

You know that annoying moment when you are waiting for an important email to come through? Or you really have to send that one formal email to your boss but you have used up all your data allowance and have no WiFi to connect to? Yup, me too.

I had recently found myself in a similar situation and missed out on an amazing opportunity! So when I went to a local hackathon I knew what I would be building - a minimalistic email interface you can control just with SMS. Today I am going to show you how you can build the same app using a little bit of Node.js and Twilio!

### So.. What do you need?

Let’s start with a checklist of all the things we will need to build the app:

* A Twilio Account to get a Twilio Phone Number – [Sign up for free](https://www.twilio.com/try-twilio)!
* Node.js and npm which you can get here: <https://nodejs.org/en/>.
* [Yarn](https://yarnpkg.com/lang/en/docs/install/) - You can use npm as well but you will have to modify the `package.json` `start` scripts to npm from yarn.
* Obviously a working email account with IMAP access.

**Getting Started**

Begin by cloning the [starter project](https://github.com/excerebrose/Emailer/tree/twilio-template) from GitHub and running Yarn to install the project dependencies. Make sure that you are checking out the `twilio-template` branch. Run:

```bash```

git clone -b twilio-template git@github.com:excerebrose/Emailer.git  
cd Emailer  
yarn install

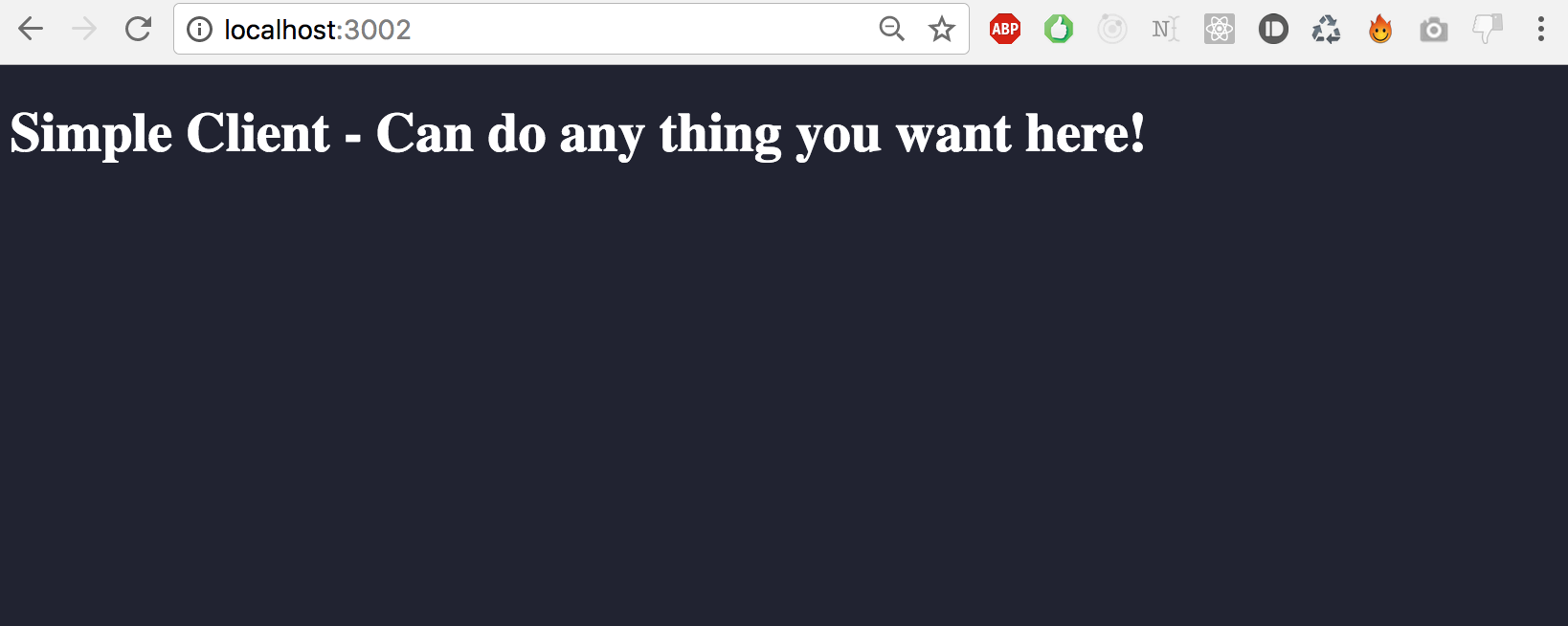
```end```

This repository contains all the skeleton setup we need to get started, including an Express web server that we’ll use to receive notifications of newly arrived emails, and to parse those emails so we can send them as SMS messages. !

Copy the `config.env.example` file to a new file called `config.env` and add the credentials of your Twilio account which you can grab from the Twilio [Console](https://www.twilio.com/console). Don’t worry about the other fields in the config file - we will fill them up as we go along.

Now you should be able to fire up the Express server with `yarn dev:server`, you can also use `node ./server.js` but the former uses [nodemon](https://www.npmjs.com/package/nodemon) to automatically restart the server when our code changes.

To see the server working, head over to <http://localhost:3002/> and you will find the following webpage:



### Receive Email on a Node IMAP Client

To receive emails on our Express server and parse them to be sent as SMS we first need to add a new route that accepts a `POST` request. Open `server.js` and add the following lines of code at then end:

```javascript```

//Handle call from IMAP to trigger a new incoming Email

app.post('/new-email', (*req*, *res*) => {

console.log(req.body);

res.sendStatus(200);

});

```end```

Now when we `POST` something to the url <http://localhost:3002/new-email> it will be shown to us in the console - great for debugging! You can test this by running your server with the command ``yarn dev:server` and in a separate terminal window running

`curl --data "param1=value1&param2=value2" <http://127.0.0.1:3002/new-email>`, you will see the output from your console.log in the terminal window running your server instance.

In the main directory create a new file `mail\_client.js` and add the following lines of code to it:

``` javascript```

**require**('dotenv').config({ path: './config.env' });

const notifier = **require**('mail-notifier');

const inbox = {

user: **process**.**env**.IMAP\_USER,

password: **process**.**env**.IMAP\_PASS,

host: **process**.**env**.IMAP\_HOST,

port: **process**.**env**.IMAP\_PORT,

tls: true,

tlsOptions: { rejectUnauthorized: false }

};

```end```

Woah! That was a huge chunk of code. Let’s walk through it.

First, [dotenv](https://www.npmjs.com/package/dotenv) is an amazing library that allows you to map a given `key: value` file to the `process.env` state of a node application. This is quite handy for sharing the configuration format but at the same time hiding things like authorisation tokens.

Next, we are using another library called [mail-notifier](https://www.npmjs.com/package/mail-notifier) which does exactly what it reads: creates an IMAP Client and listens for updates to the watched folder - can be SPAM, DRAFTS, anything. Read the library’s documentation if you want to use it to monitor any other mailbox. In our app we set up the default configuration for the server to watch our INBOX.

In that configuration note the host, port, username and password environment variables. They depend on the email service you use, and you will have to look for the correct values. Here I am going to set up the variables in my `config.env` file to work with a standard gmail account.

```javascript```

IMAP\_USER = dummy\_account@gmail.com

IMAP\_PASS = dummy\_password

IMAP\_HOST = imap.gmail.com

IMAP\_PORT = 993

```end```

Now let’s add the listener to actually do something when we receive a new email and start the server!

Require the `request` module at the top of the `mail\_client.js` file::

```javascript```

const request = **require**('request');

```end```

The [*request*](https://github.com/request/request) library allows you to perform all sorts of HTTP calls and adds quite a few nice wrappers to ease our coding.

Add the following lines after the `inbox` declaration:

```javascript```

notifier(inbox).on('mail', (*mail*) => {

const mailObject = {

from: mail.from[0].address,

name: mail.from[0].name,

date: mail.receivedDate,

subject: mail.subject ? mail.subject : 'No Subject',

message: mail.text && mail.text.replace(/\s/g,'').length > 0? mail.text : 'No Message body',

attachments: mail.attachments ? mail.attachments.map(*value* => value.fileName).join(' ') : 'None',

};

request.post(**process**.**env**.WEBHOOK\_TARGET).form(mailObject);

**console**.**info**("New Email Received!")

}).start();

```end```

Here we add an event handler that is raised when a new email is received by our IMAP Client. When that happens our code picks the values from the `mail` object and creates a new custom object, which is then posted to a URL running on the Express web server.

Note: make sure the WEBHOOK\_TARGET url in the `config.env` file matches the one of your Express server’s `/new-email`’ route.

Now that we can listen for new email and forward them to our Express server. Let’s make our lives easier and modify the `package.json` script to run both the Express web server and the IMAP client in one command.

```json```

"main": "index.js",

**"scripts"**: {

**"dev:server"**: "nodemon ./server.js",

**"dev:mail"**: "nodemon ./mail\_client.js",

**"dev"**: "yarn dev:server & yarn dev:mail",

"test": "echo \"Error: no test specified\" && exit 1"

},

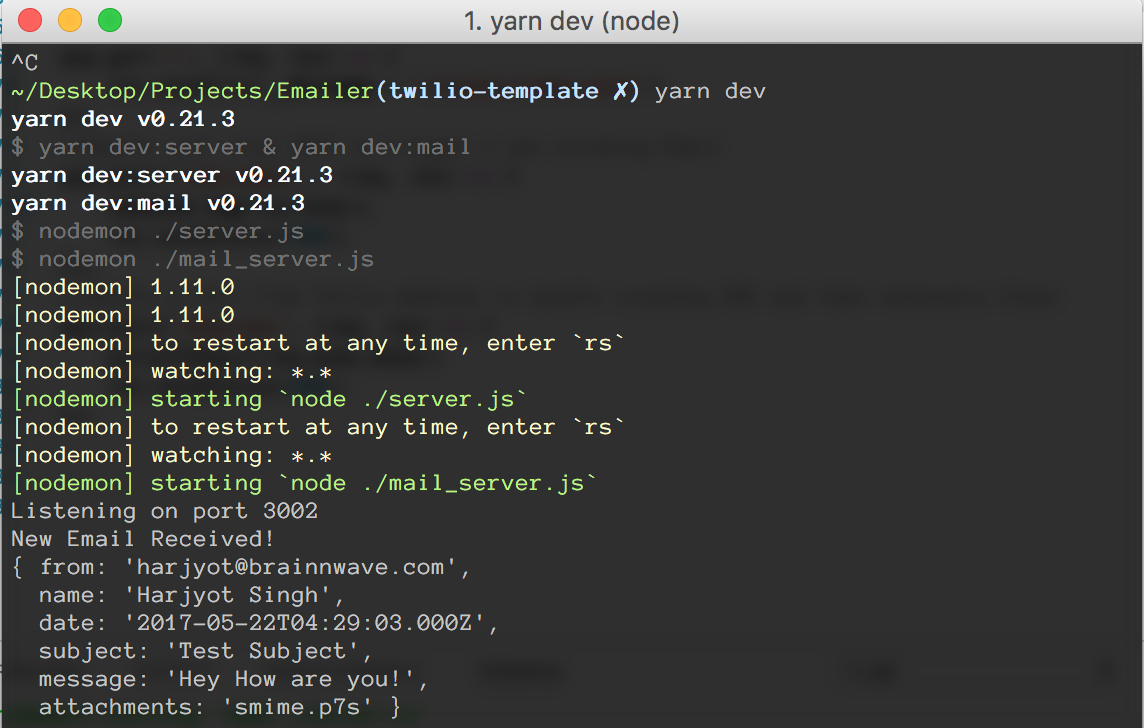
```end```

From this point onwards whenever we run `yarn dev` both servers get started!

Shall we test our application to this point? Go to another (or the same) email account of yours and write up a quick test email and send it to the email you put in *config.env*.

*Note: The IMAP client marks all incoming emails 'UNREAD'. Make sure you don’t have any unread emails in your inbox or else all are gonna come flooding in!*

You should get an output like this in your terminal -



Whoop! That works! So our Express server can now receive our custom mail object and now we are free to do with it whatever we want.

### How to forward an Email as SMS.

Let’s begin by writing up a function that handles sending SMS’s - It’s nothing but a wrapper for our convenience. In `server.js` add:

```javascript```

function sendSMS(*smsbody*) {

twilio\_client.messages.create({

to: **process**.**env**.PHONE\_NUMBER,

from: **process**.**env**.TWILIO\_NUMBER,

body: smsbody,

},

(*err*, *message*) => {

if(err)

**console**.**log**("Error sending SMS:" + err);

});

}

```end```

Now add the following function as well:

```javascript```

function parseToSMS(emailObject) {

const maxChars = 1500; *//Changed to 1500 because of Twilio's injection into free accounts - move it up to 1600 later'*

const formattedDate = moment(emailObject.date).format('ddd, MMM Do YYYY, h:m A');

let messageBody = emailObject.message;

const smsBodyHeader = `New email from:${emailObject.from}, at ${formattedDate}.\nSub: ${emailObject.subject}\nAttachments: ${emailObject.attachments}\n`;

let sms = smsBodyHeader + messageBody;

*//Helpers*

let availableChars = maxChars - smsBodyHeader.length;

const continueMsg = "\nContinued in Next SMS. - Number ";

if (messageBody.length > availableChars) {

console.info("HUGE EMAIL DETECTED! - Splitting to save us!");

let counter = 1;

availableChars -= (continueMsg.length + 2); *// + 2 for the SMS Number place holder*

while(messageBody.length > 0 ) {

console.info("Sending remaining chunks of size:" + availableChars);

if (counter == 1) {

sms = smsBodyHeader + messageBody.substring(0, availableChars) + continueMsg + counter;

availableChars += smsBodyHeader.length *// No More headers in subsequent SMSes..*

}

else if (messageBody.length <= availableChars)

sms = messageBody;

else

sms = messageBody.substring(0,availableChars) + continueMsg + counter;

messageBody = messageBody.substring(availableChars, messageBody.length);

sendSMS(sms);

counter+=1;

}

} else {

sendSMS(sms);

}

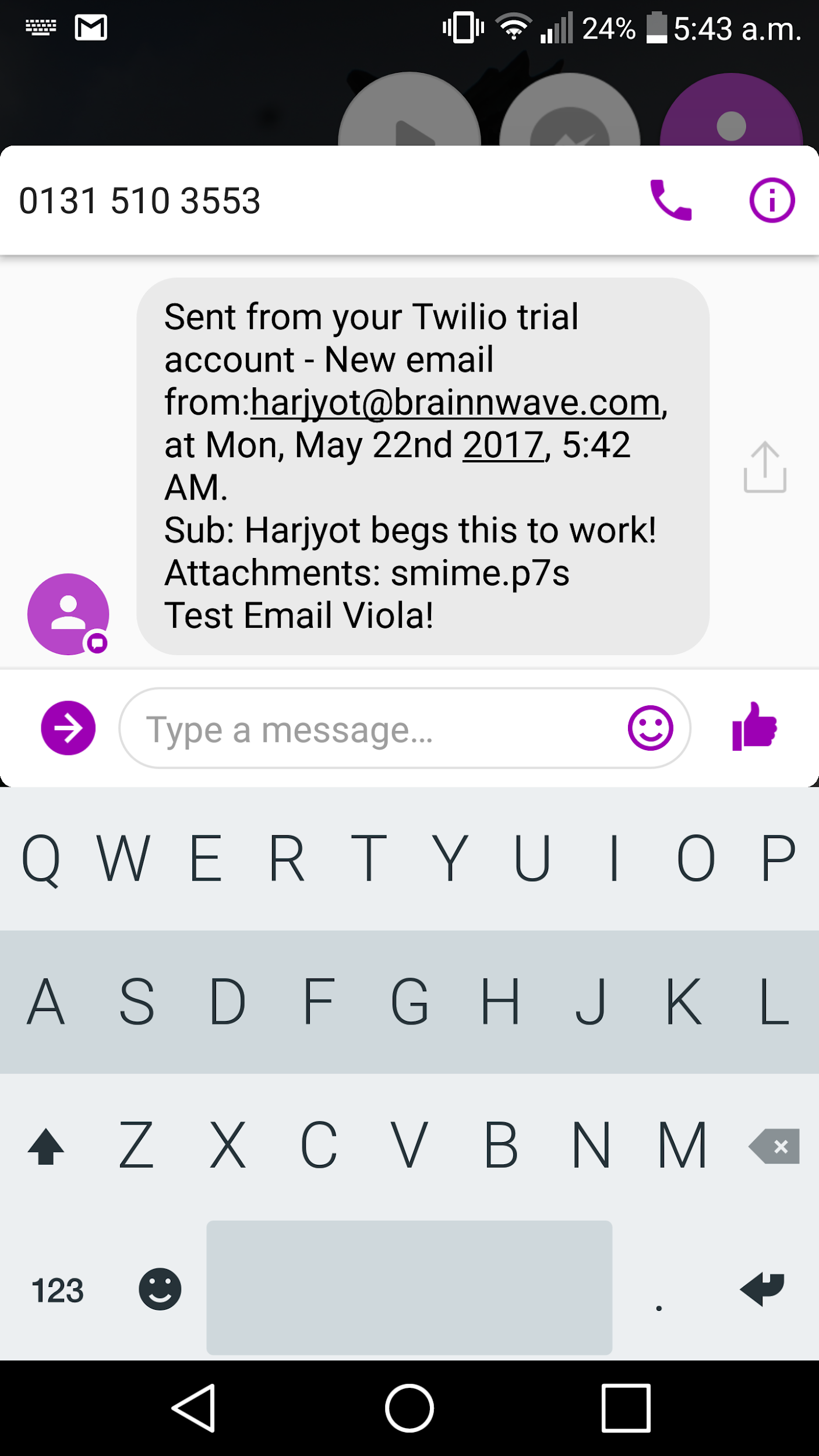
console.info("Done Sending!");

}

```end```

The job of this function is to break down our received email object and convert it into SMS body strings - the content of the SMS we will be sending. This function handles edge cases like when the email overflows the character limit imposed by Twilio, you can follow the comments in the function to understand what’s happening under the hood.

Before running this code make sure you have purchased a SMS compatible Twilio phone number and also verified one of your personal numbers. This is required if you are using a trial account which restricts you to sending SMS to only numbers you have verified.

Finally we modify our `/new-email` route to call the `parseToSMS` function send another test email!

```javascript```

//Handle call from IMAP to trigger a new incoming Email

app.post('/new-email', (*req*, *res*) => {

**console**.**log**(req.body);

parseToSMS(req.body);

res.sentdStatus(200);

});

``` end ```

### Final Thoughts

In this short tutorial we saw how easy it was to set up simple servers in Node, write custom start scripts and use the Twilio SMS API to tackle a problem we encounter in real life. If you are further interested in trying out different things with the project, check out the master branch of the project at the [Repository](https://github.com/excerebrose/Emailer) - It has the same code as above but an added MongoDB Connection that stores all incoming emails.

* Try different machine learning algorithms on the stored incoming emails (anything from sentiment analysis to trend of emails you receive from different users)
* Create an offline dashboard for all your Emails (The setup for a client page is already done! Just navigate to http://localhost:3002 ) in your browser
* Add more functionality like CC, BCC
* Improve on the attachments list

The things mentioned above are only a few of the ways you can extend this project, go wild and go crazy! If you have any questions or just wanna have a quick chat, drop me a message on [FB](https://facebook.com/harjyot) or send a tweet my way [@iharjyot](https://twitter.com/iharjyot).