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

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!

Revision #1

Created 11 March 2022 16:24:21 by gasick

Updated 11 March 2022 17:11:16 by gasick