ApiInterface MyApiImpl = MyApp
    toIO (MyApiImpl settings) (MyApiMonad r) = runReaderT r settings
```

A sample `ApiHandler` for this would be something like:

```
instance ApiHandler MyApiImpl POST User where
    handler _ req = do
        settings <- ask
        -- do something with settings
        return ()
```

4.3.2 Adding a logger

Adding a logging system to our implementation is similar to adding a `Reader`. We use `LoggingT` transformer to achieve that.

```
newtype MyApiMonad a = MyApiMonad (LoggingT (ReaderT AppSettings IO) a)
    deriving (Monad, MonadIO, MonadCatch, MonadLogger)

instance WebApiImplementation MyApiImpl where
    type HandlerM MyApiImpl = MyApiMonad
    type ApiInterface MyApiImpl = MyApp
    toIO (MyApiImpl settings) (MyApiMonad r) = runReaderT (runStdoutLoggingT r)␣
↪settings
```

CHAPTER 5

Content Serialization / Deserealization

In WebApi, `ToParam` and `FromParam` are the typeclasses responsible for serializing and deserializing data. Serialization and deserialization for your data types are automatically take care of if they have generic instances without you having to write anything. You still have to derive them though.

Lets look at an example

```
data LatLng = LatLng
   { lat :: Double
   , lng :: Double
   } deriving Generic
```

To let WebApi automatically deserialize this type, we just need to give an empty instance declaration

```
instance FromParam LatLng 'QueryParam
```

And to serialize a type (in case you are writing a client), you can give a similar ToParam instance.

```
instance ToParam LatLng 'QueryParam
```

5.1 Nested Types

Nested types are serialized with a dot notation.

```
data UserData = UserData
    { age      :: Int
    , address :: Text
    , name     :: Text
    , location :: LatLng
    } deriving (Show, Eq, Generic)
```

Here the location field would be serialized as `location.lat` and `location.lng`

5.2 Writing Custom instances

Sometimes you may want to serialize/deserialize the data to a custom format. You can easily do this by writing a custom instance of `ToParam` and `FromParam`. Lets declare a datatype and try to write `ToParam` and `FromParam` instances for those.

```
data Location = Location { loc :: LatLng } deriving Generic

data LatLng = LatLng
    { lat :: Double
    , lng :: Double
    } deriving Generic
```

Lets say we want to deserialize query parameter `loc=10,20` to `Location` where `10` and `20` are values of `lat` and `lng` respectively. We can write a `FromParam` instance for this as follows:

```
instance FromParam Location 'QueryParam where
    fromParam pt key trie = case lookupParam pt key trie of
        Just par -> case splitOnComma par of
            Just (lt, lg) -> LatLng <$> decodeParam lt <*> decodeParam lg
            Nothing       -> Validation $ Left [ParseErr key "Unable to cast to LatLng
↪"]
        _ -> Validation $ Left [NotFound key]
  where
    splitOnComma :: ByteString -> (ByteString, ByteString)
    splitOnComma x =
      let (a, b) = break (== ',') x
      in if (BS.null a) || (BS.null b) then Nothing else Just (a, b)
```

`fromParam` takes a `Proxy` of our type (here, `Location`), a key (`ByteString`) and a `Trie`. `WebApi` uses `Trie` to store the parsed data while deserialization. `fromParam` returns a value of type `Validation` which is a wrapper over `Either` type carrying the parsed result.

We use `lookupParam` function for looking up the key (`loc`). If the key matches, it'll return `Just` with the value of the key (in our case `10,20`). Now we split this value into a tuple using `splitOnComma` and make a value of type `LatLng` using these.

Similarly, a `ToParam` instance for `Location` can be written as:

```
instance ToParam Location 'QueryParam where
  toParam pt pfx (Location (LatLng lt lg)) = [("loc", Just $ encodeParam lt <> "," <>␣
↪encodeParam lg)]
```

Here we take a value of type `Location` and convert it into a key-value pair. `WebApi` uses this key-value pair to form the query string.

This example only included `QueryParam` but this can be easily extended to other param types.

5.3 Content Types

You can tell WebApi about the content-type of `ApiOut/ApiErr` using `ContentTypes`.

```
instance ApiContract MyApiService POST User where
    type FormParam    POST User = UserData
    type ApiOut       POST User = UserToken
    type ContentTypes POST User = '[JSON]
```

By default `ContentTypes` is set to `JSON`. That means you need to give `ToJSON` instances for the types associated with `ApiOut/ApiErr` while writing server side component and `FromJSON` instances while writing client side version.

Apart from `JSON` you can give other types such as `HTML`, `PlainText` etc. You can see a complete list here

CHAPTER 6

Error Handling

WebApi gives you a way to raise errors in your handler using `raise`. The following handler is an example that raises a 404 error

```
instance ApiHandler MyApiImpl GET User where
  handler _ req = do
    hasUser <- isUserInDB
    if (hasUser)
        then respond (UserToken "Foo" "Bar")
        else raise status404 ()
```

`raise` takes two arguments. First one is the status code which we need to send with the `Response`. Second argument is of type `ApiErr m r` which defaults to Unit `()`.

If you want to send some additional information with your error response, you can write a data type for error and specify that as `ApiErr` in your contract.

An example,

```
data Error = Error { error :: Text } deriving (Show, Generic)
instance ToJSON Error
instance ParamErrToApiErr Error where
    toApiErr errs = Error (toApiErr errs)

instance ApiContract MyApiService POST User where
 type FormParam POST User = UserData
 type ApiOut    POST User = UserToken
 type ApiErr    POST User = Error
```

Any type which you associate with `ApiErr`, should have a `ParamErrToApiErr` instance. This is needed for `WebApi` to map all the failures to this type. Also based on `ContentType` set in the contract (which defaults to `JSON`), we need to give the required instance. In this case it is `ToJSON`.

CHAPTER 7

Building haskell client for third-party API

WebApi framework could be used to build haskell clients for existing API services. All you have to do is

- Define the routes (as types)
- Write the **contract** for the API service.

To demonstrate, we've chosen Uber API as the third party API service and picked the two most commonly used endpoints in Uber API

- get time estimate - Gets the time estimate for nearby rides
- request a ride - Lets us request a ride.

Since we have already discussed what a **contract** is under the *Quick start* section in detail we can jump straight to our example.

Lets first define the type for the API service, call it `UberApi` and types for our routes. (get time estimate and request a ride).

```
data UberApi

-- pieces of a route are seperated using ':/'
type TimeEstimateR = "estimates" :/ "time"
-- If the route has only one piece, we use 'Static' constructor to build it.
type RequestRideR  = Static "requests"
```

Now lets define what methods (GET, POST etc.) can be used on these routes. For this we need to define WebApi instance for our service `UberApi` .

```
instance WebApi UberApi where
    type Apis UberApi =
        '[ Route '[GET] TimeEstimateR
         , Route '[POST] RequestRideR
         ]
```

So far, we have defined the routes and the methods associated with them. We are yet to define how the requests and responses will look for these two end-points (**contract**).

We'll start with the TimeEstimateR route. As defined in the Uber API doc , GET request for TimeEstimateR takes the user's current latitude, longitude, product_id (if any) as query parameters and return back a result containig a list of TimeEstimate (rides nearby along with time estimates). And this is how we represent the query and the response as data types.

```
-- query data type
data TimeParams = TimeParams
    { start_latitude   :: Double
    , start_longitude  :: Double
    , product_id       :: Maybe Text
    } deriving (Generic)

-- response data type
newtype Times = Times { times :: [TimeEstimate] }
   deriving (Show, Generic)

-- We prefix each field with 't_' to prevent name clashes.
-- It will be removed during deserialization
data TimeEstimate = TimeEstimate
   { t_product_id   :: Text
   , t_display_name :: Text
   , t_estimate     :: Int
   } deriving (Show, Generic)


instance ApiContract UberApi GET TimeEstimateR where
    type HeaderIn   GET TimeEstimateR = Token
    type QueryParam GET TimeEstimateR = TimeParams
    type ApiOut     GET TimeEstimateR = Times
```

As request to Uber API requires an Authorization header, we include that in our contract for each route. The data type Token used in the header is defined here

There is still one piece missing though. Serialization/ de-serialization of request/response data types. To do that, we need to give FromJSON instance for our response and ToParam instance for the query param datatype.

```
instance ToParam TimeParams 'QueryParam
instance FromJSON Times
instance FromJSON TimeEstimate where
    parseJSON = genericParseJSON defaultOptions { fieldLabelModifier = drop 2 }
```

Similarly we can write contract for the other routes too. You can find the full contract here .

And that's it! By simply defining a contract we have built a Haskell client for Uber API. The code below shows how to make the API calls.

```
-- To get the time estimates, we can write our main function as:
main :: IO ()
main = do
    manager <- newManager tlsManagerSettings
    let timeQuery = TimeParams 12.9760 80.2212 Nothing
        cSettings = ClientSettings "https://sandbox-api.uber.com/v1" manager
        auth'     = OAuthToken "<Your-Access-Token-here>"
        auth      = OAuth auth'

    times' <- client cSettings (Request () timeQuery () () auth () () :: WebApi.
↪Request GET TimeEstimateR)
```

We use client function to send the request. It takes ClientSettings and Request as input and gives us the Response . If you see the Request pattern synonym, we need to give it all the params, headers etc. to construct a Request . So for a particular route, the params which we declare in the contract are filled with the declared datatypes and the rest defaults to `()` **unit**

When the endpoint gives a response back, WebApi deserializes it into Response . Lets write a function to handle the response.

```
let responseHandler res fn = case res of
       Success _ res' _ _    -> fn res'
       Failure err           -> print "Request failed :("
```

We have successfully made a request and now can handle the response with `responseHandler`. If the previous request (to get time estimate) was succesfull, lets book the nearest ride with our second route.

```
  responseHandler times' $ \times -> do
      let rideId = getNearestRideId times
          reqQuery = defRideReqParams { product_id = Just rideId, start_place_id =
↪Just "work", end_place_d = Just "home" }
          ridereq   = client cSettings (Request () () () () auth' () reqQuery ::
↪WebApi.Request POST RequestRideR)
      rideInfo' <- ridereq
      responseHandler rideInfo' $ \rideInfo -> do
          putStrLn "You have successfully booked a ride. Yay!"
          putStrLn $ "Ride Status: " ++ unpack (status rideInfo)
  return ()
where
  getNearestRideId (Times xs) = t_product_id . head . sortBy (comparing t_estimate) $
↪xs
```

And that's it! We now have our haskell client. Using the same contract you can also generate a mock server

You can find the full uber client library for haskell here .

CHAPTER 8

Mocking Data

Writing a contract enables you to create a mock server or a client by just writing the `Arbitrary` instances for datatypes used in the contract.

Lets create a mock server for the contract mentioned in *Quick start* by writing arbitrary instances for our datatypes.

```
instance Arbitrary UserData where
    arbitrary = UserData <$> arbitrary
                         <*> arbitrary
                         <*> arbitrary

instance Arbitrary UserToken where
    arbitrary = UserToken <$> arbitrary
                          <*> arbitrary
```

Now we can create a `Wai.Application` for our mock server as

```
mockApp :: Wai.Application
mockApp = mockServer serverSettings (MockServer mockServerSettings)
```

`mockServer` takes `ServerSettings` and `MockServer` as arguments. `MockServer` lets you decide what kind of mock data is to be returned (`ApiOut`, `ApiError` or `OtherError`). It returns `ApiOut` (`SuccessData`) by default.

Now you can run this `Wai.Application` on some port to bring up your mock server.

```
main :: IO ()
main = run 8000 mockApp
```

You can even mock the requests for the routes. To create a mock `Request` for route `User` declared in <Section name here>, we can write:

```
req <- mockClient (Res :: Resource GET User)
```

We can use this `req` while calling `client` function to make a `Request`.