In my last article we saw how you can [forward emails as SMS](https://www.twilio.com/blog/2017/06/how-to-forward-emails-sms-node-js.html) with a simple app. Why don't we extend this app to also send emails from SMS? Well, let's get started!

### 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](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
* [Ngrok](https://ngrok.com/) - the perfect solution to demo without deploying
* 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-email2sms` branch. Run:

```bash```

git clone -b twilio-template-email2sms git@github.com:excerebrose/Emailer.git  
cd Emailer  
yarn install

```end```

This repository contains the completed app from the end of [the previous article](https://www.twilio.com/blog/2017/06/how-to-forward-emails-sms-node-js.html), so if you don’t want to read through the other post, that’s fine. I will briefly explain the code that we will be reusing from the demo application.

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). In that configuration let’s note the host, port, username and password environment variables. The values in this are the ones your application will send/receive emails from, they depend on the email service you use, and you will have to look for the correct values. Here we will going set up the variables in our `config.env` file to work with a standard gmail account.

*Note - If you use 2-factor-authorization on your account, you will need an app-specific password here.*

```javascript```

IMAP\_USER = dummy\_account@gmail.com

IMAP\_PASS = dummy\_password

IMAP\_HOST = imap.gmail.com

IMAP\_PORT = 993

```end```

There are three options to run the application.

1. `yarn dev:client` only fires the IMAP Client we used to listen for emails to be forwarded
2. `yarn dev:server` fires up the Express server (The file we will be working with, so if you don’t want to use the forward Email functionality just run this!)
3. `yarn dev` which does both.

### Receive SMS on your Express server

Open `server.js` and add the following lines of code at the end to listen for a new incoming SMS message:

```javascript ```

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

console.log(req.body.Body);

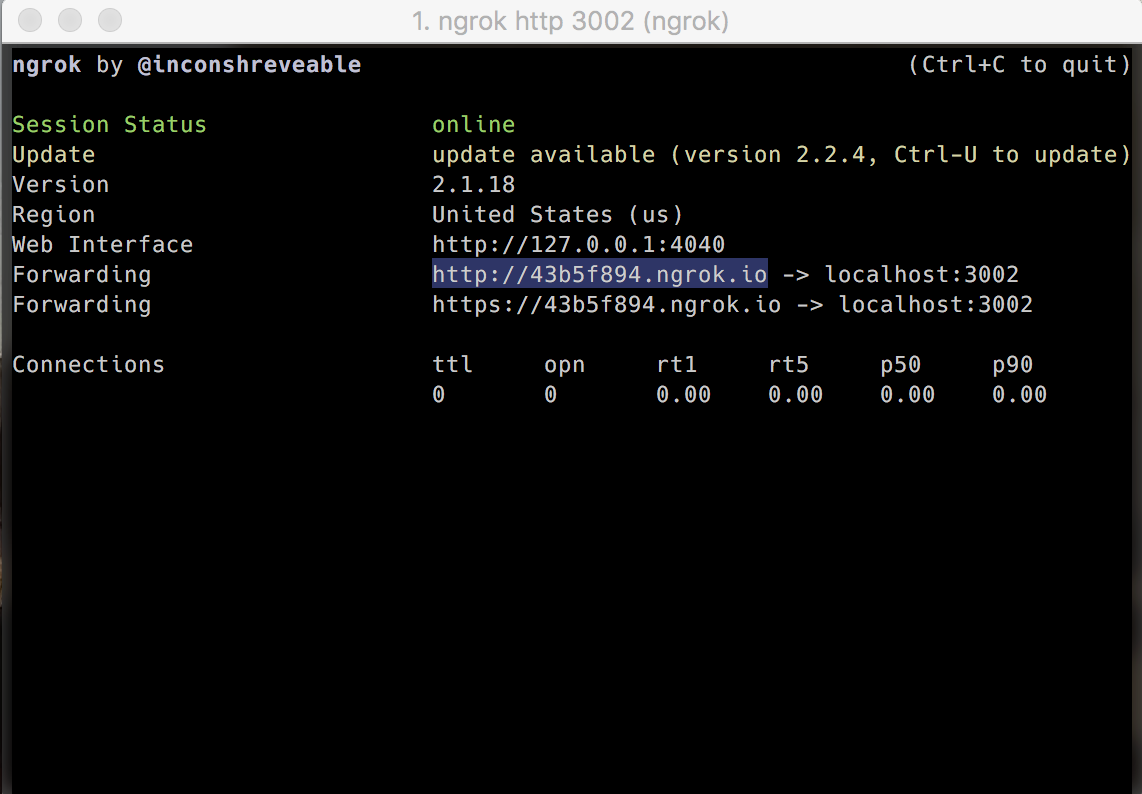
res.sendStatus(200);

});

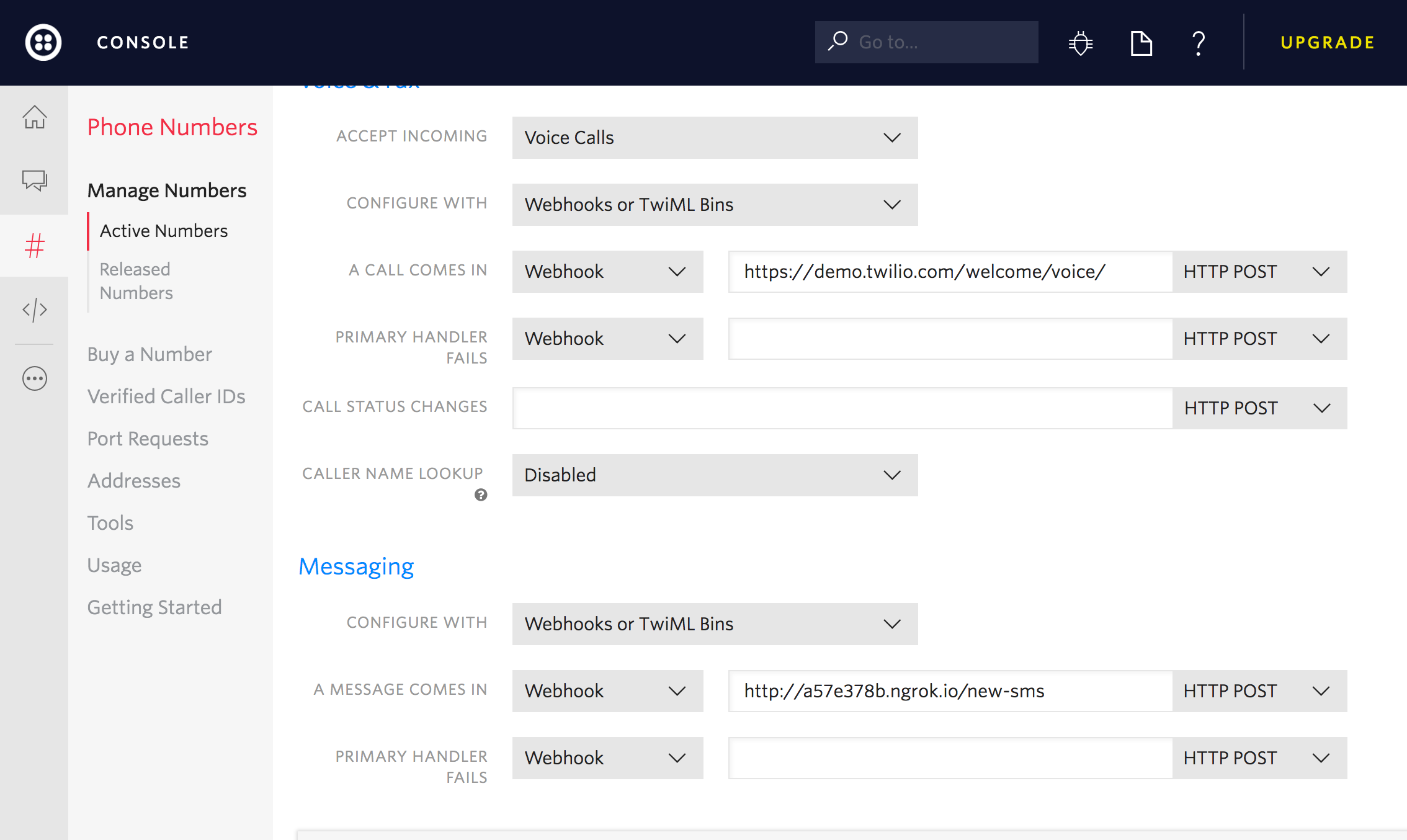
``` end ```

Next head over to the Twilio Console to configure your Twilio phone number with a Messaging webhook.

If you have a server that you can deploy the application to and get a public url - brilliant, otherwise [ngrok](https://ngrok.com/) is an amazing tool. It generates a unique public url for whatever port you expose your application on. Run `ngrok http 3002` in a separate terminal and keep it running. You will get an ngrok URL you can use as your our Twilio phone numbers Messaging webhook URL.



Once you have a publicly accessible URL, whether from a publicly deployed app or through ngrok, head over to the [Console](https://www.twilio.com/console) to configure your phone number.



Scroll down to the Messaging section and in the field ‘A Message comes in’ , add the ngrok url from the last step.

Now run the server with the command `yarn dev:server` and then try sending a new SMS to your Twilio Number. Voila! The message gets printed to your terminal

### New Email Flow

There can be multiple ways to setup a new email workflow, but here’s this short flow:



Here, the user starts with sending the Twilio number a message ‘NEW’, the express server prompts for the next step after each user step with the syntax of the next step. The first step is email address, which the user sends as ‘[a@xy.com](mailto:a@xy.com)’.hen the user sends the subject as ‘Message Subject ‘, and finally with the message body which the user sends as ‘Message body!’ , the server then goes ahead and sends the compiled email.

Let’s setup our basic objects and the helper functions that we will need for the above flow in `server.js`.

```javascript```

//Note that only one person should use this at a time due to the nature of the global object

let incomingEmail = {state: null, nextState:"init", obj: {}};  
  
function updateEmailState(state=null, nextState=null) {  
 incomingEmail.state = state;  
 incomingEmail.nextState = nextState;  
}  
  
function addNewEmailObjectProperty(key=null, value=null) {  
 if(key && value)  
 incomingEmail.obj[key] = value;  
 }

function resetEmailObject() {

incomingEmail.obj = {};

}

```end```

What we do above is declare a global Email Object that is used in the latter functions.

Add this code to parse the email and manage the flow in `server.js`.

```javascript```

function parseToEmail(smsBody) {  
 const toQuery = 'Recipient Email address?';  
 const subjectQuery = 'Subject?';  
 const messageQuery = 'Message? (Under 1600 characs)';  
  
 smsBody = smsBody.trim();  
 const isNew = smsBody.toLowerCase() === 'new';  
 let nextSMS = '';  
  
 if (isNew) {  
 if (incomingEmail.state)  
 nextSMS = 'Existing email deleted. Starting again\n';  
 updateEmailState('init', 'eid');  
 nextSMS += toQuery;  
 } else {  
 switch (incomingEmail.nextState) {  
 case 'eid':  
 if (validator.isEmail(smsBody)) {  
 updateEmailState('eid', 'sub');  
 addNewEmailObjectProperty('to', smsBody);  
 nextSMS = subjectQuery;  
 } else {  
 nextSMS = 'Invalid Email Address! Try Again:..';  
 }  
 break;  
 case 'sub':  
 updateEmailState('sub', 'msg');  
 addNewEmailObjectProperty('sub', smsBody);  
 nextSMS = messageQuery;  
 break;  
 case 'msg':  
 updateEmailState('msg', 'end');  
 addNewEmailObjectProperty('msg', smsBody);  
 nextSMS = sendEmail();  
 break;  
 default:  
 nextSMS = 'Would you like to send a new email? Start by texting in "new".';  
 break;  
 }  
 }  
 sendSMS(nextSMS);  
}

```end```

So what happened above? The above function uses a global `incomingEmail` object to maintain which step of the new email creation the user is in, handling everything from notifying the user of the next step of the process by sending them an SMS Reply with the necessary syntax and an example, handling errors, validating the recipient email using [validator](https://github.com/chriso/validator.js) which is a nifty library for string validation and then finally sending the email.

### Sending Email using Nodemailer

You will notice at the end of the switch-case in the `*parseToEmail`* function we make a call to `*sendEmail`* function which doesn’t exist yet.

Now let’s write that function:

``` server.js ```

let transporter = nodemailer.createTransport({

service: 'gmail',

auth: {

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

pass: **process**.**env**.IMAP\_PASS

}

});

function sendEmail() {

//Code to send Email and onSuccess return String and clear out global incomingEmail

const mailOptions = {

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

to: incomingEmail.obj.to,

subject: incomingEmail.obj.sub,

text: incomingEmail.obj.msg,

};

transporter.sendMail(mailOptions, (*err*, *info*) => {

if (err) {

**console**.**log**(err);

sendSMS("Email Sending Failed! Try Again..")

}

});

//Empty out the global mail object

updateEmailState();

resetEmailObject();

return 'Email Sent!';

}

``` end ```

Couple of things about this piece of code - we create a transporter object using the [nodemailer](https://github.com/nodemailer/nodemailer)library that allows us to send emails from a NodeJS Application - easy peasy! Here I am using gmail withthe same address to send emails as the one I used to watch for emails in the [previous article](https://www.twilio.com/blog/2017/06/how-to-forward-emails-sms-node-js.html).I'm also using the gmail preset the library provides, however if you want to use any other service, check the [documentation to see how you can set it up.](https://nodemailer.com/usage/)

The `*sendEmail` function* handles setting up the email and populating it with the required properties from our `*incomingEmail`* object and uses the `*transporter`* object to send it. The function returns a success string that the server sends us as an SMS to notify the Email has been sent.

### Putting it all together

We need to make just a slight change to our `/new-sms` handler in `server.js`:

```javascript```

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

console.log(req.body.body);

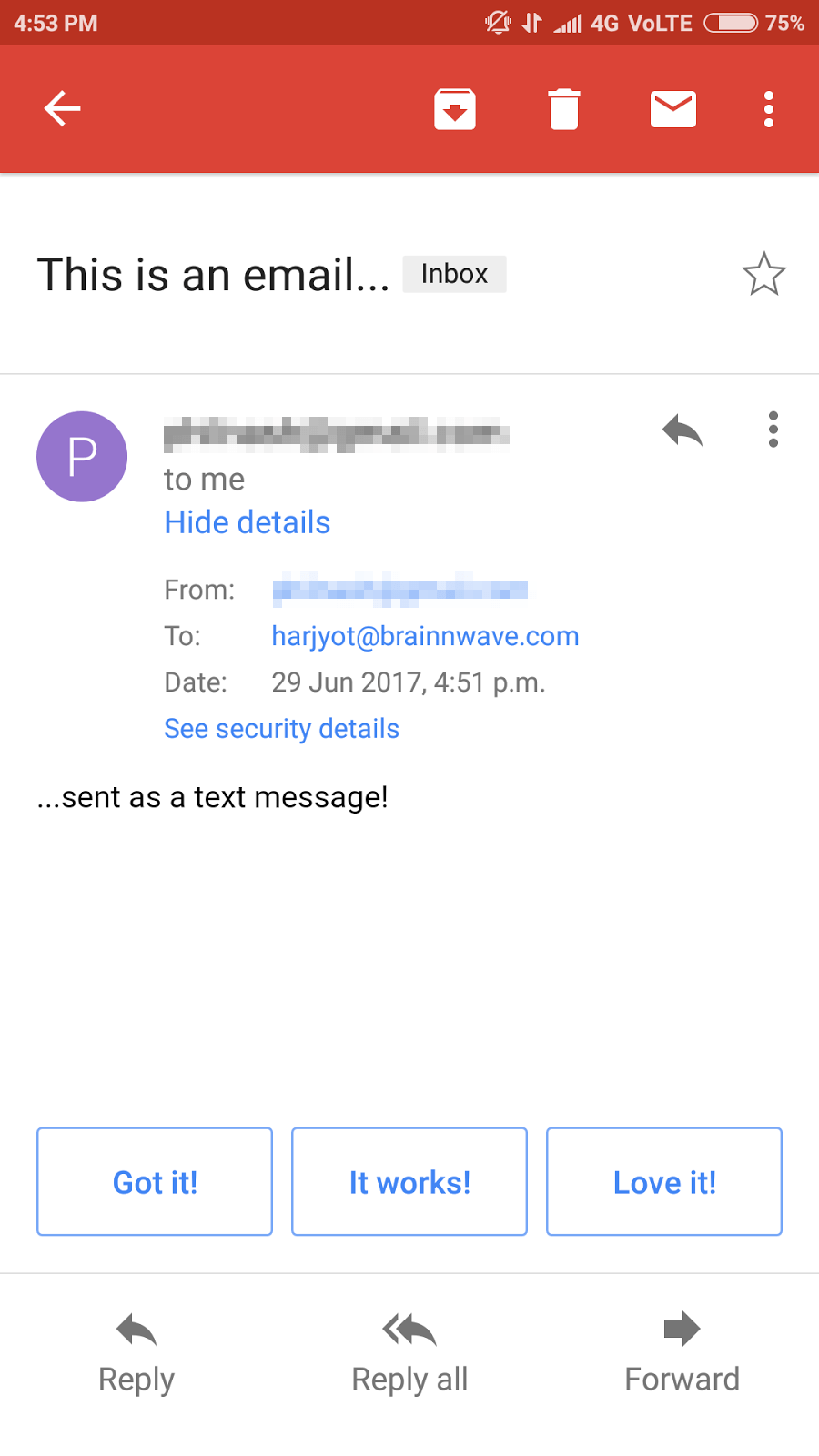
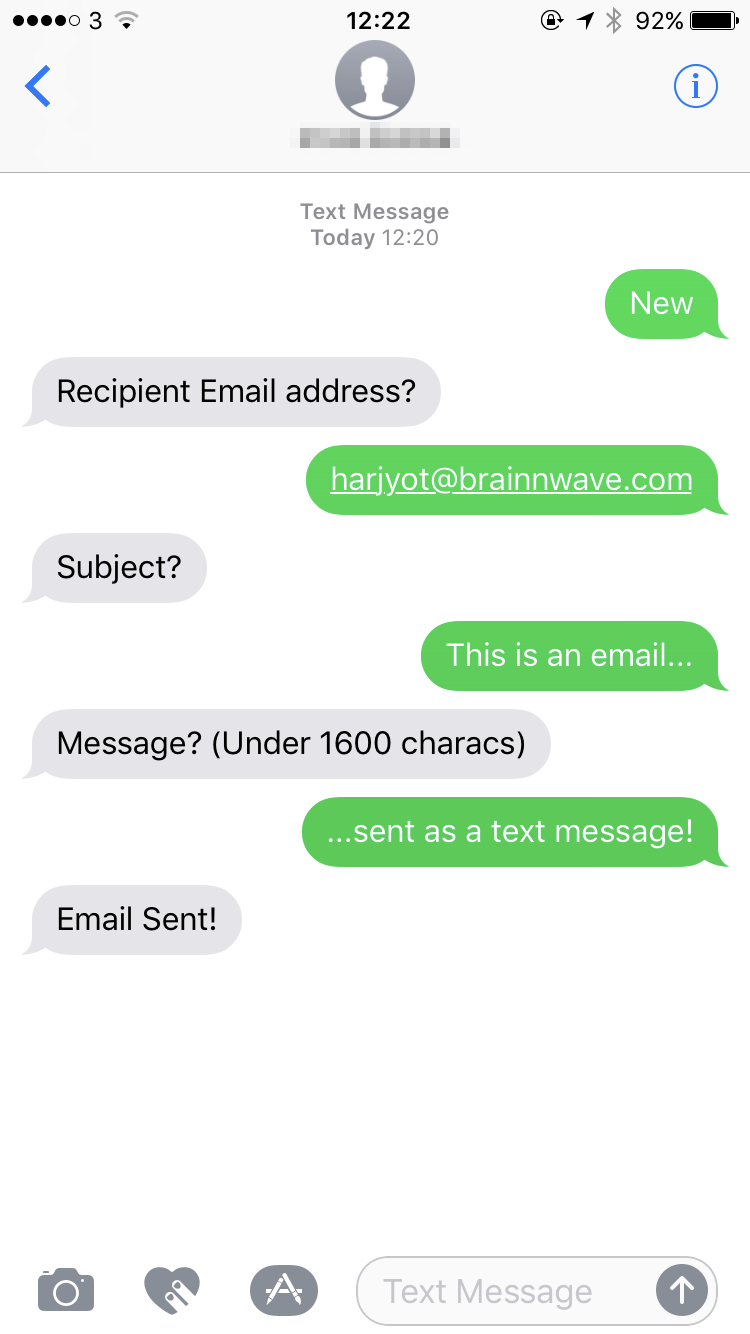
parseToEmail(req.body.Body);

res.send(‘<Response/>’);

});

```end```

Shall we try it? Send ‘New’ to your Twilio Number via SMS and follow the replied to get back!



### Final Thoughts

In this short tutorial we extended our existing application by adding webhooks, using [ngrok](https://ngrok.com/) to test an app without deploying it, using [nodemailer](https://github.com/nodemailer/nodemailer) to send emails from a NodeJS Application and chalking out a simple and effective state transitioning algorithm to solve a practical problem. So many new functionalities were added on.

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.

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).