Writing a Telegram bot in Haskell and hosting it on AWS Lambda

I've been wanting to get into Haskell for a while now. Living in the same household as an adamant Haskell enthusiast, there was never a way around eventually diving in, but I've heard many people praise it and describe it as a better, cleaner Python. I've *also* been wanting to create a personal bot on Telegram to automate things I do on a daily basis, such as sifting through job and apartment offers as well as translating words into foreign languages, note taking, or checking the weather. So I thought, why not combine the two?

I've spent a bunch of time getting AWS to cooperate with my Haskell code. My main motivation for writing up this document is, well, to document the process so that I can refer to it in the future if needed. It'll also help me reflect on design choices and fix dirty bits along the way. It's like advanced rubber duck debugging.

Current state of the bot

As of writing this *(June 18)*, the bot is unfinished, but supports a handful of commands.
The code for these commands sits on AWS Lambda. The bot responds very quickly. Adding more commands is easy enough now that I have a basic framework to use.

Requirements

- A Telegram bot + token
- An AWS account
- A setup for writing Haskell
- Docker

Telegram bot + token

These are straightforward enough. I messaged @BotFather on Telegram and followed the instructions to create a new bot. I had to give it a unique @username and then received the corresponding API token. The token is meant to be kept private.

AWS account

I created a free account on aws.amazon.com. I did have to provide credit card information, but the services I need for hosting our bot (Lambda, API Gateway, CloudWatch) have a free tier that more than suffices for a project as small in scope as this.

How things will work together

- My Haskell project will consist of two main parts: **the (expandable) part that does some work for me**, say, scraping a website for its job offers, and a **handler function** that will correctly execute our program when it's called on AWS Lambda.
- **AWS Lambda** will host my Haskell code and execute a handler function upon request. Specifically, I will be using an **AWS API Gateway.**
- The API Gateway will communicate with the **Telegram API** using a **webhook**.

Code: Job offer scraper

I have my bot ready to fire and my AWS account standing by. Time to build a tool.
I need a job. I've been having a look at offers on slo-tech.com daily, but this can be automated! I will need a few packages for the stop, namely `Aeson` for JSON formatting (and the corresponding language extensions), `Text` to use instead of Strings, `Scalpel` to scrape the website, and `Req` to make HTTP requests.
I start by declaring a datatype to hold the offers. I also declare a Card type, synonymous with the raw text extracted from the website, and a synonym for `scrapeStringLike` for readibility.

```
extractFrom = scrapeStringLike
type Card = Text
data JobAd = JobAd {  title         :: Text
                    , keywords      :: [Text]
                    , companyName   :: Text
                    , entryDate     :: Text
                    , url           :: Text}
deriving (Generic, ToJSON, FromJSON, Show)
```

Next, I declare a function to get the page content for me.

```
getJobPage :: IO Text
getJobPage = runReq defaultHttpConfig $ do
  response <- req GET -- method
                  (https "slo-tech.com" /: "delo")
                  (NoReqBody)
                  bsResponse $
                  mEmpty

  pure $ decodeUtf8 $ responseBody response
```

As this is a simple GET request, I don't need to add any data to my request `(NoReqBody; mEmpty)`, but I do want to handle the response as a ByteString `(bsResponse)`. The function evaluates to the response body, which is just the page content in this case.
I need a way to parse the response now. Here is a function that (maybe) returns a list of cards using this page content.

```
scrapeJobCards :: IO (Maybe [Card])
scrapeJobCards = do
   let scraper = htmls $ ("table" @: [hasClass "forums"] // "tbody" //
"tr")
  page <- getJobPage
  let jobCards' = extractFrom page scraper
  pure jobCards'
```

Ideally, I have my list of offers in textual form now, and the last step is to match the fields of my custom datatype with the list of cards. I made several helper functions for each field, along the lines of this title fetcher.

```
cardTitle :: Card -> Text
cardTitle card = title'
 where
  maybeTitle = innerHTML $ ("td" @: [hasClass "name"] // "a")
  title'     = fromMaybe "" (extractFrom card maybeTitle)
```

Finally, my main function will map each helper function over each card I extracted and return a ByteString with the Aeson-encoded list of ads. I'll limit the output to 5 offers.

```
getJobs :: IO (BS.ByteString)
getJobs = do
  maybeCards <- scrapeJobCards
  let cards = fromMaybe [] maybeCards

  let ads = map
        (\card -> JobAd { title       = (cardTitle card)
                        , keywords    = (cardKeywords card)
                        , companyName = (cardCompanyName card)
                        , entryDate   = (cardEntryDate card)
                        , url         = (cardUrl card)
                        })
        (if length cards > 5 then take 5 cards else cards)
  let adsStr = encode ads
  pure $ toStrict adsStr
```

Final code can be found on https://github.com/rekkuso/telegram-haskell-bot.
So far, so normal. This is where things get interesting.

Code: Lambda Function handler

This is also where things get hairy for me. I don't actually know if this is the correct way to go about things, and I appreciate any possible feedback.

Right now, we can call this tool and receive a list of ads in the CLI at any time. Wouldn't it be nicer to send a command like `/jobs` to a bot on Telegram and get it in a nicely formatted manner, from any of my devices? Let's see.

For this part of the project I will use a package called `aws-lambda-haskell-runtime` . There is some nice documentation on its website. I'm using version 2.0.4. As mentioned earlier, I will be using a webhook for this. The Telegram Bot API is very thorough on this part as well. With this approach, whenever a message is sent to the bot, Telegram's API will POST a request to the URL I specify, along with a JSON-serialized Update object which contains all the data needed to correctly respond. AWS API Gateway will provide this URL.

Request:

Telegram API ———> API Gateway ———> AWS Lambda Function

Response:

Telegram API <——— AWS Lambda Function

In practice, what I need to do is to define two new datatypes - one to parse the POST request that my Gateway forwards to AWS Lambda, and a response to send back to the API. Datatypes modelled after Telegram's Update object with many nested values might look like so.

```
-- Handling incoming POST request
data MessageEvent = MessageEvent
                  { update_id  :: Int
                  , message :: Message }
deriving (Generic, FromJSON, Show)

data Message = Message
                  { message_id :: Int
                  , from :: MessageMetadata
                  , text :: Text }
deriving (Generic, FromJSON, Show)

data MessageMetadata = MessageMetadata
                  { id :: Int
                  , is_bot :: Bool
                  , first_name :: Text
                  , last_name :: Text
                  , username :: Text
                  , language_code :: Text }
deriving (Generic, FromJSON, Show)
```

A possible response datatype might look like so.

```
-- Sending response to the Telegram API
data Response = Response
                  { statusCode :: Int
                  , body :: String
                  } deriving (Generic, ToJSON)
```

With the datatypes in place, the function that actually handles the traffic is next. There is a template for this function in the docs for `aws-lambda-haskell-runtime. Aeson` paired with record wildcards makes data extraction from the `MessageEvent` trivial.

```
handler :: MessageEvent -> Context -> IO (Either String Response)
handler MessageEvent {..} context = do
  let chatid = id $ from message
  let messageContent = text message
  case messageContent of
    "/jobs" -> do
      sendMessage "Latest jobs on Slo-Tech:" chatid
      jobs <- getJobs
      let decodedJobs =
            fromMaybe [] ((decodeStrict jobs) :: Maybe [JobAd])
      mapM (\job -> sendMessage (styleJob job) chatid) decodedJobs
  pure $ Right Response { statusCode = 200, body = "handler ok" }
```

The sendMessage helper function looks as follows and simply sends a message with the specified text to the current chat's id - extracted from the `MessageEvent`. This is done with a POST request with our bot token to Telegram's API, specifically the `/sendMessage` method.

```
sendMessage :: Text -> Int -> IO ()
sendMessage text chatid = runReq defaultHttpConfig $ do
  let payload = object
        [ "text" .= text
        , "chat_id" .= ((show chatid) :: String)
        ]
  r <- req
    POST -- method
    (  https "api.telegram.org"
    /: "bot<TOKEN>"
    /: "sendMessage"
    )
    (ReqBodyJson payload)
    jsonResponse
    mempty
liftIO $ putStrLn $ responseBody r
```

Finally for the project's `main` function in `/App.hs` - it couldn't be simpler. It writes itself using `TemplateHaskell`!

```
{-# LANGUAGE TemplateHaskell #-}
module Main where
import           Lib
import           Aws.Lambda
generateLambdaDispatcher
```

That's all there is to this file.

Final settings

There are a handful more things that need to happen before deployment. Namely, a gateway needs to be created and the Telegram bot needs to be pointed to it. Simple enough, I create an RESTful API on AWS, name it Telegram, create a setting for POST methods and set that to passthrough. This will return a URL that Telegram can be pointed towards by opening the following URL:

```
http://api.telegram.org/bot<BOT TOKEN>/setWebhook?url=<RETURNED URL>
```

Deploying to AWS Lambda

So, the correct datatypes are in place, there exists a handler to work with them, the APIs are ready - time for deployment.
AWS Lambda doesn't natively support Haskell, but it does run a primitive version of Linux (called *Amazon Linux*). But it is possible to upload custom runtimes. So, using Docker and a Makefile provided in the documentation for `aws-lambda-haskell-runtime`, the project can be compiled into a .zip file that can then be uploaded and ran on AWS Lambda after creating an empty Lambda function.

Conclusion

I now have a bot that is expandable in functionality and reliably running on any of my devices - for free, if you don't count the hours spent on it. It's a rewarding feeling. And writing up this little document was surprisingly fun. I might just have to do the same thing with my next project.

All code and this document will stay public on https://github.com/rekkuso/telegram-haskell-bot.