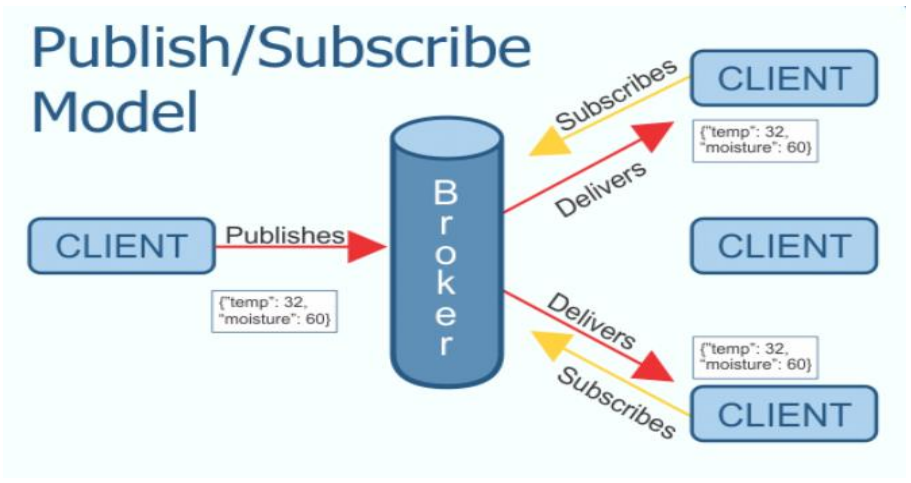
Wireless Networks

A Project by

Soham Dave - 60003198002
Ansh Mehta - 60003198003
Shubham Patil - 60003198004

1. Literature on Publisher-Subscriber Model using MQTT connection

The publish/subscribe pattern (also known as pub/sub) provides an alternative to a traditional client-server architecture. In the client-server model, a client communicates directly with an endpoint. The pub/sub model decouples the client that sends a message (the publisher) from the client or clients that receive the messages (the subscribers). The publishers and subscribers never contact each other directly. In fact, they are not even aware that the other exists. The connection between them is handled by a third component (the broker). The job of the broker is to filter all incoming messages and distribute them correctly to subscribers.
The publish-subscribe model is different from the traditional client-server model. It separates the client (publisher) that sends the message from the client (subscriber) that receives the message. The publisher and the subscriber do not need to establish direct contact. We can either let multiple publishers publish messages to one subscriber, or let multiple subscribers receive messages from one publisher at the same time. The essence of it is that an intermediate role called a broker is responsible for all message routing and distribution. The traditional client-server model can achieve similar results, but it cannot be as simple and elegant as the publish-subscribe model.



The most important aspect of pub/sub is the decoupling of the publisher of the message from the recipient (subscriber). This decoupling has several dimensions:

- Space decoupling: Publisher and subscriber do not need to know each other (for example, no exchange of IP address and port).
- Time decoupling: Publisher and subscriber do not need to run at the same time.
- Synchronization decoupling: Operations on both components do not need to be interrupted during publishing or receiving.

In summary, the pub/sub model removes direct communication between the publisher of the message and the recipient/subscriber. The filtering activity of the broker makes it possible to control which client/subscriber receives which message. The decoupling has three dimensions: space, time, and synchronization.

Message routing

As the key role of the publish-subscribe model, the broker needs to accurately and efficiently forward the desired messages to the subscribers. Generally speaking, there are two methods:

- Based on the topic. Subscribers subscribe to topics they are interested in from the MQTT broker. All messages published by the publisher will include their topics. The broker determines which subscribers need to be forwarded to the message according to the topic of the message.
- Based on message content. The subscriber defines the conditions of the message that they are interested in. Only when the attributes or content of the message meet the conditions defined by the subscriber, the message will be published to the subscriber. Strictly speaking, the topic can also be regarded as a kind of message content.

The loosely-coupled nature of the publish-subscribe model also has some side effects. Since the publisher is not aware of the subscriber's status, the publisher cannot know whether the subscriber has received the message or whether the message has been processed correctly. In such cases, securing delivery often requires a more interactive flow of messages. For example, a subscriber sends a response to a topic after receiving a message, and the publisher is now a subscriber waiting for a response.

MQTT protocol

The MQTT protocol distributes messages based on topics rather than message content. Each message contains a topic, and the broker does not need to parse user data. This provides the possibility to implement a general, business-independent MQTT broker. Users can also encrypt their data at will, which is very useful for WAN communication.

MQTT topics can have multiple levels, and allow fuzzy matching of one or more levels, enabling clients to subscribe to multiple topics at once. We will introduce the detailed features of the MQTT topic in the following articles.

Since we have mentioned message queues, it is time to explain the difference between MQTT and message queues. MQTT is not a message queue, although many behaviors and characteristics of the two are very close, such as using a publish-subscribe model. The scenarios they face are significantly different. Message queues are mainly used for message storage and forwarding between server-side applications. In this kind of scenario, the data volume is often large but the access volume is small. MQTT is targeted at the IoT field and the mobile Internet field. The focus of such scenarios is massive device access, management, and messaging. In practical scenarios, the two are often used in combination. For example, MQTT Broker first receives data uploaded by IoT devices, and then forwards these data to specific applications for processing through message queues.

MQTT uses subject-based filtering of messages. Every message contains a topic (subject) that the broker can use to determine whether a subscribing client gets the message or not. To handle the challenges of a pub/sub system, MQTT has three Quality of Service (QoS) levels. You can easily specify that a message gets successfully delivered from the client to the broker or from the broker to a client. However, there is the chance that nobody subscribes to the particular topic. If this is a problem, the broker must know how to handle the situation.

2. Methodology:

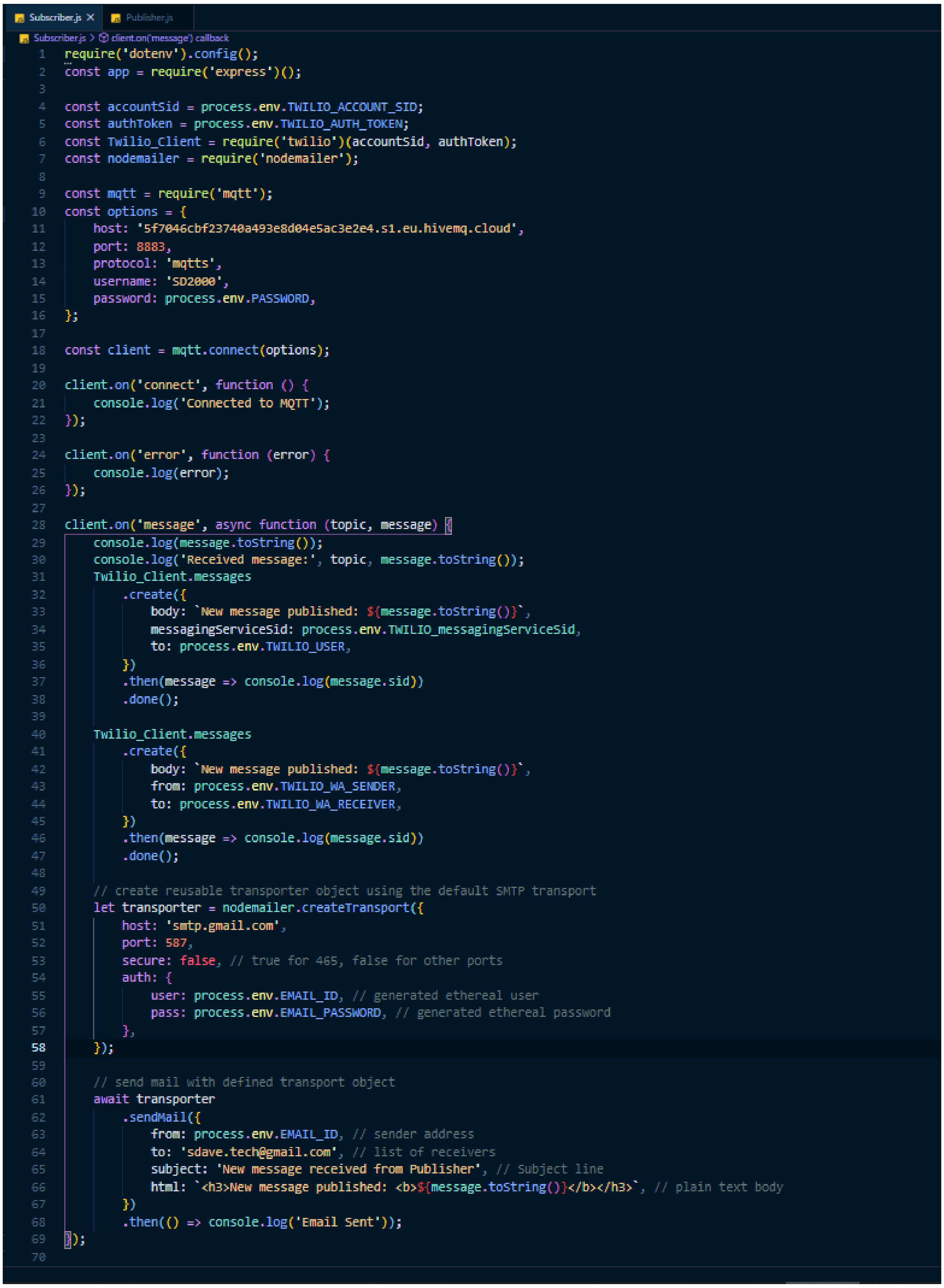
Introduction: The project successfully implements a publisher-subscriber communication model using the MQTT protocol. The project implemented makes use of 2 servers that act as the subscriber and publisher. The end goal was for the publisher to send an email to the subscriber along with a confirmatory message signifying a successful MQTT connection.

Working:

- The client i.e. the publisher establishes an MQTT channel and gets himself connected.
- After verifying the publisher's credentials, the subscriber on the other end establishes his connection to the channel.
- The publisher proceeds to send a message and an email to the subscriber, which is then received by the subscriber on the email-id mentioned.
- All the communication takes place over the MQTT channel of which the publisher and the subscriber are a part.

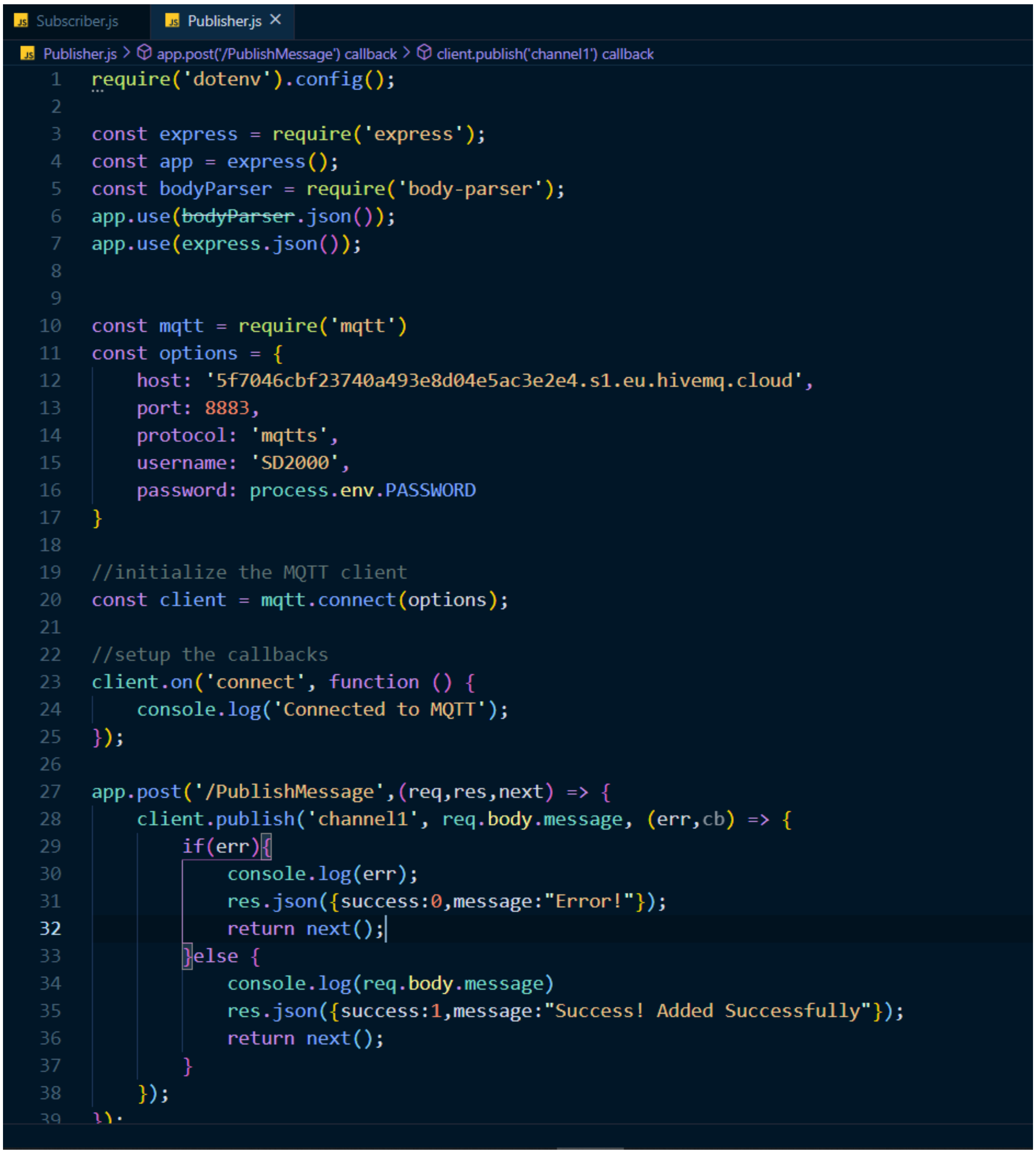
Code snippets:

- Code on the client-side i.e. the publisher who sends the message to the subscriber along with the SMS, WhatsApp Message and an email.



```javascript
require('dotenv').config();
const app = require('express')();

const accountSid = process.env.TWILIO_ACCOUNT_SID;
const authToken = process.env.TWILIO_AUTH_TOKEN;
const Twilio_Client = require('twilio')(accountSid, authToken);
const nodemailer = require('nodemailer');

const mqtt = require('mqtt');
const options = {
    host: '5f7046cbf23740a493e8d04e5ac3e2e4.s1.eu.hivemq.cloud',
    port: 8883,
    protocol: 'mqtts',
    username: 'SD2000',
    password: process.env.PASSWORD,
};

const client = mqtt.connect(options);

client.on('connect', function () {
    console.log('Connected to MQTT');
});

client.on('error', function (error) {
    console.log(error);
});

client.on('message', async function (topic, message) {
    console.log(message.toString());
    console.log('Received message:', topic, message.toString());
    Twilio_Client.messages
        .create({
            body: `New message published: ${message.toString()}`,
            messagingServiceSid: process.env.TWILIO_messagingServiceSid,
            to: process.env.TWILIO_USER,
        })
        .then(message => console.log(message.sid))
        .done();

    Twilio_Client.messages
        .create({
            body: `New message published: ${message.toString()}`,
            from: process.env.TWILIO_WA_SENDER,
            to: process.env.TWILIO_WA_RECEIVER,
        })
        .then(message => console.log(message.sid))
        .done();

    // create reusable transporter object using the default SMTP transport
    let transporter = nodemailer.createTransport({
        host: 'smtp.gmail.com',
        port: 587,
        secure: false, // true for 465, false for other ports
        auth: {
            user: process.env.EMAIL_ID, // generated ethereal user
            pass: process.env.EMAIL_PASSWORD, // generated ethereal password
        },
    });

    // send mail with defined transport object
    await transporter
        .sendMail({
            from: process.env.EMAIL_ID, // sender address
            to: 'sdave.tech@gmail.com', // list of receivers
            subject: 'New message received from Publisher', // Subject line
            html: `<h3>New message published: <b>${message.toString()}</b></h3>`, // plain text body
        })
        .then(() => console.log('Email Sent'));
});
```

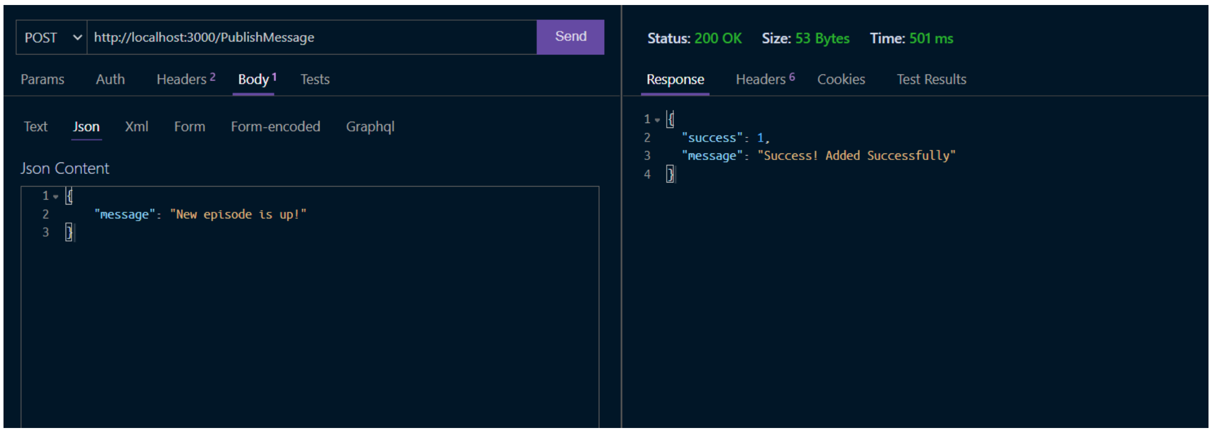
- Code to connect the subscriber to the channel via the MQTT protocol. The client will send an SMS, a WhatsApp message, and an email to the subscriber via the channel and will get a confirmatory message.


Subscriber.js
Publisher.js
Publisher.js > app.post('/PublishMessage') callback > client.publish('channel1') callback

```
require('dotenv').config();

const express = require('express');
const app = express();
const bodyParser = require('body-parser');
app.use(~~bodyParser~~.json());
app.use(express.json());


const mqtt = require('mqtt')
const options = {
    host: '5f7046cbf23740a493e8d04e5ac3e2e4.s1.eu.hivemq.cloud',
    port: 8883,
    protocol: 'mqtts',
    username: 'SD2000',
    password: process.env.PASSWORD
}

//initialize the MQTT client
const client = mqtt.connect(options);

//setup the callbacks
client.on('connect', function () {
    console.log('Connected to MQTT');
});

app.post('/PublishMessage',(req,res,next) => {
    client.publish('channel1', req.body.message, (err,cb) => {
        if(err){
            console.log(err);
            res.json({success:0,message:"Error!"});
            return next();
        }else {
            console.log(req.body.message)
            res.json({success:1,message:"Success! Added Successfully"});
            return next();
        }
    });
});
```

Output/Result snippets:

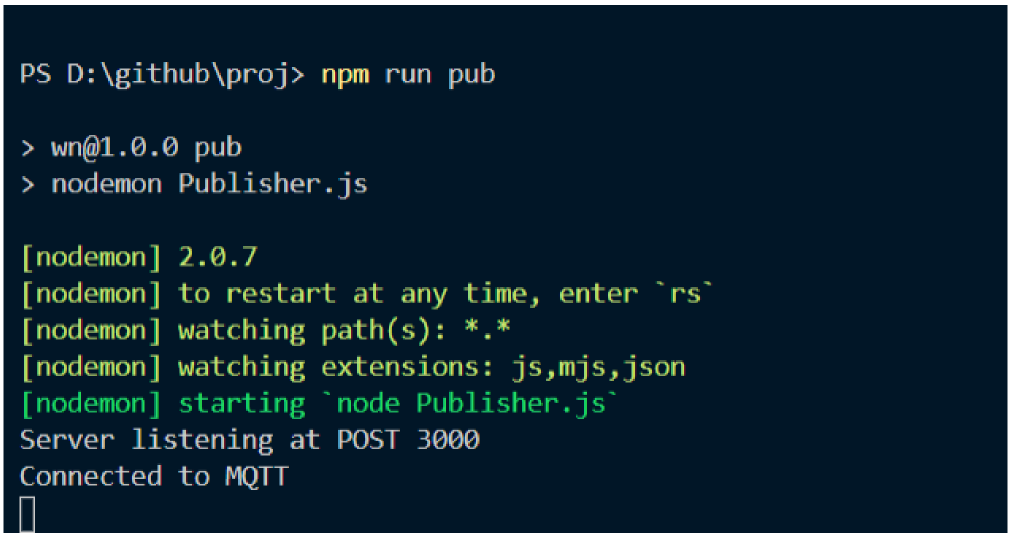
API created on Thunder Client, sending a message in the request body:



```
POST  http://localhost:3000/PublishMessage  Send
Params  Auth  Headers 2  Body 1  Tests
Text  Json  Xml  Form  Form-encoded  Graphql
Json Content
{
    "message": "New episode is up!"
}
```

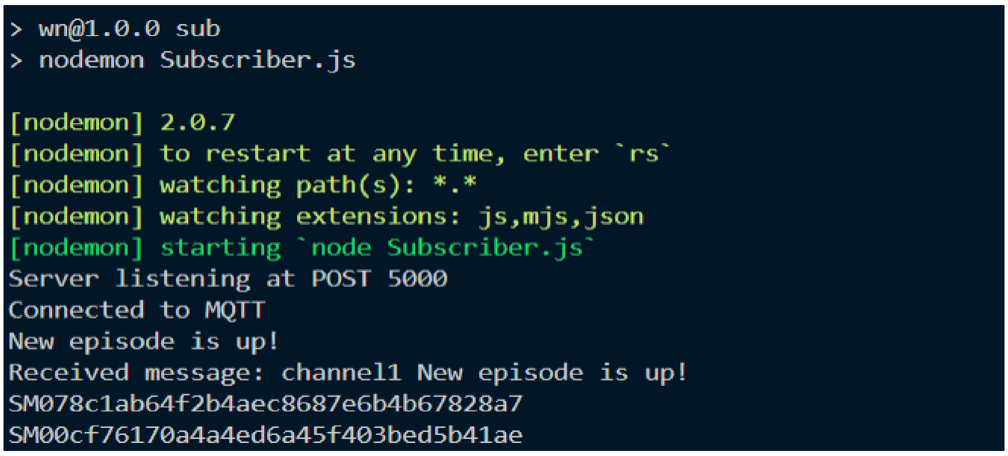
```
Status: 200 OK   Size: 53 Bytes   Time: 501 ms
Response  Headers 6  Cookies  Test Results
{
    "success": 1,
    "message": "Success! Added Successfully"
}
```

Starting the node application and connecting to the server:



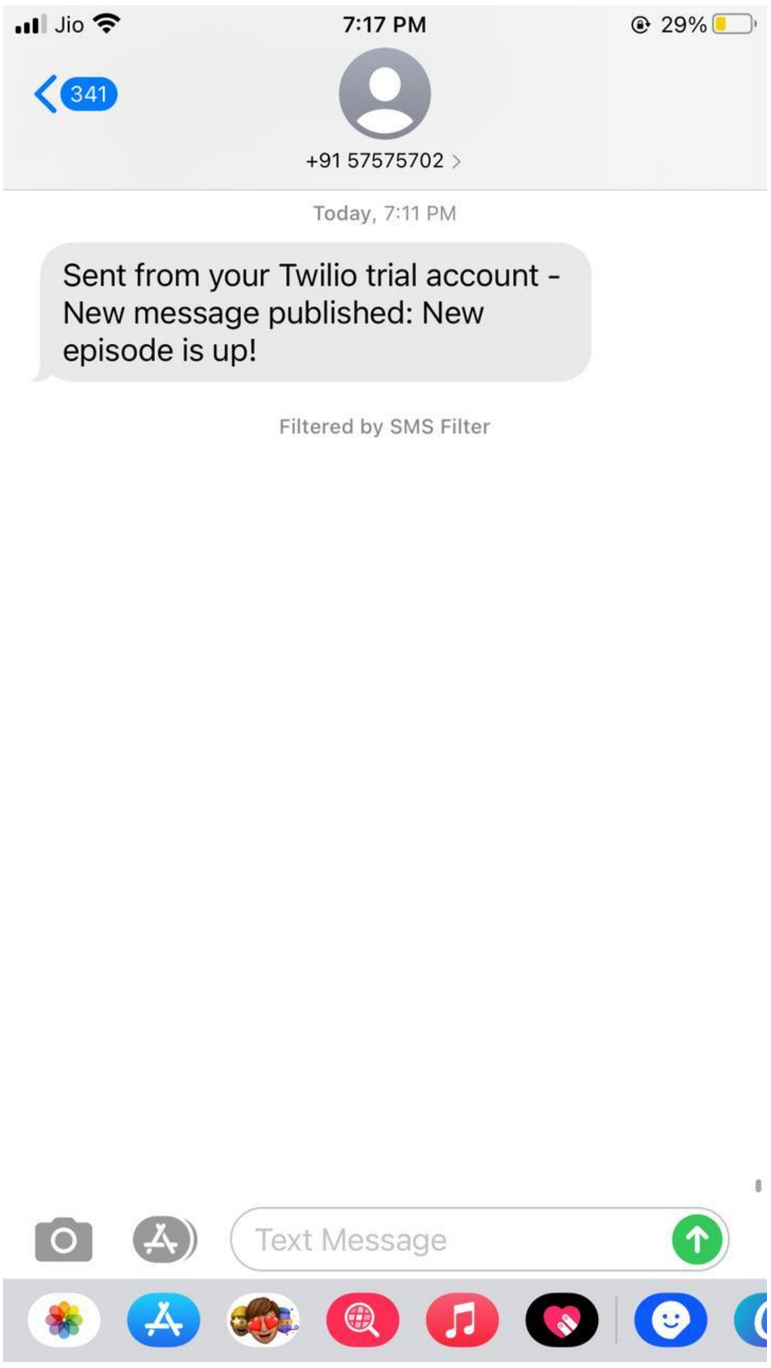
```
PS D:\github\proj> npm run pub

> wn@1.0.0 pub
> nodemon Publisher.js

[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node Publisher.js`
Server listening at POST 3000
Connected to MQTT
```

After ensuring the connection is secure, sending a message to the subscriber along with an email, an SMS and a WhatsApp message:



```
> wn@1.0.0 sub
> nodemon Subscriber.js

[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node Subscriber.js`
Server listening at POST 5000
Connected to MQTT
New episode is up!
Received message: channel1 New episode is up!
SM078c1ab64f2b4aec8687e6b4b67828a7
SM00cf76170a4a4ed6a45f403bed5b41ae
```

Received SMS:



Jio 7:17 PM 29%

341

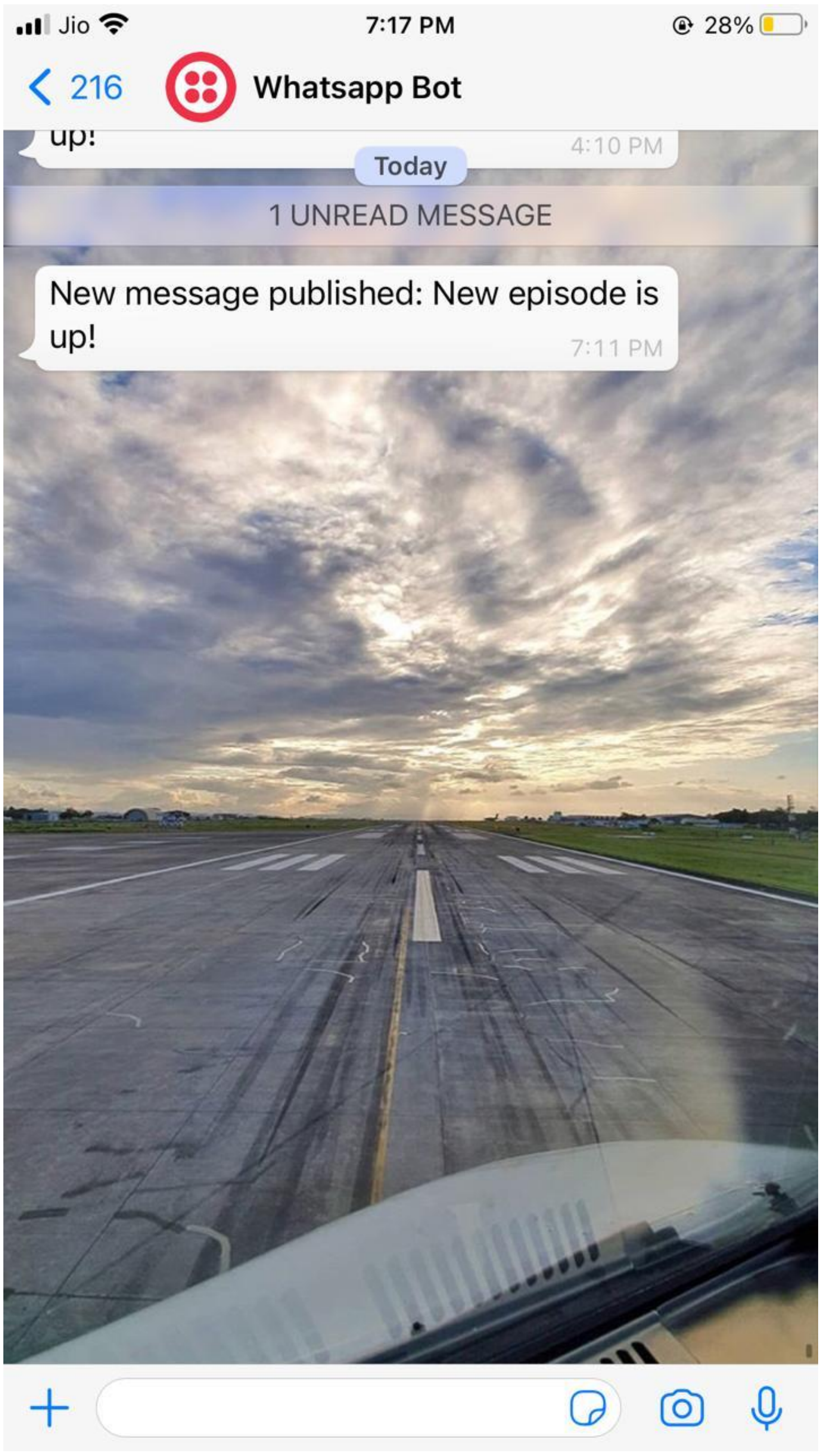
+91 57575702

Today, 7:11 PM

Sent from your Twilio trial account - New message published: New episode is up!

Filtered by SMS Filter

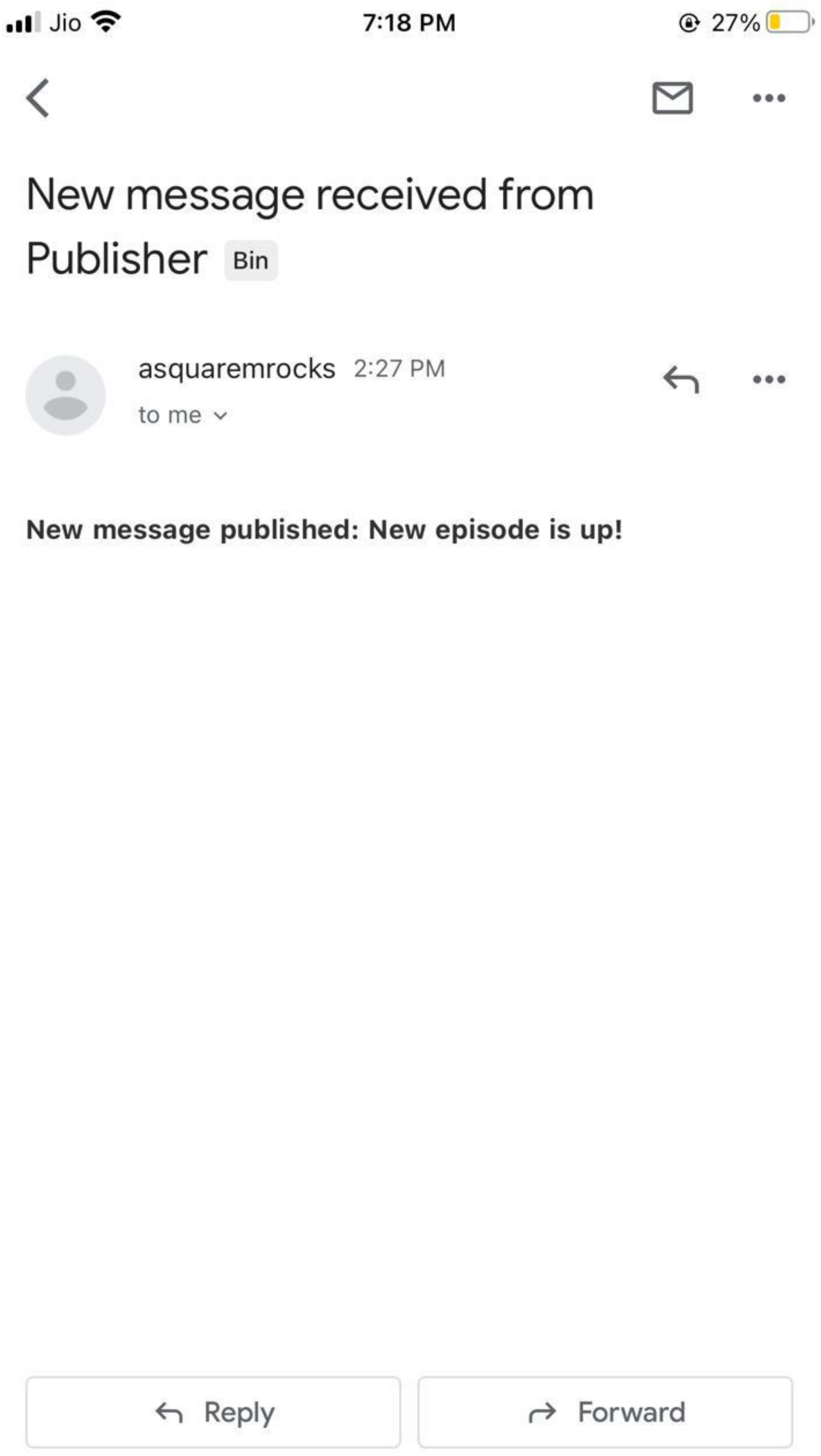
Received WhatsApp Message:



up!
4:10 PM

New message published: New episode is up!
7:11 PM

Received email:



New message received from Publisher

asquaremrocks 2:27 PM

to me

New message published: New episode is up!