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

Haskell API Integrations

Haskell claims to have functional purity, and hence lacks side effects. This is of course a bit of a simplication that lends itself to many jokes. Of course we can do lots of interesting communication tasks with Haskell, as long as we know the right libraries! In this series, we explore a variety of ways we can interact with our users over the internet!

- Twilio and Text Messages
- Sending Emails with Mailgun
- Mailchimp and Building Our Own Integration

Twilio and Text Messages

Welcome to part 1 of our series on Haskell API integrations! Writing our own Haskell code using only simple libraries is fun. But we can't do everything from scratch. There are all kinds of cools services out there to use so we don't have to. We can interface with a lot of these by using APIs. Often, the most well supported APIs use languages like Python and Javascript. But adventurous Haskell developers have also developed bindings for these systems! So in this series, we'll explore a couple of these. We'll also see how to develop our own integration in case one isn't available for the service we want.

In this first part, we'll focus on the Twilio API. We'll see how we can send SMS messages from our Haskell code using the twilio library. We'll also write a simple server to use Twilio's callback system to receive text messages and process them programmatically. You can follow along with the code here on the Github repository for this series. You can find the code for this part in two modules: the SMS module, which has re-usable SMS functions, and the SMSServer module, which has the Servant related code. If you're already familiar with the Haskell Twilio library, you can move onto part 2 where we discuss sending emails with Mailgun!

Of course, none of this is useful if you've never written any Haskell before! If you want to get started with the language basics, download our Beginners Checklist. To learn more about advanced techniques and libraries, grab our Production Checklist!

SETTING UP OUR ACCOUNT

Naturally, you'll need a Twilio account to use the Twilio API. Once you have this set up, you need to add your first Twilio number. This will be the number you'll send text messages to. You'll also see it as the sender for other messages in your system. You should also go through the process of verifying your own phone number. This will allow you to send and receive messages on that phone without "publishing" your app.

You also need a couple other pieces of information from your account. There's the account SID, and the authentication token. You can find these on the dashboard for your project on the Twilio page.

You'll need these values in your code. But since you don't want to put them into version control, you should save them as environment variables on your machine. Then when you need to, you can fetch them like so:

```
fetchSid :: IO String
fetchSid = getEnv "TWILIO_ACCOUT_SID"

fetchToken :: IO String
fetchToken = getEnv "TWILIO_AUTH_TOKEN"
```

In addition, you should get a Twilio number for your account to send messages from (this might cost a dollar or so). For testing purposes, you should also verify your own phone number on the account dashboard so you can receive messages. Save these as environment variables as well.

```
import Data.Text (pack)

fetchTwilioNumber :: IO Text
fetchTwilioNumber = pack <$> getEnv "TWILIO_PHONE_NUMBER"

fetchUserNumber :: IO Text
fetchUserNumber = pack <$> getEnv "TWILIO_USER_NUMBER"
```

SENDING A MESSAGE

The first thing we'll want to do is use the API to actually send a text message. We perform Twilio actions within the Twilio monad. It's rather straightforward to access this monad from IO. All we need is the runTwilio' function:

```
runTwilio' :: IO String -> IO String -> Twilio a -> IO a
```

The first two parameters to this function are IO actions to fetch the account SID and auth token like we have above. Then the final parameter of course is our Twilio action.

```
sendBasicMessage :: IO ()
sendBasicMessage = runTwilio' fetchSid fetchToken $ do
  ...
```

To compose a message, we'll use the PostMessage constructor. This takes four parameters. First, the "to" number of our message. Fill this in with the number to your physical phone. Then the second parameter is the "from" number, which has to be our Twilio account's phone number. Then the third parameter is the message itself. The fourth parameter is optional, we can leave it as Maybe. To send the message, all we have to do is use the post function! That's all there is to it!

```
sendBasicMessage :: IO ()
sendBasicMessage = do
  toNumber <- fetchUserNumber
  fromNumber <- fetchTwilioNumber
  runTwilio' fetchSid fetchToken $ do
    let msg = PostMessage toNumber fromNumber "Hello Twilio!"
    _ <- post msg
    return ()
```

And just like that, you've sent your first Twilio message! You can just run this IO function from GHCI, and it will send you a text message as long as everything is set up with your Twilio account! Note that it does cost a small amount of money to send messages over Twilio. But a trial account should give you enough free credit to experiment a little bit (as well as cover the initial number).

RECEIVING MESSAGES

Now, it's a little more complicated to deal with incoming messages. First, you need a web server running on the internet. For basic projects like this, I tend to rely on Heroku. If you fork our Github repo, you can easily turn it into your own Heroku server! Just take a look at these instructions in our repo!

The first thing we need to do is create a webhook on our Twilio account. To do this, go to "Manage Numbers" from your project dashboard page (Try this link if you can't find it). Then select your Twilio number. You'll now want to scroll to the section called "Messaging" and then within that, find "A Message Comes In". You want to select "Webhook" in the dropdown. Then you'll need to specify a URL where your server is located, and select "HTTP Post". You can see in this screenshot that we're using my Heroku server with the endpoint path /api/sms.

With this webhook set up, Twilio will send a post request to the endpoint every time a user texts our number. The request will contain the message and the number of the sender. So let's set up a server using Servant to pick up that request.

We'll start by specifying a simple type to encode the message we'll receive from Twilio:

```
data IncomingMessage = IncomingMessage
  { fromNumber :: Text
  , body :: Text
  }
```

Twilio encodes its post request body as FormURLEncoded. In order for Servant to deserialize this, we'll need to define an instance of the FromForm class for our type. This function takes in a hash map from keys to lists of values. It will return either an error string or our desired value.

```
instance FromForm IncomingMessage where
  fromForm :: Form -> Either Text IncomingMessage
  fromForm (From form) = ...
```

So form is a hash map, and we want to look up the "From" number of the message as well as its body. Then as long as we find at least one result for each of these, we'll return the message. Otherwise, we return an error.

```
instance FromForm IncomingMessage where
  fromForm :: Form -> Either Text IncomingMessage
  fromForm (From form) = case lookupResults of
    Just ((fromNumber : _), (body : _)) ->
      Right $ IncomingMessage fromNumber body
    Just _ -> Left "Found the keys but no values"
    Nothing -> Left "Didn't find keys"
    where
      lookupResults = do
        fromNumber <- HashMap.lookup "From" form
        body <- HashMap.lookup "Body" form
        return (fromNumber, body)
```

Now that we have this instance, we can finally define our API endpoint! All it needs are the simple path components and the request body. For now, we won't actually post any response.

```
type SMSServerAPI =
  "api" :> "sms" :> ReqBody '[FormUrlEncoded] IncomingMessage :> Post '[JSON] ()
```

WRITING OUR HANDLER Now let's we want to write a handler for our endpoint that will echo the user's message back to them.

```
incomingHandler :: IncomingMessage -> Handler ()
incomingHandler (IncomingMessage from body) = liftIO $ do
  twilioNum <- fetchTwilioNumber
  runTwilio' fetchSid fetchToken $ do
    let newMessage = PostMessage from twilioNum body Nothing
    _ <- post newMessage
    return ()
```

We'll also add an extra endpoint to "ping" our server, just so it's easier to verify that the server is working at a basic level. It will return the string "Pong" to signify the request has been received.

```
type SMSServerAPI =
  "api" :> "sms" :> ReqBody '[FormUrlEncoded] IncomingMessage :> Post '[JSON] () :<|>
  "api" :> "ping" :> Get '[JSON] String

pingHandler :: Handler String
pingHandler = return "Pong"
```

And now we wrap up with some of the Servant mechanics to run our server.

```
smsServerAPI :: Proxy SMSServerAPI
smsServerAPI = Proxy :: Proxy SMSServerAPI

smsServer :: Server SMSServerAPI
smsServer = incomingHandler :<|> pingHandler

runServer :: IO ()
runServer = do
  port <- read <$> getEnv "PORT"
  run port (serve smsServerAPI smsServer)
```

And now if we send a text message to our Twilio number, we'll see that same message back as a reply!

CONCLUSION

In this part, we saw how we could use just a few simple lines of Haskell to send and receive text messages. There was a fair amount of effort required in using the Twilio tools themselves, but most of that is easy once you know where to look! You can now move onto part 2, where we'll explore how we can send emails with the Mailgun API. We'll see how we can combine text and email for some pretty cool functionality.

An important thing making these apps easy is knowing the right tools to use! One of the tools we used in this part was the Servant web API library. To learn more about this, be sure to check out our Real World Haskell Series. For more ideas of web libraries to use, download our Production Checklist.

And if you've never written Haskell before, hopefully I've convinced you that it IS possible to do some cool things with the language! Download our Beginners Checklist to get stated!

Sending Emails with Mailgun

In part 1 of this series, we started our exploration of the world of APIs by integrating Haskell with Twilio. We were able to send a basic SMS message, and then create a server that could respond to a user's message. In this part, we're going to venture into another type of effect: sending emails. We'll be using Mailgun for this task, along with the Hailgun Haskell API for it.

You can take a look at the full code for this article by looking on our Github repository. For this part, you'll want to look at the Email module and the Full Server. If this article sparks your curiosity for more Haskell libraries, you should download our Production Checklist! If you've already read this part, feel free to move onto part 3 where we look at managing an email list with Mailchimp!

MAKING AN ACCOUNT

To start with, we'll need a mailgun account obviously. Signing up is free and straightforward. It will ask you for an email domain, but you don't need one to get started. As long as you're in testing mode, you can use a sandbox domain they provide to host your mail server.

With Twilio, we had to specify a "verified" phone number that we could message in testing mode. Similarly, you will also need to designate a verified email address. Your sandboxed domain will only be able to send to this address. You'll also need to save a couple pieces of information about your Mailgun account. In particular, you need your API Key, the sandboxed email domain, and the reply address for your emails to use. You'll also want the verified email you can send to. Save these as environment variables on your local system and remote machine.

BASIC EMAIL

Now let's get a feel for the Hailgun code by sending a basic email. All this occurs in the simple IO monad. We ultimately want to use the function sendEmail, which requires both a HailgunContext and a HailgunMessage:

```
sendEmail
  :: HailgunContext
  -> HailgunMessage
  -> IO (Either HailgunErrorResponse HailgunSendResponse)
```

We'll start by retrieving our environment variables. With our domain and API key, we can build the HailgunContext we'll need to pass as an argument.

```
import Data.ByteString.Char8 (pack)

sendBasicMail :: IO ()
sendBasicMail = do
  domain <- getEnv "MAILGUN_DOMAIN"
  apiKey <- getEnv "MAILGUN_API_KEY"
  replyAddress <- pack <$> getEnv "MAILGUN_REPLY_ADDRESS"
  toAddress <- pack <$> getEnv "MAILGUN_USER_ADDRESS"
  -- Last argument is an optional proxy
  let context = HailgunContext domain apiKey Nothing
  ...
```

Now to build the message itself, we'll use a builder function hailgunMessage. It takes several different parameters:

```
hailgunMessage
 :: MessageSubject
 -> MessageContent
 -> UnverifiedEmailAddress -- Reply Address, just a ByteString
 -> MessageRecipients
 -> [Attachment]
 -> Either HailgunErrorMessage HailgunMessage
```

These are all very easy to fill in. The MessageSubject is Text and then we'll pass our reply address and verified address from above. For the content, we'll start by using the TextOnly constructor for a plain text email. We'll see an example later of how we can use HTML in the content:

```
sendMail :: IO ()
sendMail = do
  ...
  replyAddress <- pack <$> getEnv "MAILGUN_REPLY_ADDRESS"
```

```
  let msg = mkMessage replyAddress
  ...
  where
    mkMessage toAddress replyAddress = hailgunMessage
      "Hello Mailgun!"
      (TextOnly "This is a test message.")
      replyAddress
      ...
```

The MessageRecipients type has three fields. First are the direct recipients, then the CC'd emails, and then the BCC'd users. We're only sending to a single user at the moment. So we can take the emptyMessageRecipients item and modify it. We'll wrap up our construction by providing an empty list of attachments for now:

```
where
  mkMessage toAddress replyAddress = hailgunMessage
    "Hello Mailgun!"
    (TextOnly "This is a test message.")
    replyAddress
    (emptyMessageRecipients { recipientsTo = toAddress } )
    []
```

If there are issues, the hailgunMessage function can throw an error, as can the sendEmail function itself. But as long as we check these errors, we're in good shape to send out the email!

```
sendBasicEmail :: IO ()
sendBasicEmail = do
  domain <- getEnv "MAILGUN_DOMAIN"
  apiKey <- getEnv "MAILGUN_API_KEY"
  replyAddress <- pack <$> getEnv "MAILGUN_REPLY_ADDRESS"
  toAddress <- pack <$> getEnv "MAILGUN_USER_ADDRESS"
  let context = HailgunContext domain apiKey Nothing
  case mkMessage toAddress replyAddress of
    Left err -> putStrLn ("Making failed: " ++ show err)
    Right msg -> do
      result <- sendEmail context msg -- << Send Email Here!
      case result of
        Left err -> putStrLn ("Sending failed: " ++ show err)
        Right resp -> putStrLn ("Sending succeeded: " ++ show resp)
```

```
  where
    mkMessage toAddress replyAddress = hailgunMessage
```

Notice how it's very easy to build all our functions up when we start with the type definitions. We can work through each type and figure out what it needs. I reflect on this idea some more in this article on Compile Driven Learning, which is part of our Haskell Brain Series for newcomers to Haskell!

EXTENDING OUR SERVER

Now that we know how to send emails, let's incorporate it into our server! We'll start by writing another data type that will represent the potential commands a user might text to us. For now, it will only have the "subscribe" command.

```
data SMSCommand = SubscribeCommand Text
```

Now let's write a function that will take their message and interpret it as a command. If they text subscribe {email}, we'll send them an email!

```
messageToCommand :: Text -> Maybe SMSCommand
messageToCommand messageBody = case splitOn " " messageBody of
  ["subscribe", email] -> Just $ SubscribeCommand email
  _ -> Nothing
```

Now we'll extend our server handler to reply. If we interpret their command correctly, we'll send a replay email with a new function sendSubscribeEmail. Otherwise, we'll send them back a text saying we couldn't understand them.

```
incomingHandler :: IncomingMessage -> Handler ()
incomingHandler (IncomingMessage from body) = liftIO $ do
  case messageToCommand body of
    Nothing -> do
      twilioNum <- fetchTwilioNumber
      runTwilio' fetchSid fetchToken $ do
        let body = "Sorry, we didn't understand that request!"
        let newMessage = PostMessage from twilioNum body Nothing
```

```
        _ <- post newMessage
        return ()
    Just (SubscribeCommand email) -> sendSubscribeEmail email

sendSubscribeEmail :: Text -> IO ()
sendSubscribeEmail = ...
```

Now all we have to do is construct this new email. Let's add a couple new features beyond the basic email we made before.

MORE ADVANCED EMAILS

Let's start by adding an attachment. We can build an attachment by providing a path to a file as well as a string describing it. To get this file, our message making function will need the current running directory.

```
mkSubscribeMessage :: ByteString -> ByteString -> FilePath -> Either HailgunErrorMessage
HailgunMessage
mkSubscribeMessage replyAddress subscriberAddress currentDir =
  hailgunMessage
    "Thanks for signing up!"
    content
    replyAddress
    (emptyMessageRecipients { recipientsTo = [subscriberAddress] })
    -- Notice the attachment!
    [ Attachment
        (rewardFilepath currentDir)
        (AttachmentBS "Your Reward")
    ]
  where
    content = TextOnly "Here's your reward!"

rewardFilepath :: FilePath -> FilePath
rewardFilepath currentDir = currentDir ++ "/attachments/reward.txt"
```

As long as the reward file lives on our server, that's all we need to do to send that file to the user. Now to show off one more feature, let's change the content of our email so that it contains some

HTML instead of only text. In particular, we'll give them the chance to confirm their subscription by clicking a link to our server. All that changes here is that we'll use the TextAndHTML constructor instead of TextOnly. We do want to provide a plain text interpretation of our email in case HTML can't be rendered for whatever reason. Notice the use of the `<a>` tags for the link:

```
content = TextAndHTML
   textOnly
   ("Here's your reward! To confirm your subscription, click " <>
     link <> "!")
  where
    textOnly = "Here's your reward! To confirm your subscription, go to "
       <> "https://mmh-apis.herokuapp.com/api/subscribe/"
       <> subscriberAddress
       <> " and we'll sign you up!"
   link = "<a href=\"https://mmh-apis.herokuapp.com/api/subscribe/"
     <> subscriberAddress <> "\">this link</a>"
```

If you're running our code on your own Heroku server, you'll need to change the app name (mmh-apis) in the URLs above.

Then to round this code out, all we'll need to do is fill out sendSubscribeEmail to use our function above. It will reference the same environment variables we have in our other function:

```
sendSubscribeEmail :: Text -> IO ()
sendSubscribeEmail email = do
  domain <- getEnv "MAILGUN_DOMAIN"
  apiKey <- getEnv "MAILGUN_API_KEY"
  replyAddress <- pack <$> getEnv "MAILGUN_REPLY_ADDRESS"
  let context = HailgunContext domain apiKey Nothing
  currentDir <- getCurrentDirectory
  case mkSubscribeMessage replyAddress (encodeUtf8 email) currentDir of
    Left err -> putStrLn ("Making failed: " ++ show err)
    Right msg -> do
      result <- sendEmail context msg
      case result of
        Left err -> putStrLn ("Sending failed: " ++ show err)
        Right resp -> putStrLn ("Sending succeeded: " ++ show resp)
```

CONCLUSION

Our course, we'll want to add a new endpoint to our server to handle the subscribe link we added above. But we'll handle that in the last part of the series. Hopefully from this part, you've learned that sending emails with Haskell isn't too scary. The Hailgun API is quite intuitive and when you break things down piece by piece and look at the types involved.

There's a lot of advanced material in this series, so if you think you need to backtrack, don't worry, we've got you covered! Our Real World Haskell Series will teach you how to use libraries like Persistent for database management and Servant for making an API. For some more libraries you can use to write enhanced Haskell, download our Production Checklist!

If you've never programmed in Haskell at all, you should try it out! Download our Haskell Beginner's Checklist or read our Liftoff Series!

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!