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

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!

Revision #1
Created 2022-03-11 16:22:45 UTC by gasick
Updated 2022-03-11 17:11:16 UTC by gasick