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

Mailchimp and Building Our Own Integration

Welcome to the third and final part in our series on Haskell API integrations! We started this series off by learning how to send and receive text messages using Twilio. Then we learned how to send emails using the Mailgun service. Both of these involved applying existing Haskell libraries suited to the tasks. This week, we'll learn how to connect with Mailchimp, a service for managing email subscribers. Only this time, we're going to do it a bit differently.

There are a couple different Haskell libraries out there for Mailchimp. But we're not going to use them! Instead, we'll learn how we can use Servant to connect directly to the API. This should give us some understanding for how to write one of these libraries. It should also make us more confident of integrating with any API of our choosing!

To follow along the code for this article, you can read the code on our Github Repository! For this part, you'll want to focus on the Subscribers module and the Full Server.

The topics in this article are quite advanced. If any of it seems crazy confusing, there are plenty of easier resources for you to start off with!

1. If you've never written Haskell at all, see our Beginners Checklist to learn how to get started!
2. If you want to learn more about the Servant library we'll use, check out my talk from BayHac 2017 and download the slides and companion code.
3. Our Production Checklist has some further resources and libraries you can look at for common tasks like writing web APIs!

MAILCHIMP 101

Now let's get going! To integrate with Mailchimp, you first need to make an account and create a mailing list! This is pretty straightforward, and you'll want to save 3 pieces of information as environment variables. First is base URL for the Mailchimp API. It will look like:

```
https://{server}.api.mailchimp.com/3.0
```

Where {server} should be replaced by the region that appears in the URL when you log into your account. For instance, mine is: https://us14.api.mailchimp.com/3.0. You'll also need your API Key, which appears in the "Extras" section under your account profile (you might need to create one). Then you'll also want to save the name of the mailing list you made.

OUR 3 TASKS

We'll be trying to perform three tasks using the API. First, we want to derive the internal "List ID" of our particular Mailchimp list. We can do this by analyzing the results of calling the endpoint at:

```
GET {base-url}/lists
```

It will give us all the information we need about our different mailing lists.

Once we have the list ID, we can use that to perform actions on that list. We can for instance retrieve all the information about the list's subscribers by using:

```
GET {base-url}/lists/{list-id}/members
```

We'll add an extra count param to this, as otherwise we'll only see the results for 10 users:

```
GET {base-url}/lists/{list-id}/members?count=2000
```

Finally, we'll use this same basic resource to subscribe a user to our list. This involves a POST request and a request body containing the user's email address. Note that all requests and responses will be in the JSON format:

```
POST {base-url}/lists/{list-id}/members

{
```

```
  "email_address": "person@email.com",
  "status": "subscribed"
}
```

On top of these endpoints, we'll also need to add basic authentication to every API call. This is where our API key comes in. Basic auth requires us to provides a "username" and "password" with every API request. Mailchimp doesn't care what we provide as the username. As long as we provide the API key as the password, we'll be good. Servant will make it easy for us to do this.

TYPES AND INSTANCES

Once we know the structure of the API, our next goal is to define wrapper types. These will allow us to serialize our data into the format demanded by the Mailchimp API. We'll have four different newtypes. The first will represent a single email list in a response object. All we care about is the list name and its ID, which we represent with Text:

```
newtype MailchimpSingleList = MailchimpSingleList (Text, Text)
  deriving (Show)
```

Now we want to be able to deserialize a response containing many different lists:

```
newtype MailchimpListResponse =
  MailchimpListResponse [MailchimpSingleList]
deriving (Show)
```

In a similar way, we want to represent a single subscriber and a response containing several subscribers:

```
newtype MailchimpSubscriber = MailchimpSubscriber
  { unMailchimpSubscriber :: Text }
deriving (Show)

newtype MailchimpMembersResponse =
  MailchimpMembersResponse [MailchimpSubscriber]
deriving (Show)
```

The purpose of using these newtypes is so we can define JSON instances for them. In general, we only need FromJSON instances so we can deserialize the response we get back from the API. Here's what our different instances look like:

```
instance FromJSON MailchimpSingleList where
  parseJSON = withObject "MailchimpSingleList" $ \o -> do
    name <- o .: "name"
    id_ <- o .: "id"
    return $ MailchimpSingleList (name, id_)

instance FromJSON MailchimpListResponse where
  parseJSON = withObject "MailchimpListResponse" $ \o -> do
    lists <- o .: "lists"
    MailchimpListResponse <$> forM lists parseJSON

instance FromJSON MailchimpSubscriber where
  parseJSON = withObject "MailchimpSubscriber" $ \o -> do
    email <- o .: "email_address"
    return $ MailchimpSubscriber email

instance FromJSON MailchimpListResponse where
  parseJSON = withObject "MailchimpListResponse" $ \o -> do
    lists <- o .: "lists"
    MailchimpListResponse <$> forM lists parseJSON
```

And then, we need a ToJSON instance for our individual subscriber type. This is because we'll be sending that as a POST request body:

```
instance ToJSON MailchimpSubscriber where
  toJSON (MailchimpSubscriber email) = object
    [ "email_address" .= email
    , "status" .= ("subscribed" :: Text)
    ]
```

Finally, we also need one extra type for the subscription response. We don't actually care what the information is, but if we simply return (), we'll get a serialization error because it returns a JSON object, rather than "null".

```
data SubscribeResponse = SubscribeResponse

instance FromJSON SubscribeResponse where
  parseJSON _ = return SubscribeResponse
```

DEFINING A SERVER TYPE

Now that we've defined our types, we can go ahead and define our actual API using Servant. This might seem a little confusing. After all, we're not building a Mailchimp Server! But by writing this API, we can use the client function from the servant-client library. This will derive all the client functions we need to call into the Mailchimp API. Let's start by defining a combinator that will description our authentication format using BasicAuth. Since we aren't writing any server code, we don't need a "return" type for our authentication.

```
type MCAuth = BasicAuth "mailchimp" ()
```

Now let's write the lists endpoint. It has the authentication, our string path, and then returns us our list response.

```
type MailchimpAPI =
  MCAuth :> "lists" :> Get '[JSON] MailchimpListResponse :<|>
  ...
```

For our next endpoint, we need to capture the list ID as a parameter. Then we'll add the extra query parameter related to "count". It will return us the members in our list.

```
type Mailchimp API =
  ...
  MCAuth :> "lists" :> Capture "list-id" Text :> "members" :>
    QueryParam "count" Int :> Get '[JSON] MailchimpMembersResponse
```

Finally, we need the "subscribe" endpoint. This will look like our last endpoint, except without the count parameter and as a post request. Then we'll include a single subscriber in the request body.

```
type Mailchimp API =
  ...
```

```
  MCAuth :> "lists" :> Capture "list-id" Text :> "members" :>
    ReqBody '[JSON] MailchimpSubscriber :> Post '[JSON] SubscribeResponse

mailchimpApi :: Proxy MailchimpApi
mailchimpApi = Proxy :: Proxy MailchimpApi
```

Now with servant-client, it's very easy to derive the client functions for these endpoints. We define the type signatures and use client. Note how the type signatures line up with the parameters that we expect based on the endpoint definitions. Each endpoint takes the BasicAuthData type. This contains a username and password for authenticating the request.

```
fetchListsClient :: BasicAuthData -> ClientM MailchimpListResponse
fetchSubscribersClient :: BasicAuthData -> Text -> Maybe Int -> ClientM
MailchimpMembersResponse
subscribeNewUserClient :: BasicAuthData -> Text -> MailchimpSubscriber -> ClientM ()
( fetchListsClient :<|>
  fetchSubscribersClient :<|>
  subscribeNewUserClient) = client mailchimpApi
```

RUNNING OUR CLIENT FUNCTIONS

Now let's write some helper functions so we can call these functions from the IO monad. Here's a generic function that will take one of our endpoints and call it using Servant's runClientM mechanism.

```
runMailchimp :: (BasicAuthData -> ClientM a) -> IO (Either ServantError a)
runMailchimp action = do
  baseUrl <- getEnv "MAILCHIMP_BASE_URL"
  apiKey <- getEnv "MAILCHIMP_API_KEY"
  trueUrl <- parseBaseUrl baseUrl
  -- "username" doesn't matter, we only care about API key as "password"
  let userData = BasicAuthData "username" (pack apiKey)
  manager <- newTlsManager
  let clientEnv = ClientEnv manager trueUrl
```

```
  runClientM (action userData) clientEnv
```

First we derive our environment variables and get a network connection manager. Then we run the client action against the ClientEnv. Not too difficult.

Now we'll write a function that will take a list name, query the API for all our lists, and give us the list ID for that name. It will return an Either value since the client call might actually fail. It calls our list client and filters through the results until it finds a list whose name matches. We'll return an error value if the list isn't found.

```
fetchMCListId :: Text -> IO (Either String Text)
fetchMCListId listName = do
  listsResponse <- runMailchimp fetchListsClient
  case listsResponse of
    Left err -> return $ Left (show err)
    Right (MailchimpListResponse lists) ->
      case find nameMatches lists of
        Nothing -> return $ Left "Couldn't find list with that name!"
        Just (MailchimpSingleList (_, id_)) -> return $ Right id_
  where
    nameMatches :: MailchimpSingleList -> Bool
    nameMatches (MailchimpSingleList (name, _)) = name == listName
```

Our function for retrieving the subscribers for a particular list is more straightforward. We make the client call and either return the error or else unwrap the subscriber emails and return them.

```
fetchMCListMembers :: Text -> IO (Either String [Text])
fetchMCListMembers listId = do
  membersResponse <- runMailchimp
    (\auth -> fetchSubscribersClient auth listId (Just 2000))
  case membersResponse of
    Left err -> return $ Left (show err)
    Right (MailchimpMembersResponse subs) -> return $
      Right (map unMailchimpSubscriber subs)
```

And our subscribe function looks very similar. We wrap the email up in the MailchimpSubscriber type and then we make the client call using runMailchimp.

```
subscribeMCMember :: Text -> Text -> IO (Either String ())
subscribeMCMember listId email = do
  subscribeResponse <- runMailchimp (\auth ->
    subscribeNewUserClient auth listId (MailchimpSubscriber email))
  case subscribeResponse of
    Left err -> return $ Left (show err)
    Right _ -> return $ Right ()
```

MODIFYING THE SERVER

The last step of this process is to incorporate the subscription into our server, on the "subscribe" handler. First, since we wrote all our Mailchimp functions as IO (Either String ...), we'll write a quick helper for running such actions in the Handler monad. This monad lets us throw "Error 500" if these calls fail, echoing the error message:

```
tryIO :: IO (Either String a) -> Handler a
tryIO action = do
  result <- liftIO action
  case result of
    Left e -> throwM $ err500 { errBody = BSL.fromStrict $ BSC.pack (show e)}
    Right x -> return x
```

Now using this helper and our Mailchimp functions above, we can write a fairly clean handler that handles subscribing to the list:

```
subscribeEmailHandler :: Text -> Handler ()
subscribeEmailHandler email = do
  listName <- pack <$> liftIO (getEnv "MAILCHIMP_LIST_NAME")
  listId <- tryIO (fetchMailchimpListId listName)
  tryIO (subscribeMailchimpMember listId email)
```

And now the user will be subscribed on our mailing list after they click the link!

CONCLUSION

That wraps up our exploration of Mailchimp and our series on integrating APIs with Haskell! In part 1 of this series, we saw how to send and receive texts using the Twilio API. Then in part 2, we sent emails to our users with Mailgun. Finally, we used the Mailchimp API to more reliably store our list of subscribers. We even did this from scratch, without the use of a library like we had for the other two effects. We used Servant to great effect here, specifying what our API would look like even though we weren't writing a server for it! This enabled us to derive client functions that could call the API for us.

This series combined tons of complex ideas from many other topics. If you were a little lost trying to keep track of everything, I highly recommend you check out our Real World Haskell series. It'll teach you a lot of cool techniques, such as how to connect Haskell to a database and set up a server with Servant. You should also download our Production Checklist for some more ideas about cool libraries!

And of course, if you're a total beginner at Haskell, hopefully you understand now that Haskell CAN be used for some very advanced functionality. Furthermore, we can do so with incredibly elegant solutions that separate our effects very nicely. If you're interested in learning more about the language, download our free Beginners Checklist!

Revision #1
Created 2022-03-11 16:26:28 UTC by gasick
Updated 2022-03-11 17:11:16 UTC by gasick