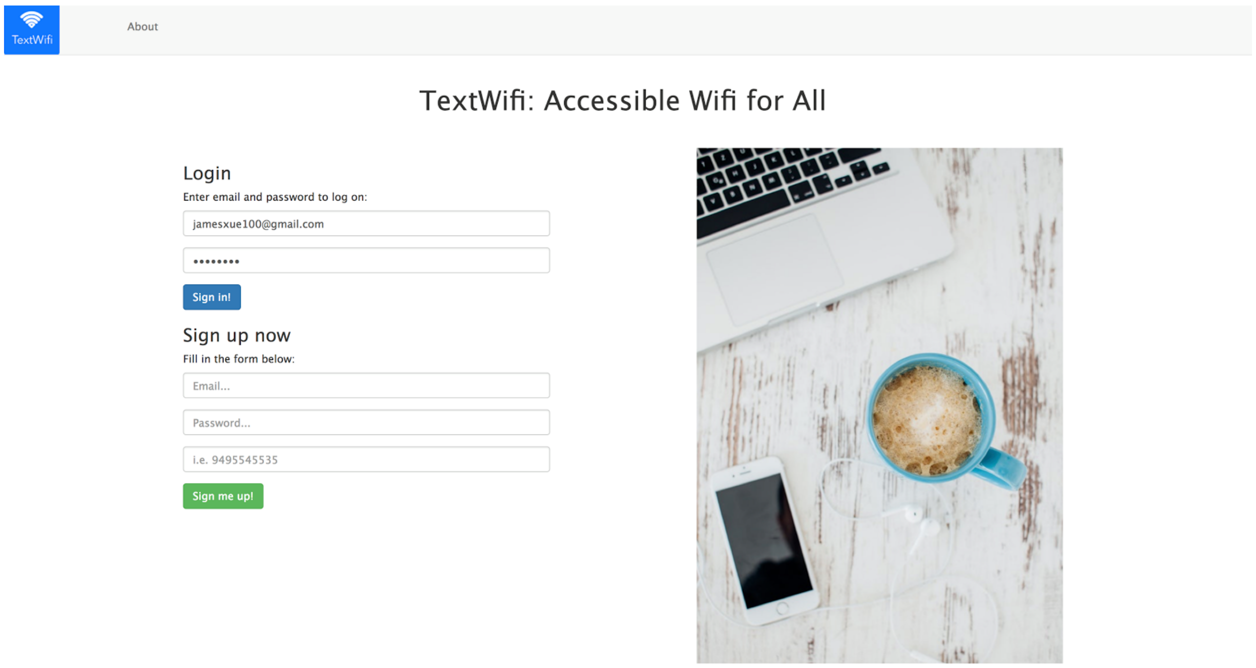
TextWifi: Project Report

Chan Pham (tqp2001) & James Xue (jx2218)



TextWifi: Accessible Wifi for All

Login

Enter email and password to log on:

jamesxue100@gmail.com

••••••••

Sign in!

Sign up now

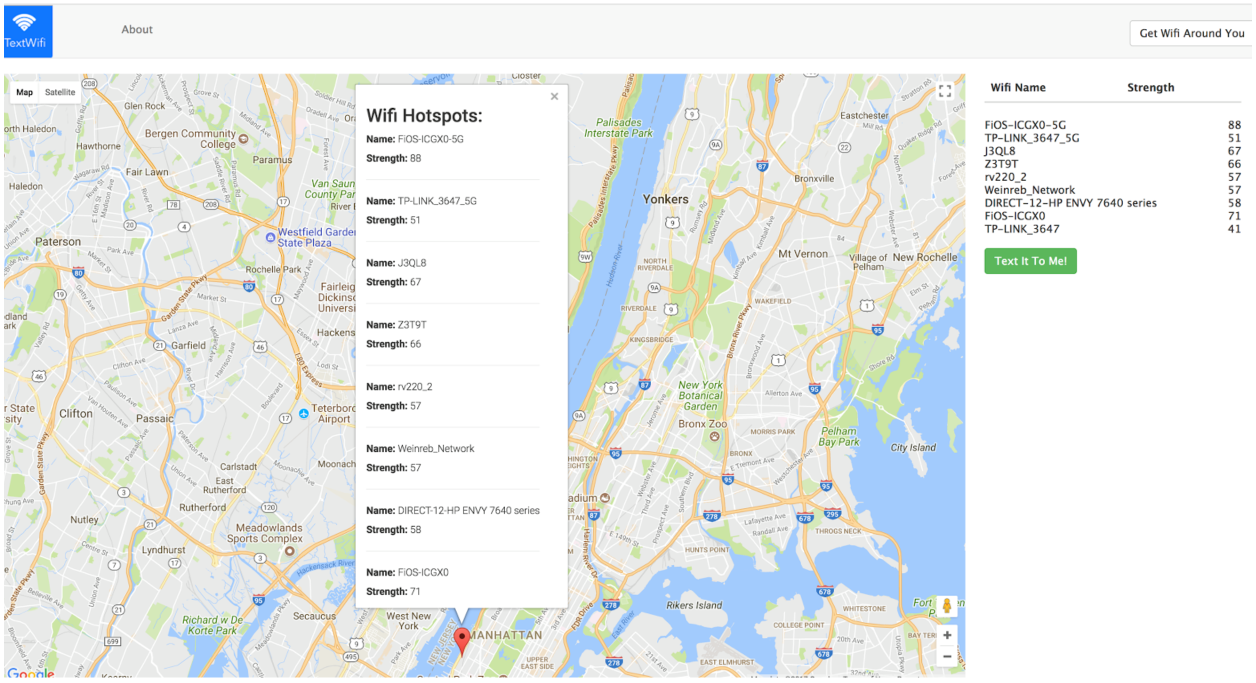
Fill in the form below:

Email...

Password...

i.e. 9495545535

Sign me up!



Wifi Hotspots:

Name: FiOS-ICGX0-5G
Strength: 88

Name: TP-LINK_3647_5G
Strength: 51

Name: J3QL8
Strength: 67

Name: Z3T9T
Strength: 66

Name: rv220_2
Strength: 57

Name: Weinreb_Network
Strength: 57

Name: DIRECT-12-HP ENVY 7640 series
Strength: 58

Name: FiOS-ICGX0
Strength: 71

Wifi Name	Strength
FiOS-ICGX0-5G	88
TP-LINK_3647_5G	51
J3QL8	67
Z3T9T	66
rv220_2	57
Weinreb_Network	57
DIRECT-12-HP ENVY 7640 series	58
FiOS-ICGX0	71
TP-LINK_3647	41

Text It To Me!

Table of Contents

Introduction

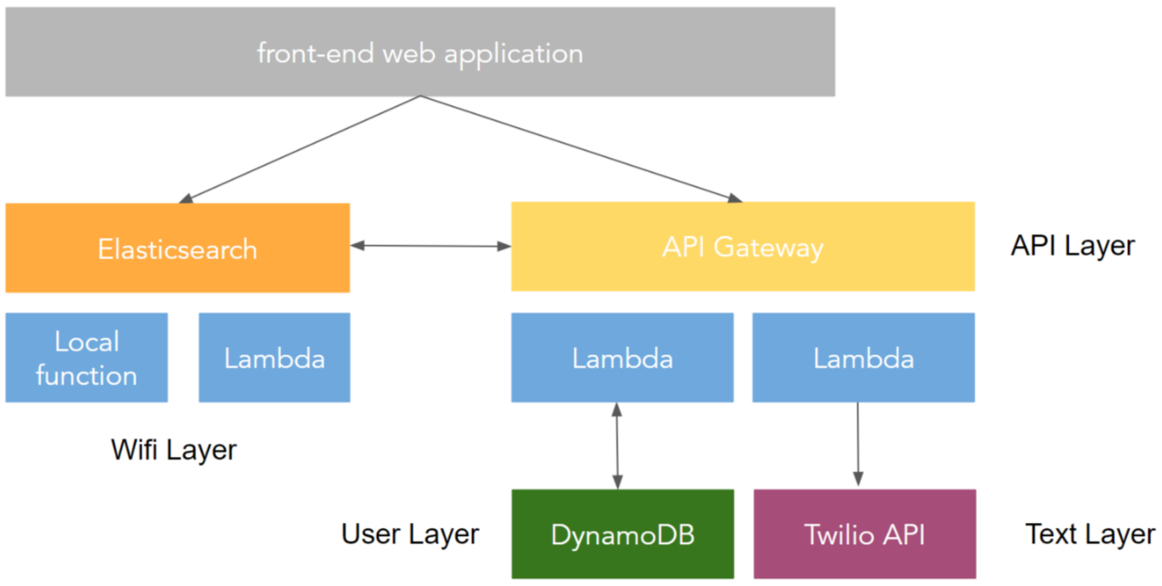
We all know how much overcharging of unlimited data plans and the Internet makes up for a significant cost of utilities, especially for those of us who use our phones 24/7. However, currently there is no great way of finding out easily accessible and free wifi hotspots around you, in a seamless and non-intrusive way. So we built TextWifi with the goal of providing accessible wifi for all.

Inspired by the sharing economy and the on-the-go lifestyle of New Yorkers + college students, TextWifi taps into previously untapped sources of wifi, such as public networks (e.g. LinkNYC, Downtown Brooklyn, MTA), as well as private guest networks (e.g. Starbucks, Columbia University) to provide a unique platform for connecting community and connecting to Internet.

The application itself is a web application that is seamless and non-intrusive. As a web application, it is accessible from both laptops and mobile devices. Our web application runs a script to identify available networks at user location (currently only supporting OSX Sierra). These networks are indexed and aggregated in our dynamically updating DynamoDB database. This same database is accessed to identify nearby hotspots on user queries. Our active + updating database reflects how based on user location, networks are updated to the most current and available networks, so you can rest assured that you're always getting the most up to date wifi networks around you.

With a visual and intuitive UI integrated with SMS support and Google maps display, it's easy to travel with TextWifi!

Architecture



- Front-end Web Application: our front-end application is written in Python and Flask, and utilizes the Google Maps API. The application supports user queries of locations by type (for example, coffee shops and LinkNYC data spots) as well as texting, login/logout/signup, and more.
- Wifi Layer: the Wifi layer consists of lambdas to post (wifiLocationPost.py) and query (wifiRadiusQuery.py) locations. Both lambdas interact with elasticsearch with documents indexed by name, wifi strength, latitude & longitude, and hotspot type (coffee, LinkNYC, user).
- API Layer: The API layer sets up the API Gateway to connect the lambdas to the front-end web application via /wifi and /user GET and POST end-points
- User Layer: The user layer manages authentication and storing user info (such as phone number) for user-based sessions
- Text Layer: The text layer utilizes Twilio API and user information to send the wifi locations to the user's phone for on-the-go information!

README: Lambda Functions

i. wifiLocationPost

```python
from elasticsearch import Elasticsearch
import certifi

def lambda_handler(event, context):
  host = 'https://search-textwifi-jhk6t4hsbrgwl4636zeyrabk74.us-east-1.es.amazonaws.com'
  es = Elasticsearch([host], ca_certs=certifi.where())
  loc = event[0]["location"]
  q = {
    "query": {
      "bool" : {
        "must" : {
          "match_all" : {}
        },
        "filter" : {
          "geo_distance" : {
            "distance" : "0.1km",
            "location" : loc
          }
        }
      }
    }
  }

  response = es.delete_by_query(index='wifi', doc_type='hotspot', body=q)
  print(response)
  for w in event:
    print(w)
    hotspot = {}
    hotspot["name"] = str(w["name"])
    hotspot["strength"] = int(w["strength"])
    hotspot["location"] = w["location"]
    hotspot["loc_type"] = str(w["loc_type"])
    response = es.index(index='wifi', doc_type='hotspot', body=hotspot)

  return response
```

Essentially, this function makes a POST request to our ElasticSearch instance to upload the hotspots near the user's current location, which is determined by our wifi location query script (see next section). We make sure to also delete the previous hotspots at the current location before uploading the new ones. We make sure to add the following fields to our hotspots stored in our ElasticSearch instance: name, strength, location, and location type.

ii. wifiRadiusQuery

```
from elasticsearch import Elasticsearch
import certifi
import json

def lambda_handler(event, context):
  print(event)
  host = 'https://search-textwifi-jhk6t4hsbrgwl4636zeyrabk74.us-east-1.es.amazonaws.com'
  es = Elasticsearch([host], ca_certs=certifi.where())
  #data_string = json.dumps(event["queryStringParameters"]["getDataString"])
  data_string = event["queryStringParameters"]["getDataString"].encode('utf-8')
  data = json.loads(data_string)
  loc = data["location"]
  loc_type = data["loc_type"]
  print(loc)
  print(loc_type)
  if loc_type:
    q = {
      "query": {
        "bool" : {
          "must" : {
            "match" : {"loc_type": loc_type}
          },
          "filter" : {
            "geo_distance" : {
              "distance" : "5km",
              "location" : loc
            }
          }
        }
      }
    }
  else:
    q = {
      "query": {
        "bool" : {
          "must" : {
            "match_all" : {}
          },
          "filter" : {
            "geo_distance" : {
              "distance" : "2km",
              "location" : loc
            }
          }
        }
      }
    }

  response = es.search(index="wifi", doc_type="hotspot", body=q)
  print(response)
  json_response = {
    "statusCode": 200,
    "headers":{
        "Access-Control-Allow-Origin": "*"
    },
    "body": json.dumps(response)
  }
  return json_response
```

This function makes a GET request to our ElasticSearch instance to retrieve the hotspots, based on where the user clicked on the map (passed into the event object). We make sure to filter by location and only get those results that matched the location_type, which could be a coffee shop (Starbucks), LinkNYC hotspot, or a user-generated hotspot. Afterwards, a successful response is returned (200) if it is successful.

iii. wifiUserCheck

```python
import boto3
import json

def lambda_handler(event, context):
  dynamo = boto3.client('dynamodb')
  print(event)

  data_string = event["queryStringParameters"]["getDataString"].encode('utf-8')
  data = json.loads(data_string)
  print(data)
  print(type(data))

  response = dynamo.get_item(
    TableName='wifiUsers',
    Key={
      'email': {
        'S': data["email"]
      }
    }
  )
  print(response)

  if not "Item" in response:
    json_response = {
      "statusCode": 404,
      "headers": {
        "Access-Control-Allow-Origin": "*"
      },
      "body": "Email not found"
    }
  else:
    password = str(hash(data["password"]))
    if password == response["Item"]["password"]["S"]:

      json_response = {
        "statusCode": 200,
        "headers": {
          "Access-Control-Allow-Origin": "*"
        },
        "body": str(json.dumps(response))
      }
    else:
      json_response = {
        "statusCode": 401,
        "headers": {
          "Access-Control-Allow-Origin": "*"
        },
        "body": "Incorrect password"
      }
  print(json_response)
  return json_response
```

This function is called whenever a user signs into the app. The user email is looked up in the DynamoDB application and if there is a hit, it is returned in the JSON response as a 200. Otherwise, an error is thrown. These users are stored in our DynamoDB table called wifiUsers.

iv. wifiUserUpload

```
import boto3
import json

def lambda_handler(event, context):
  dynamo = boto3.client('dynamodb')

  print(event["body"])
  print(type(event["body"]))
  data_string = event["body"].encode('utf-8')
  data = json.loads(data_string)
  print(data)
  print(type(data))
  response = dynamo.get_item(
    TableName='wifiUsers',
    Key={
      'email': {
        'S': data["email"],
      }
    }
  )

  if not "Item" in response:
    email = data["email"]
    password = str(hash(data["password"]))
    phone = data["phone"]
    response = dynamo.put_item(
      TableName='wifiUsers',
      Item={
        'email':{'S':email},
        'password':{'S':password},
        'phone':{'S':phone}
      }
    )

  json_response = {
    "statusCode": 200,
    "headers": {
      "Access-Control-Allow-Origin": "*"
    },
    "body": str(json.dumps(response))
  }
  print(response)
  return json_response
```

This function makes a POST request to upload user information to our DynamoDB table called wifiUsers. We store the email, password, and phone number to enable easy SMS functionality and signup functionality. This function is called whenever a new user signs up to the application. We also make sure to hash each password to keep our application secure. Afterwards, we return a successful JSON response (200).

v. textData

```
from twilio.rest import Client
import json
```

```
def lambda_handler(event, context):
    print(event)
    print(context)

    print(event["body"])
    print(type(event["body"]))
    data_string = event["body"].encode('utf-8')
    print(data_string)
    data = json.loads(data_string)
    print(data)
    print(type(data))

    phone = "(949)554-5535"
    wifi_string = "Here are your wifi hotspots!\n"

    print(data["items"])
    for i in range(0, len(data["items"])):
      wifi = data["items"][i]
      print(wifi)
      if "name" in wifi:
        wifi_name = wifi["name"]
        wifi_string += (str(wifi_name) + "\n")
      else:
        print("phone changed")
        phone = str(wifi["phone"])


    with open('twilio_auth.txt') as f:
      lines = f.read().splitlines()

    account_sid = lines[0]
    auth_token = lines[1]

    client = Client(account_sid, auth_token)

    #message to client
    message = client.messages.create(to=phone, from_="(509)774-2976",
        body=wifi_string)
    response = {
        "statusCode": 200,
        "headers": {
        "Access-Control-Allow-Origin": "*"
        },
        "body": '{"success": "success"}'
    }

    return response
    # return 'Hello from Lambda'
```

This function is called whenever the user wishes to receive a text message of wifi hotspots from the application. It calls the Twilio API and fetches the phone number of the user (stored in a query parameter) and then calls the API with a formatted message of the wifi hotspots. A successful response (200) is returned.

README: Web Application Code

i. app.py

```
import json
from flask import Flask, render_template
from wifi_positioning_system import getWifi
import time

app = Flask(__name__)
app.config.update(
  DEBUG=True,
  SECRET_KEY='TWITTMAP'
)

'''
Loads webpage
'''
@app.route('/', methods=['GET', 'POST'])
def index():
  return render_template('index.html')

'''
Loads webpage
'''
@app.route('/login', methods=['GET', 'POST'])
def login():
  return render_template('login.html')

'''
Returns wifi around a given user's location
'''
@app.route('/generateWifi', methods=['GET', 'POST'])
def generateWifi():
  result = getWifi()
  return json.dumps(result)

if __name__ == '__main__':
  app.run()
```

This just shows the basic setup for our Flask infrastructure and internal API for generating wifi. Here is how our endoints ('/' and '/login') are hit, and the HTMLs are rendered. The script that generates wifi can be found in our code submission on Github.

ii. script.js

This file can be found in our code repository on Github, as it is too long to paste here. It supports all of our Google Maps functionality, and calls all of our lambda functions shown above. It is JavaScript code and uses jQuery as well as Twitter Bootstrap and the Google Maps API.

For example, here is an example of a typical GET request to our Lambda function:

```
getDataString = `{"loc_type": "` + checkedCategory + `", "location": {"lat": "` + lat + `", `
+ `"lon": "` + lng + `"}}`;

    var getUrl = 'https://opyo6yseaa.execute-api.us-east-1.amazonaws.com/chan1/wifi'

    $.ajax({
      url: getUrl,
      type: 'GET',
      beforeSend: function (xhr) {
        xhr.setRequestHeader('Content-Type', 'application/json');
      },
      headers: headers,
      contentType: 'application/json',
      data: { getDataString: getDataString },
      success: function (response) {
        clearOverlays();

        var hits = response["hits"]["hits"];
        console.log(hits);
        var hits_dict = {};
        for (var i = 0; i < hits.length; i++) {
          hit = hits[i]["_source"];
          hit_strength = hit["strength"];
          hit_name = hit["name"];
          hit_location = JSON.stringify(hit["location"]);
          if (hits_dict[hit_location]) {
            hits_dict[hit_location].push(hit);
          } else {
            hits_dict[hit_location] = [hit];
          }
        }

        console.log(hits_dict);

        for (var index in hits_dict) {
          createMarker(JSON.parse(index), hits_dict[index]);
        }

        // createMarker(location, wifi_list);
        console.log(response);
      },
      error: function (response) {
        console.log(response);
      },
    })
```

README: Python Scripts
In addition to user-generated locations, we bootstrapped the databases with LinkNYC and Starbucks locations for additional public access points. To do this, we utilized the Socrata Open Data[1] for Starbucks locations worldwide and NYC Open Data[2] for all LinkNYC access points within the city. We parsed the databses and uploaded relevant info to Elasticsearch via two Python scripts (load_linkNYC.py and load_sbux.py both in the Github).

i. load_linkNYC.py segment

```
with open(sys.argv[1], 'rb') as f:
    freader = csv.reader(f)
    for row in freader:
        if row[6] == "Link Active!":
            hotspot = {}
            hotspot["name"] = row[0]
            hotspot["strength"] = "0"
            hotspot["location"] = {
                "lat": row[4],
                "lon": row[5]
            }
            hotspot["loc_type"] = "LinkNYC"
            print(hotspot)
            response = es.index(index='wifi', doc_type='hotspot',
                                body=hotspot)
            print(response)
        else:
            print(row[0] + " is not active")
```

[1] https://opendata.socrata.com/Business/All-Starbucks-Locations-in-the-World/xy4y-c4mk
https://opendata.socrata.com/Business/All-Starbucks-Locations-in-the-US/txu4-fsic
[2] https://data.cityofnewyork.us/Social-Services/LinkNYC-Locations/s4kf-3yrf/data

The Future of TextWifi

There are several additional features that, given the time, we would love to have implemented.

- heatmap/density tracking: with more users on the application, we could identify how many users are at a certain wifi location and so suggest less congested networks for optimal user experience
- similarly, with the user layer established, we could personalize suggestions based on user preferences over time (e.g. if user tends to frequent coffee shops vs libraries)
- with establishment and expansion of TextWifi, we could potentially reach out to vendors with private password covered networks and provide access to those for premium members, to guarantee stronger and more secure networks

Work Breakdown

Chan: Worked on most of the AWS side of the application, including writing the four core lambda functions for wifi GET and POST, and user GET and POST. Chan also set up the DynamoDB table as well as the ElasticSearch instance for our application to work. She also wrote the scripts for Starbucks and LinkNYC location bootstrapping of the Elasticsearch. Chan gave the introduction to the demo video.

James: Worked on most of the web application, including setting up the Python/Flask application structure and writing the Javascript code to communicate with the API gateway and lambda functions. Also wrote the text lambda function that communicates with the Twilio API. James went through the application demo in the demo video.

Thank you's

- Professor Sahu: We would like to thank Professor Sahu for guiding our vision of TextWifi from the hackathon and teaching us the tools needed to allow for its final development!
- Bindia & Dhruv: Thank you for all your help throughout the projects and encouraging our efforts with TextWifi when we almost dropped the idea. Without you, we would likely have pivoted and not made TextWifi a reality.