TEXTNET

Internet Over SMS

ELEC 490/498 Capstone Final Report

Akin Shonibare, Alex Everitt, Chris Maltais, Cory Crowley

Supervised by Professor Ying Zou

Table of Contents

Table of Figures

Executive Summary

What if anyone with an SMS enabled device could access the internet? Internet access has been declared a human right. The inspiration for TextNet came from the inflated mobile internet data rates in Canada and throughout North America. The primary goal of TextNet is to bridge the accessibility gap for mobile internet access that disadvantages individuals and deprives them of their rights, by creating an SMS based interface to the Internet which provides basic Internet functionality. Before starting the design process, the team laid out functional, interface, and performance requirement, and outlined the time and budget constraints.

In the design process, the first step was to lay out a set of design principles. The team concluded that TextNet should be modular, parallelizable, extensible, robust, easy to use, and device independent. The high-level design was then broken into four manageable components: connecting SMS devices to the server, scoping the internet functionality for the MVP, selecting appropriate internet APIs to fetch data, and parsing SMS messages to derive intent. The following functionality is supported by the TextNet MVP:

1. *Answer who, what, when, where, why questions*
2. *Provide step-by-step directions between locations*
3. *Recommend businesses based on location and business type.*
4. *Translate strings of text between languages.*
5. *Provide news updates*

The team used an agile development approach in implementing TextNet. The ZenHub project management extension for GitHub was used to outline a backlog of tasks and create user stories for the end product. Next, the basic Node.js infrastructure was set up. By week 9 of development, Twilio SMS functionality was implemented. By week 13, the external API providers were connected. By week 15, all core functionality was implemented. By week 17, user acceptance and unit testing were conducted.

Testing was extremely important for the project because functionality is directly dependent on user input, and users will not always interact with the application in an expected way. Testing and validation was implemented using *Codecov* for code coverage analysis and *Jest* as an assertion framework. The team achieved an 83% test coverage of the codebase. The controllers, where the main TextNet logic sits, achieved a 93% testing code coverage compared to the project's 83%.

In conclusion, the project met requirements and the team is extremely proud of the final product. TextNet is fast, flexible, robust, accurate, modular, and extensible! In future development, natural language message parsing could be implemented. The codebase could be refactored into a containerized microservice architecture. And lastly, more internet primitives can easily be added to increase the Internet functionality of TextNet.

To see a demo of TextNet in action, visit [*https://textnetsms.com*](https://textnetsms.com).

1. Background and Goals

In 2016, the United Nations declared internet access to be a human right in order to protect an individual's freedom of expression [1]. As of 2018, roughly 95% of the world's population have access to text-based mobile messaging (SMS), but only 55% have access to mobile cellular data (3G or better) [2]. In regions that do support cellular data, the prices of these plans are often prohibitively expensive to low-income demographics (students, new grads, families, etc.). Particularly in Canada, the financial barrier is massive, with mobile data plans that are among the costliest in the top 32 wealthiest countries worldwide (Figure 1: Price per GB of mobile data plans in wealthy countries) [3]. The impact is shocking, as a 2016 study revealed that 71% of surveyed Canadian families are forced to take money out of their food budget to pay for their essential internet services [4]. The stakeholders include the TextNet team, Queen's University, cell phone users with strong cell networks, and cell phone users with poor cell networks. TextNet seeks to bridge this financial and technological accessibility gap that disadvantages individuals worldwide and deprives them of their right to internet access.

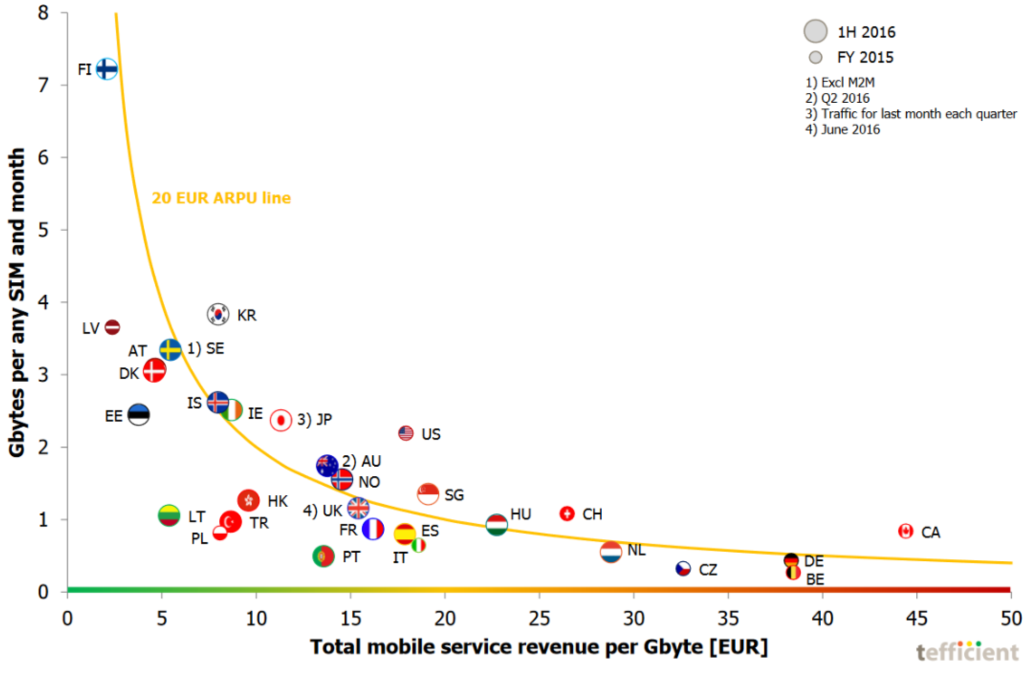


Figure 1: Price per GB of mobile data plans in wealthy countries

1.1 Functional Requirements/Specifications

The solution must to provide end users with quick and reliable access to a range of important internet functionalities using their SMS-enabled mobile device without internet access. These functionalities include general queries, map directions, language translations and local business recommendations. In addition to supporting these functions, TextNet should also be able to easily support the addition of further functions in the future. The TextNet service should accurately determine the intent of each received query and handle it appropriately. All queries to the TextNet service should return some kind of text message within 30 seconds of receiving the request. In the event of a timeout or other error, the response should communicate the cause of the issue in a clear and user-friendly manner.

1.2 Constraints

TextNet has received funding of $200 from the Department of Electrical and Computer Engineering at Queen's University. The total timeline for this project is 21 weeks.

Changes to existing cell phones or networks are costly and influencing them for the sake of this project is not feasible. For this reason, TextNet's performance should not require any changes to existing SMS networks.

It is not within the scope of TextNet to build out the services which provide the search functionalities outlined above. For this reason, the performance of TextNet will ultimately be constrained by the performance of the existing APIs that provide these services.

1.3 Project & Risk Assessment

The completed project takes all of these factors into consideration and successfully meets every requirement. The stakeholders in this project, which include the TextNet team, Queen's University, cell phone users with strong cell networks, and cell phone users with poor cell networks. User acceptance testing with cell phone users in Ontario have has shown strong evidence of the system's effectiveness, and high potential for a wide scale implementation. This shows evidence that all stakeholders will be happy with the performance. Performance is the only concern of cell phone users, and though it has only been tested on Ontario users, it is believed that it will be enjoyed even more by users in regions with poor networks. The introduction of TextNet into areas of the developing world which do not currently support cellular internet would greatly increase the overall capabilities of the network without requiring changes to infrastructure. The implementation of this project into regions with expensive data plans, such as Canada, could result in individuals canceling their data plans and saving hundreds of dollars per year. This would free up budget space and improve the lives of many people. If TextNet were widely scaled, it would be important to focus intently on mitigating the risk of performance issues which might lead to a loss of these subscribers. The stakeholders at Queen's University and the team are pleased that their goals for performance, budget, timing and impact have all been met.

2. Design Process & Key Decisions

In designing TextNet, the team had to break down a very large problem into manageable chunks. The overall goal was to create a SMS based interface to the Internet which provides basic Internet functionality. Designing a text-message based alternative to web browsers is a daunting challenge, so the problem was broken up into more manageable goals.

2.1 Design Principles

Before fragmenting the project into smaller problems, the team laid out a set of design principles to guide the definition and implementation of each project component. TextNet is a software-based project, and therefore follows popular software development principles, in addition to unique principles set by the team.

TextNet should be **modular**, meaning each component of the architecture should have few interdependencies and should be replaceable if needed. Each component should also be **parallelizable**, meaning problem chunks should fall into an agile based development approach, rather than a waterfall approach. When possible, all team members should be able to work on separate functionality in parallel. Each component should be **extensible**, meaning every piece of code should be readable and easily edited to add or remove functionality. **Ease of use** takes top priority in interface and interaction design. The TextNet system should also be **robust**. TextNet should handle edge-case inputs, variable loads, API failures, etc. without crashing the system or compromising the user interface. Lastly, TextNet should be SMS **device independent.** This means that any SMS enabled device from flip phones to iPhones should be able to access TextNet's services.

2.2 Connecting SMS Devices to The Server (Bi-directional)

The first problem chunk was with SMS itself - how to send / receive SMS data to / from an Internet-connected server. Modern alternatives to SMS such as iMessage and WhatsApp are Internet based, but SMS is not routed through the Internet. After researching SMS to Internet connection, the team concluded that Twilio's SMS to server technology would be the basis of the communication channel. Twilio provides servers which can accept SMS messages like any other mobile device. These servers are called 'virtual phone numbers' and are the middleman between the TextNet backend and a user's mobile device. This bi-directional SMS communication channel is illustrated in Figure 2.

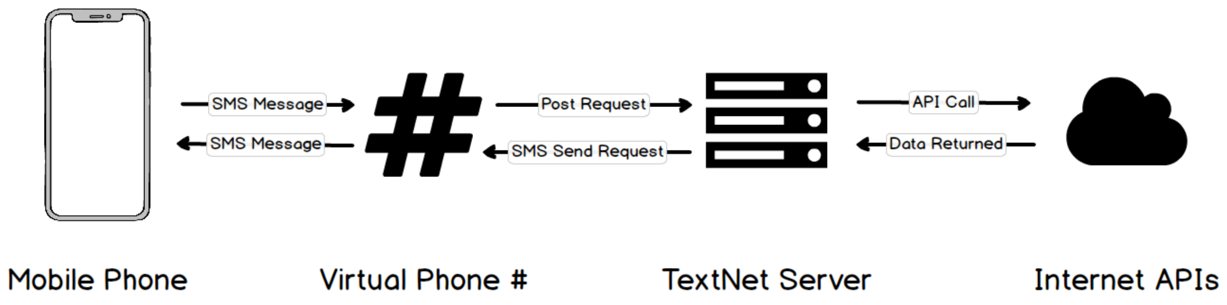


Figure 2: TextNet Communication Channel

2.3 Internet Functionality Scope

Once the communication channel for TextNet was implemented, the next problem chunk was to scope the Internet functionality that TextNet would provide. A full interface to the Internet with functionality such as: URL queries, web page access, account login, search results, etc. was outside the scope of the capstone project. A minimal interface with, for example, only one function such as text translation was too small of a scope for a yearlong capstone project. With these upper and lower boundaries, the team was tasked with creating a useable interface to the Internet. One which provided enough functionality to be useful and valuable without entering 'feature creep' or compromising robustness of the core features.

In order to create a scope for TextNet a minimum viable product vision was developed. The team brainstormed the core TextNet functionality by imagining ourselves in the shoes of users. The following are examples of this process:

Example Use-Case #1
Sally is a visitor, walking around the streets of Toronto. She ran out of data for this month's cycle and wants to find a highly rated Chinese food place nearby. Sally simply messages TextNet to recommend Chinese food nearby. After a few seconds, a list of nearby Chinese restaurants with ratings and distances is returned to Sally. She then selects one and asks for directions to the address. The walking directions to that restaurant are texted back to Sally.

Example Use-Case #2
Jerry recently crossed over into Montreal from the US and doesn't have access to a data plan, but still has SMS access. Jerry is interested in learning French, but without data, he can't look up translations. With TextNet, he can simply ask to translate a text string to English and will receive a data-less translation seconds later. Jerry is also missing his daily, on-the-go, news updates. With TextNet he can ask for a news update to see snippets of the current headlines.

After brainstorming use-cases, the team narrowed the core features down to the following Internet primitives as a minimum viable product.

1. *Answer who, what, when, where, why questions*
2. *Provide step-by-step directions between locations*
3. *Recommend businesses based on location and business type.*
4. *Translate strings of text between languages.*
5. *Provide news updates*

These five features are at the core of the mobile Internet. They are simple enough to be entirely text based and offer enough value to make TextNet a viable solution. Lastly, and most importantly, these features are backed by large Internet APIs (application program interfaces) with huge amounts of easily accessible and verified data. This realization is key to the success of TextNet. Leveraging existing APIs provides a rich interface to select Internet primitives.

2.4 API Selection

Creating a knowledge engine, a global map, restaurant database, or language dictionary is outside the scope of this project. Instead, the team elected to leverage existing APIs to query for data. Once the minimum viable product was developed, each internet primitive had to be matched to an API to access relevant data.

General questions such as *who, what, when, where,* and *why* questions are routed to Wolfram Alpha, a knowledge engine which supports such queries. Directions are routed to Google Directions API. Business recommendations are routed to Yelp's business recommendation API. Translations are routed to Google Translate. And news updates are routed to CNN's news API. These APIs provide all of the data for TextNet and are considered an approximation of the internet for basic mobile usage. An overview of the APIs utilized are shown below in Figure 3.

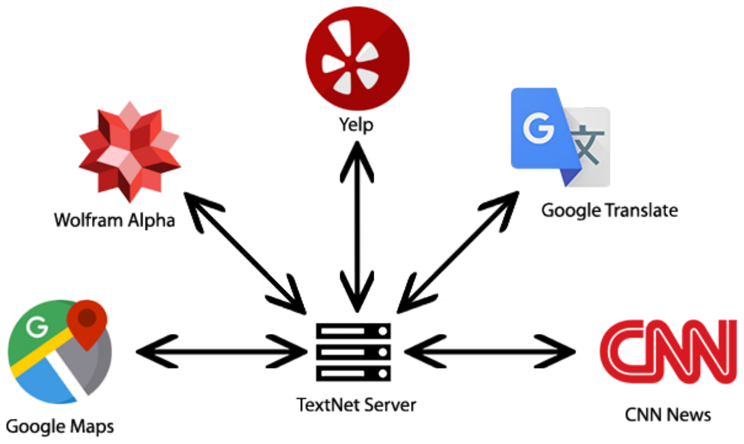
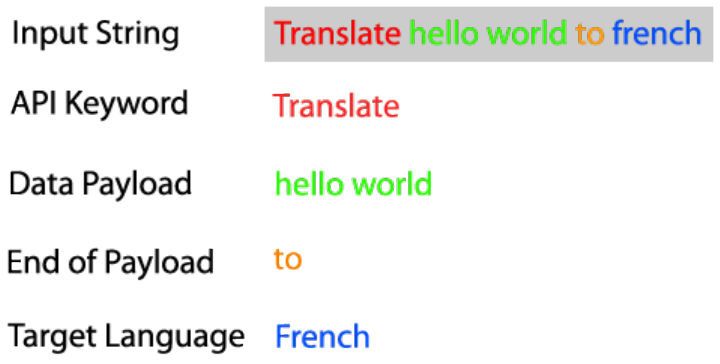


Figure 3: TextNet API Routing

2.5 Parsing SMS Messages to Derive Intent

The next step in the design process was to determine how to translate a user's text message query into actionable data to pass to the most relevant API. Then return the result back in SMS format, or return a message indicating that the data cannot be parsed. The following summarizes the design steps the team took to parse the SMS data.

The team's original design decision was to use NLP (natural language processing) to parse the intent of the user's message and extract key data. After early tests, this approach was set as a stretch goal due to lack of team experience in this area. Instead, the team opted for a keyword-based approach. This approach allows for exact interpretation without approximation, matching input data directly to an API and extracting key data based on keywords. Figure 4 below shows the details of this keyword based parsing approach for the translation function.



Input String	Translate hello world to french
API Keyword	Translate
Data Payload	hello world
End of Payload	to
Target Language	French

Figure 4: TextNet keyword parsing for translation

3. Implementation

3.1 Software Architecture

3.1.1 Front-End

In the Front-end tech stack, the team considered developing a mobile application that made use of the user's phones messaging features (which is the ability to send text messages without the need for cellular data) with additional features like having access to the users' GPS location which would be attached to every message sent. This information was valuable for the directions service giving directions from users' exact locations (latitude and longitude) to their desired address. The second option was to make use of the user's native text messaging application; giving up on the additional features like access to users' location which the team decided on over the initial option as there seemed to be more cons to the first option compared to the second.

Having the mobile application came with several caveats, as follows:

1. *Privacy Issues due to location access*
2. *Developing multiple applications for different mobile devices*
3. *Time constraint*
4. *User requires internet/cellular data access to download application*
5. *Today's users are not interested in downloading more applications*

Using the user's native text messaging apps solved a lot of the foreseen problems with the initial solution. Time was saved not developing and testing mobile applications for different devices (iPhone, Android, Blackberry) and in order to conserve users' privacy the team required them to provide their chosen current location to use the directions service. Another major reason the team decided against the mobile application was the fact that the user would be required to download the mobile application which would require them to have access to internet or cellular data, as this was the problem the team were trying to solve as the target market do not have access to these services it would be impractical to use that solution. Finally in the **2017 U.S mobile report** by comScore 51% of U.S consumers do not download a new application per month, this article shows that the App boom is over and if you are not one of the top contenders(WhatsApp, Snapchat, Instagram, Facebook, Messenger, Twitter, YouTube) it is difficult to get downloads.

Not using a mobile application put the service in the hands of every user with a cellphone that was capable of sending text.

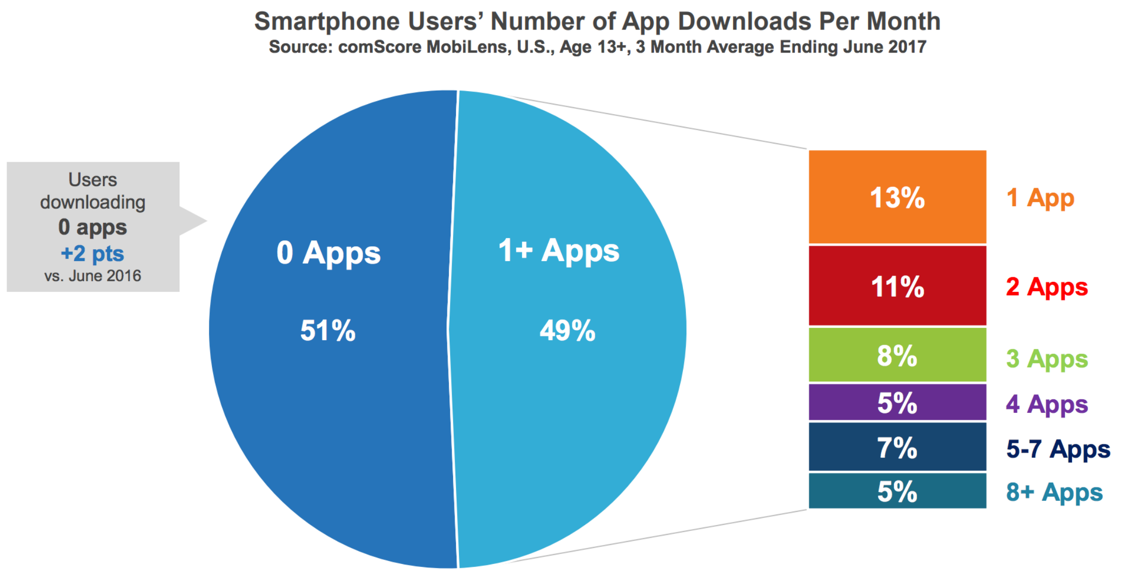


Figure 5: Quantifying App Downloads

3.1.2 Back-End

The backend consists of two parts: *Twilio* and the *TextNet* server. Twilio is a developer platform for communications. Software companies use Twilio APIs to add capabilities like voice, video, and messaging to their applications. Behind Twilio APIs is a software layer that connects and optimizes communications networks around the world. TextNet makes use of Twilio's Messaging service as it provides API and SDKs to send and receive SMS, MMS, and IP messages globally from the web app, and use intelligent delivery features to ensure messages get through. Twilio provides a virtual number which the users can text, this message is then sent to the *TextNet* server via an HTTP POST request. The code block below in Figure 6 is a simple application that sends the SMS 'Hi from TextNet' to the recipient's phone number.

```
const accountSid = 'ACXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX';
const authToken = 'your_auth_token';
const client = require('twilio')(accountSid, authToken);

client.messages
  .create({
     body: 'Hi from TextNet',
     from: '+15017122661',
     mediaUrl: 'https://c1.staticflickr.com/3/2899/14341091933_1e92e62d12_b.jpg',
     to: '+15558675310'
   })
  .then(message => console.log(message.sid));
```

Figure 6: Twilio Hello World

This brings us to the server which is written in JavaScript using the *Node.JS* and *Express* web frameworks. *Node.js* is an asynchronous event driven JavaScript runtime, designed to build scalable network applications while *Express* is a minimal and flexible middleware web application framework specifically for *Node.js* that provides a robust set of features for web and mobile applications.

Before choosing this tech stack the team also considered writing the server in python, using Django web framework. A major factor for choosing JavaScript over python was the team. All members of the team have had experience developing web applications prior to this project, there was going to be little to no learning curve compared to writing the server in python which the team had no experience in. In addition to that *Node.js* has become the most popular web framework, it also has the advantage with performance. *Node.js* applications can be scaled using the cluster module to clone different instances of the application's workload using a load balancer.

Code block below is the simplest Express app you can create. It is a single file app.

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => res.send('Hello World!'))

app.listen(port, () => console.log(`Example app listening on port ${port}!`))
```

Figure 7: Simplicity of Node.js With Express

This starter app starts a server and listens on port 3000 for connections. The app then responds with "Hello World, I am TextNet!" for requests to the root URL (/). For every other path, it will respond with a **404 Not Found** as no other route has been defined.

Upon receiving the user's message in the express server, the message is parsed getting the user's intent which could either be DIRECTIONS, TRANSLATION, RECOMMENDATION & WHO, WHAT, WHEN, WHY. For these services, Google Directions, Google Translation, Yelp and Wolfram Alpha were used respectively.

Google Directions API is a service that calculates directions between locations using an HTTP request. The application allows searching for directions for several modes of transportation, including transit, driving, walking or cycling. The API returns the most efficient routes when calculating directions. An example of the API call is shown in Figure 8 below.

```
https://maps.googleapis.com/maps/api/directions/json?origin=Toronto&destination=Mon
treal&key=YOUR_API_KEY
```

Figure 8: Example Google Directions Request

Google Translate API is one of the biggest translation services that can dynamically translate text between thousands of language pairs. An example of translation is shown below in Figure 9.

```
let [translations] = await translate.translate(text, target);
translations = Array.isArray(translations) ? translations : [translations];
console.log('Translations:');
translations.forEach((translation, i) => {
  console.log(`${text[i]} => (${target}) ${translation}`);
});
```

Figure 9: Example Google Translation Request

The Yelp API allows us to get the local content from millions of businesses across 32 countries.

```
GET https://api.yelp.com/v3/businesses/search
```

Figure 10: Example Yelp API Request

The Wolfram Alpha API returns a single plain text result, in general, the API is designed to deliver brief answers in the most basic format possible. It is implemented in a standard REST protocol using HTTP GET requests.

```
http://api.wolframalpha.com/v1/result?appid=DEMO&i=How+far+is+Los+Angeles+from+New+
York%3f
```

Figure 11: Example Wolfram API Request

The Google Maps Direction API returns very detailed data on different possible routes, latitudes and longitudes, and is HTML formatted. Because of this, the data had to be thoroughly parsed to retrieve the desired result.

If the user's intent could not be determined, a message notifying them that their question could not be understood is sent. When the intent is determined, the user's message is formatted into the accepted data for the correct REST API. The response received from the service is then formatted into useful information for the user to understand.

Figure 12 shows the architecture for TextNet. Starting from an SMS capable device, a user sends a message to a virtual phone number. That message is routed through Twilio, and sent to the TextNet server, which is connected to the internet. The TextNet server then parses the message, matches the query, and retrieves data from the most appropriate API. The resulting data is then formatted into an SMS message and sent back to the user.

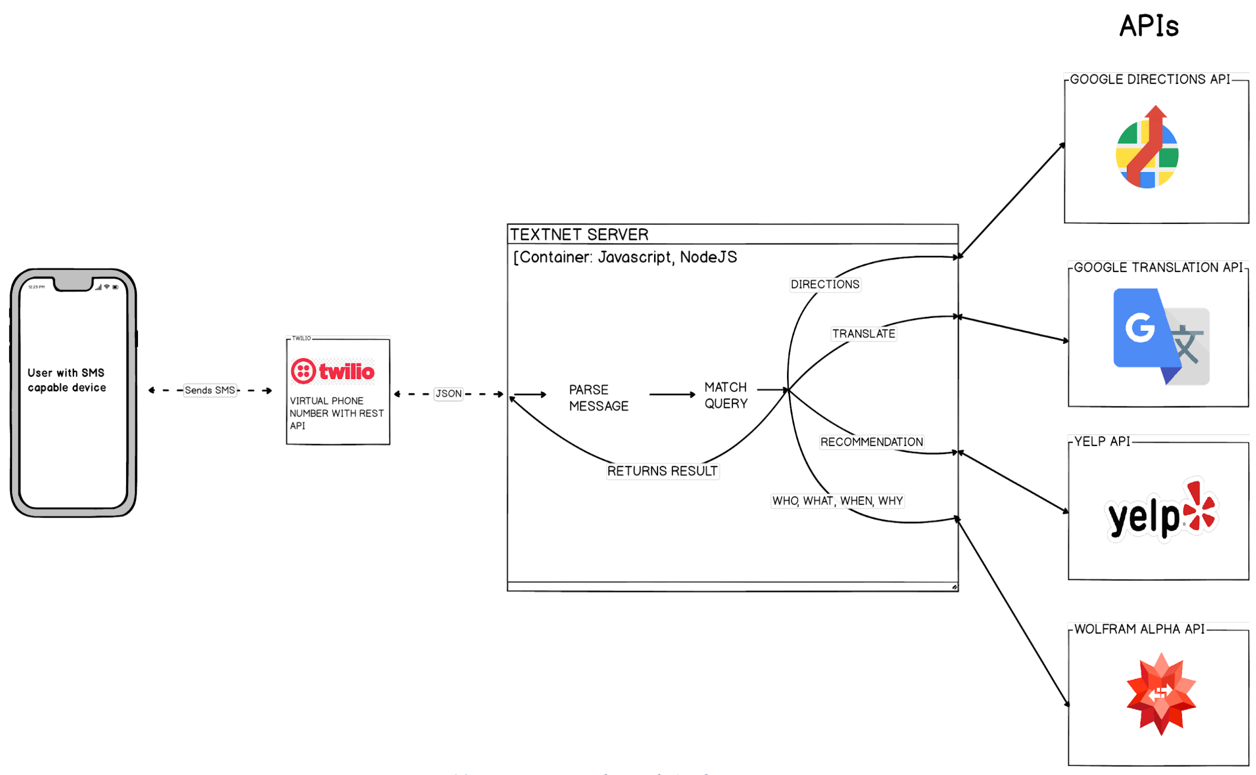


Figure 12: TextNet Technical Architecture Diagram

3.2 Implementation Schedule

Major Tasks are as follows:

1. *ZenHub set-up and tasks assigning*
2. *Basic Node.js. / Express server Infrastructure*
3. *Twilio API set-up to send and receive messages*
4. *Connecting External APIs*
5. *Core Functionalities Integrated*
6. *User Acceptance Testing*

As shown above and in Figure 13 are the major tasks the team had to accomplish. The first week was spent identifying project requirements, then translating the requirements into user stories (an agile project management methodology), which were further broken down into tasks to be slotted into ZenHub. The tasks were assigned based on each team member's strengths as well as what they wanted to learn, which proved to work as team members were interested in different tasks and had different technical experience. The following week was then spent designing out Node.js server ensuring all endpoints worked and had set up error checks and middleware.

Following that during weeks 9 to 12 each team member was tasked to creating a free trial Twilio account, also reading the docs to understand how to send and receive text messages via the Node.js Framework. The team then all applied it and tested on our local machines to understand the workings of the process. Finally, it was applied to our product and at this stage the team had a working Node.js server that could send and receive text messages. At this point was where the team had missed an important deadline which was submitting the budget for the project, this was due to a miscalculation that the free trials given by Twilio could cover the whole process of building and finalizing the project. Twilio free trial provided **$15** dollars per account, and with a cost of **$0.03** for out bounding SMS and **$0.005** for in bounding, with the project relying heavily on these SMS the limit was reach, in addition to that the free trial only provided the ability to send and receive text from only certain numbers. This did not suit the team's need, especially for demoing, the team wanted anyone to be able to send a text from their device and receive a response, so the team invested **$20** to upgrade to a premium account.

The weeks to come were then followed by each team member choosing a certain service offered i.e. translation, directions, recommendation, queries, news and applying their providers' APIs into TextNet server, the team had to understand the required parameters the API could accept and also format the response into information the end user found useful. At this point the team ran into issues managing environment variables specially with the format Google provided their keys, with further research and more time a solution was found to fix this making these variables accessible to all members in the team but private to the public.

Finally, the team integrated the external APIs with Twilio text messaging service and had the full application working cohesively; from the users text message to the service of choice with the response then sent back to the user via text, the team spent considerable about of time with user testing to ensure the usability of our product. After this was done the team was then posed with the task of how to make this public the team had considered deploying the application on Heroku, Zeit Now or even Google Cloud but decided against it because the team wanted to control when it was live since there wasn't much credit on the Twilio account, the team then decided on **Ngrok**; Ngrok exposes local servers behind NATs and firewalls to the public internet over secure tunnels. This was useful for Demoing without deploying, so all the team had to do was run the server on our local machine whenever needed and exposing the local port on our machine to the Twilio using this tool.

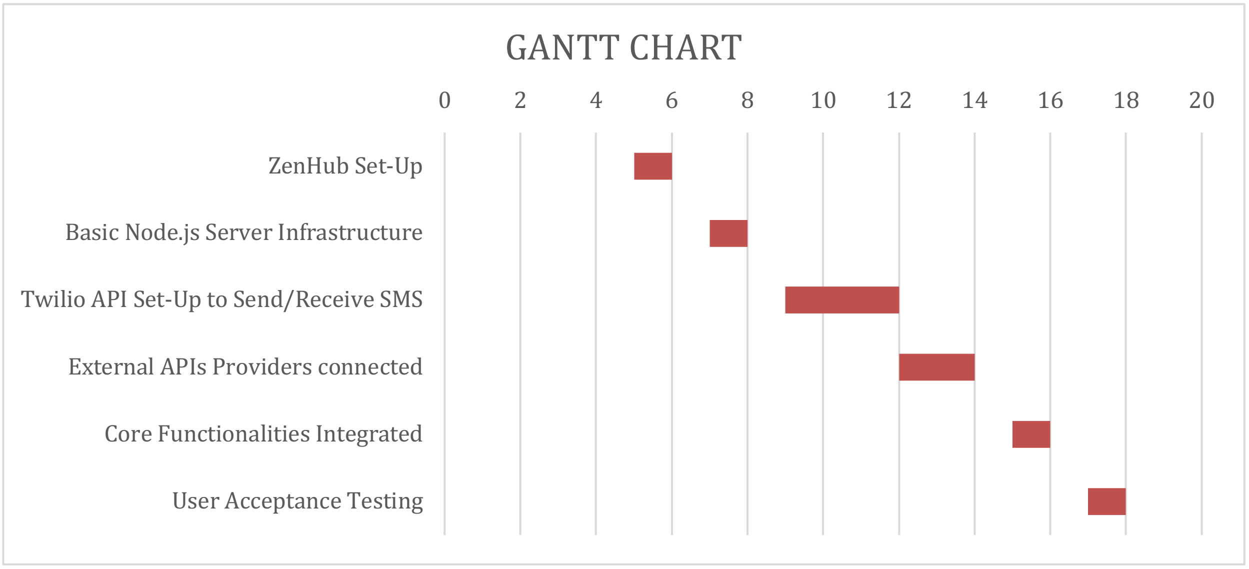


Figure 13: Gantt chart

4. Testing and Evaluation

4.1 Important Concepts

Testing was important for the project because functionality of the application was directly dependent on user input, and users are not guaranteed to interact with the application as expected.

Software testing for the application is composed of three concepts: unit tests, integration tests, and test suites. Unit tests are a level of software testing where individual components of a software are tested. In the case of this application, unit tests are composed for each individual function within the controllers. Integration tests are a different level of software testing where individual units are combined and tested as a group, with the purpose of exposing faults in the interaction between integrated units. In the case of this application, unit tests and integration tests are grouped together because certain functions within a controller encompass all other functions, essentially performing the integration with one function call. This function is tested against, and if the output is as expected, then it can be assumed that integration has been tested. Finally, test suites, shown below in Figure 14 are a collection of test cases (both integration and unit) that are intended to be used to test a software program to show that it has some specified set of behaviors. In the case of this application, each controller is associated with an individual test suite.

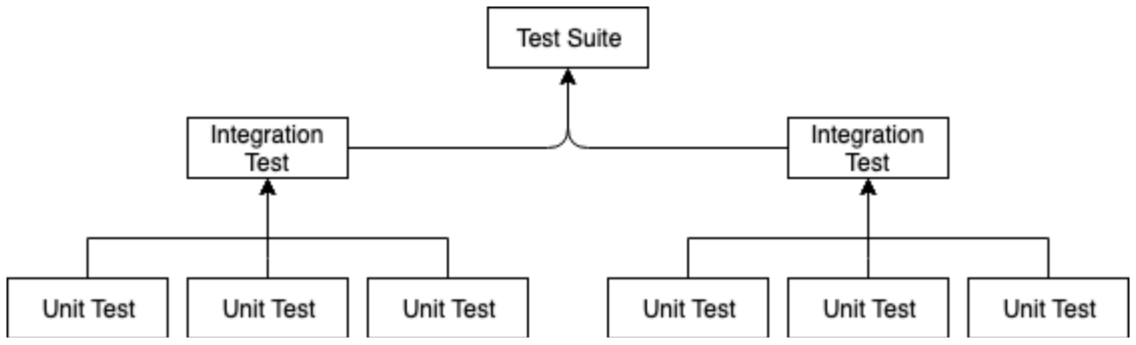


Figure 14: Test Suites and Their Components

Code coverage was used to quantify the effectiveness of the developed test suites. Code coverage is a measure used to describe the degrees to which the source code of software is executed when a particular test suite runs. Measured as a percentage, a program with low test coverage has had less of its source code executed during testing, which suggests it has a higher chance of containing undetected software bugs compared to a program with high test coverage.

There are three primary terms used to describe each line of the program. A hit indicates that the source code was executed by the test suite. A partial indicates that the source code was not fully executed by the test suite, and there are branches of logic that were not executed. A miss indicates the source code was not executed by the test suite.

Coverage is the ratio of **hits / sum (hits + partials + misses)** [5]. For example, a program that has 20 lines executed by tests out of 100 lines will receive a coverage ratio of 20%.

4.2 Tools and Approach

The testing and validation of the software was done using two tools: Jest and Codecov. These tools can be imported as packages into projects built with Node using the Node Package Manager.

4.2.1 Jest

Jest is a JavaScript testing framework with a focus on simplicity, built and open-sourced by Facebook that provides a wide array of assertion functionalities and a visually pleasing Command Line Interface to execute test suites.

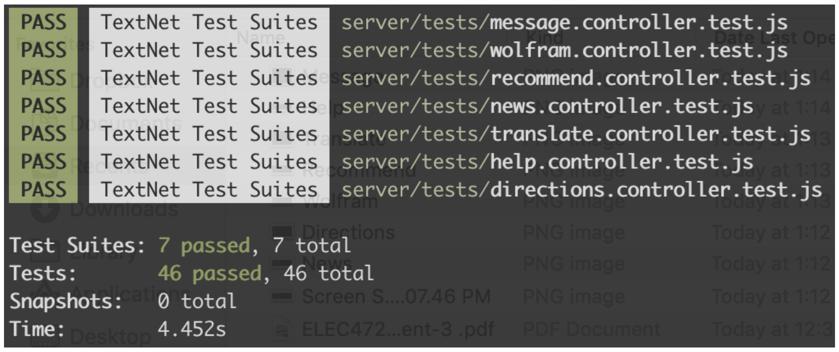
Other robust open-sourced JavaScript frameworks exist, such as Mocha and Chai, however Jest was selected because members of the team had prior experience using the tool and wanted to focus on feature development as opposed to learning the syntax and semantics of a new framework.

Jest can run with zero configuration out of the box; however, some minor configurations were needed in order to identify files within the application that contained unit tests. These files were identified with a filename pattern of ***.test.js**. Controller and test suite folder/file structure can be found in the appendix.

4.2.2 Breakdown of Test Suites

Testing for the application was broken down into seven test suites, one for each controller. Only controllers were tested within the application due to all business logic being contained within these files - everything ranging from API integration, message parsing, formatting and response was done through each individual controller. Other parts of the application were assumed to function according to specification, such as the Node.js and Express server. The potential time spent working on test suites for these tools was used for feature development instead, a tradeoff the team agreed to make.

An example of running the entirety of the test suites for the application can be seen in Figure 15. There are 7 test suites, composed of 46 unit tests, which take 4.452 seconds to execute. See the appendix for a breakdown of each test suite into individual unit tests.



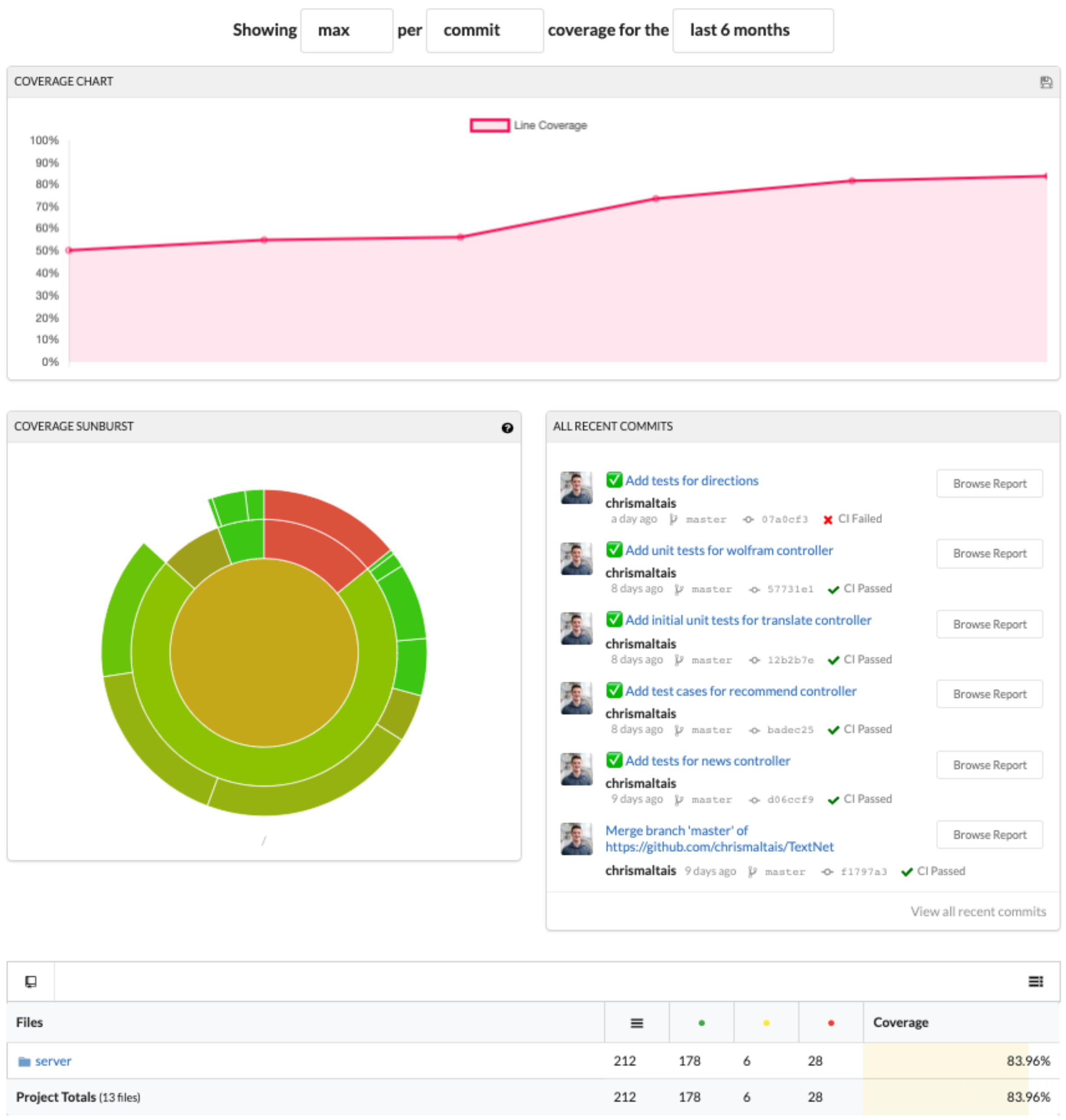
```
 PASS   TextNet Test Suites  server/tests/message.controller.test.js
 PASS   TextNet Test Suites  server/tests/wolfram.controller.test.js
 PASS   TextNet Test Suites  server/tests/recommend.controller.test.js
 PASS   TextNet Test Suites  server/tests/news.controller.test.js
 PASS   TextNet Test Suites  server/tests/translate.controller.test.js
 PASS   TextNet Test Suites  server/tests/help.controller.test.js
 PASS   TextNet Test Suites  server/tests/directions.controller.test.js

Test Suites: 7 passed, 7 total
Tests:       46 passed, 46 total
Snapshots:   0 total
Time:        4.452s
```

Figure 15: Test Suite Results

4.2.3 Codecov

Codecov is a code coverage report analysis tool used to visualize what branching logic has been executed within the source code. An example of the home screen for a project can be seen in Figure 16**.**



Showing max per commit coverage for the last 6 months

COVERAGE CHART

COVERAGE SUNBURST

ALL RECENT COMMITS

Add tests for directions chrismaltais a day ago master 07a0cf3 ✗ CI Failed	Browse Report
Add unit tests for wolfram controller chrismaltais 8 days ago master 57731e1 ✓ CI Passed	Browse Report
Add initial unit tests for translate controller chrismaltais 8 days ago master 12b2b7e ✓ CI Passed	Browse Report
Add test cases for recommend controller chrismaltais 8 days ago master badec25 ✓ CI Passed	Browse Report
Add tests for news controller chrismaltais 9 days ago master d06ccf9 ✓ CI Passed	Browse Report
Merge branch 'master' of https://github.com/chrismaltais/TextNet chrismaltais 9 days ago master f1797a3 ✓ CI Passed	Browse Report

View all recent commits

Files	≡	●	●	●	Coverage
server	212	178	6	28	83.96%
Project Totals (13 files)	212	178	6	28	83.96%

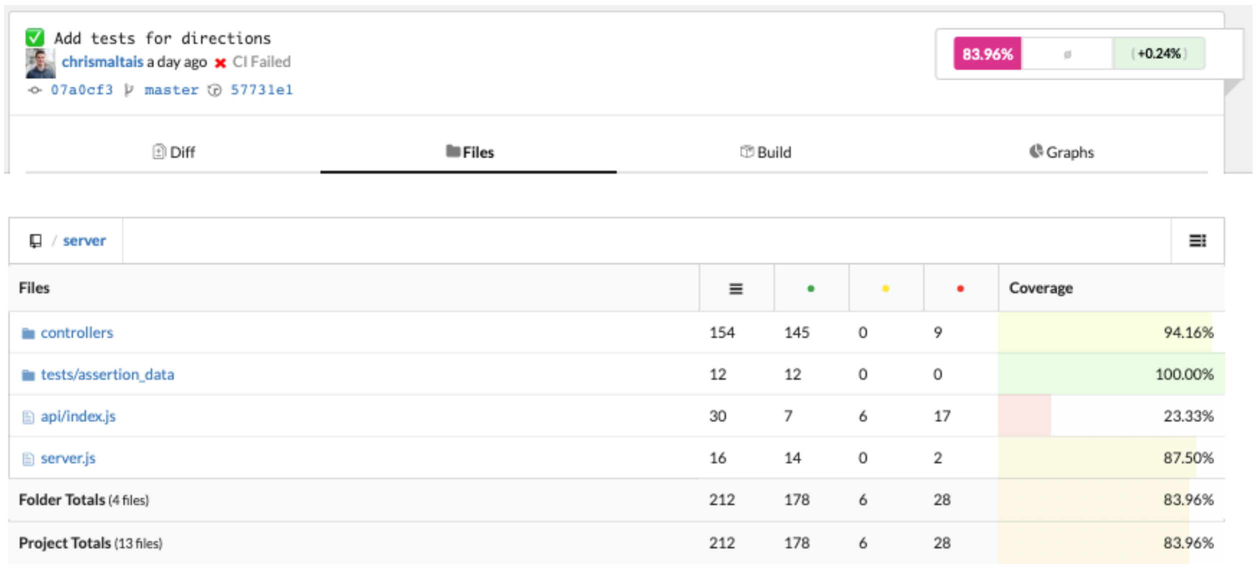
Figure 16: Codecov Home Screen Example

The application is connected to Codecov via authentication token that is connected to GitHub and syncs the code coverage with the repository via GitHub Webhooks. This means that commits can be tracked on the Codecov platform, and the incremental improvement of the project can be tracked against pushes to the repository.

At the bottom of Figure 16 there is a high-level analysis of lines of code tracked, broken into hits, partials and misses. The lines of source code to be included in the code coverage can be modified to exclude configuration files, packages, etc. so as to not skew the code coverage percentage. In the case of this application, examples of files excluded were Node modules (packages), the Dockerfile, environment variables, and version control folders/files.

At the bottom of Figure 16**,** there is also the total code coverage percentile score: ~84%. This means that 84% of our application is executed and validated against via automated tests.

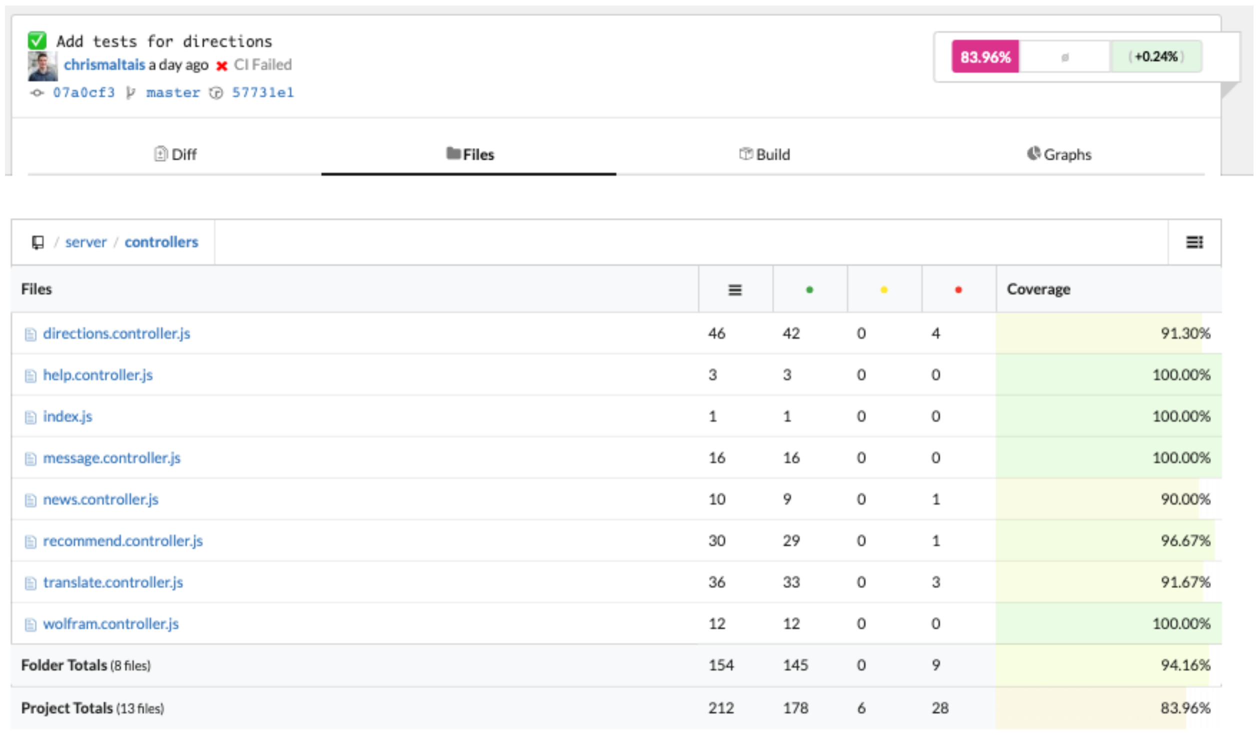
Figure 17 is an example of Codecov's ability to drill down into folders of a project, for example the *server* folder which contains the majority of code for the application. The *controllers* folder has a coverage of 94%, compared the overall 84% for the application. This reiterates the teams focus on writing tests for the parts of the application that contain business logic and API connections. Another aspect to note about Figure 17 is the 23% coverage for the *api/index.js* file. No tests were written for this file because it contains basic infrastructure for Node.js and Express with minimal business logic. As previously mentioned, the team made the tradeoff to focus on feature development as opposed to unit testing robust tools. Figure 17 also shows the ability for Codecov to track percentage increase of code coverage for each commit to the repository. This is a useful feature for quantifying how many unit tests were contributed for different aspects of the project.



Files	≡	●	●	●	Coverage
controllers	154	145	0	9	94.16%
tests/assertion_data	12	12	0	0	100.00%
api/index.js	30	7	6	17	23.33%
server.js	16	14	0	2	87.50%
Folder Totals (4 files)	212	178	6	28	83.96%
Project Totals (13 files)	212	178	6	28	83.96%

Figure 17: CodeCov - Test Coverage Details

Figure 18 shows the code coverage breakdown for each individual controller. Every controller has > 90% coverage, which the team deemed satisfactory. A tradeoff between effort required to write unit tests and amount of code covered by tests was discussed, and the team deemed the extra effort to make all controllers 100% to be overkill. In the future, additional tests can always be written to make the application more robust.



Add tests for directions
chrismaltais a day ago ✖ CI Failed
07a0cf3 master 57731e1

83.96% (+0.24%)

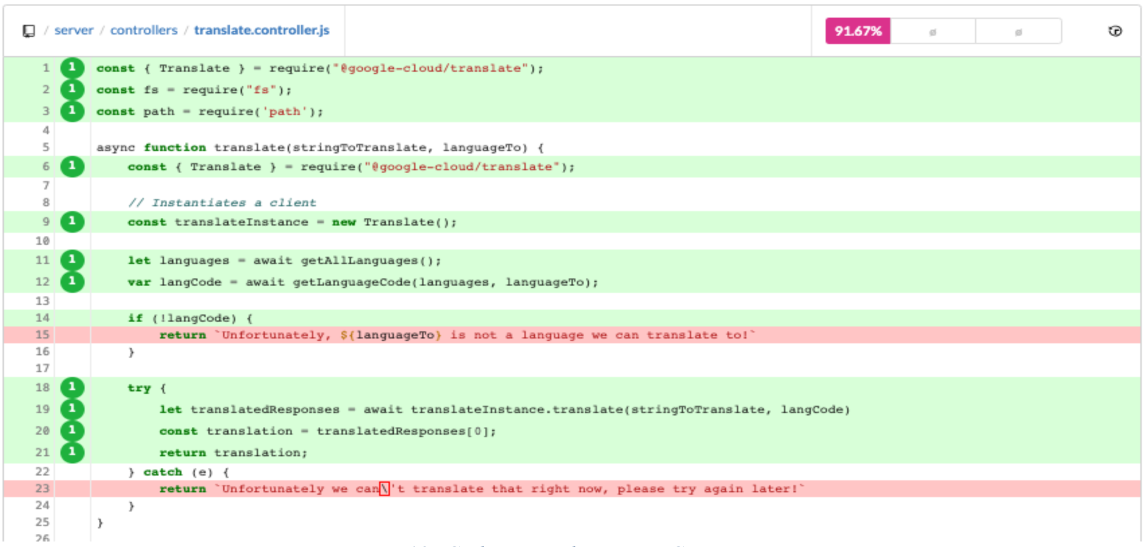
Diff | Files | Build | Graphs

server / controllers

Files	≡	●	●	●	Coverage
directions.controller.js	46	42	0	4	91.30%
help.controller.js	3	3	0	0	100.00%
index.js	1	1	0	0	100.00%
message.controller.js	16	16	0	0	100.00%
news.controller.js	10	9	0	1	90.00%
recommend.controller.js	30	29	0	1	96.67%
translate.controller.js	36	33	0	3	91.67%
wolfram.controller.js	12	12	0	0	100.00%
Folder Totals (8 files)	154	145	0	9	94.16%
Project Totals (13 files)	212	178	6	28	83.96%

Figure 18: CodeCov - Controller Script Test Coverage Percentages (%)

Figure 19 shows the ability of Codecov to inspect files of code. The green highlighted lines indicate that the line was marked as a hit at least once in the amalgamation of all test suites, and the red highlighted lines indicate that the line was marked as a miss, where no test suites cover that specified branch of logic. In Figure 19 the *translate(stringToTranslate, languageTo)* function can be seen, and it becomes evident that no test cases were written to check if the target language is, in fact, a language. Of course, the code contains the appropriate error handling, however the error handling has not been asserted to work via automated tests. This is an example of how Codecov's code coverage visualization helps to determine areas of an application's code base that still needs to be tested.



server / controllers / translate.controller.js — 91.67%

```
const { Translate } = require("@google-cloud/translate");
const fs = require("fs");
const path = require('path');

async function translate(stringToTranslate, languageTo) {
    const { Translate } = require("@google-cloud/translate");

    // Instantiates a client
    const translateInstance = new Translate();

    let languages = await getAllLanguages();
    var langCode = await getLanguageCode(languages, languageTo);

    if (!langCode) {
        return `Unfortunately, ${languageTo} is not a language we can translate to!`
    }

    try {
        let translatedResponses = await translateInstance.translate(stringToTranslate, langCode)
        const translation = translatedResponses[0];
        return translation;
    } catch (e) {
        return `Unfortunately we can\'t translate that right now, please try again later!`
    }
}
```

Figure 19: Codecov - File Inspect Coverage View

4.3 Twilio

One of the difficulties with testing the application was determining how to handle the Twilio API. There were two underlying problems to tackle, the ability to mock an incoming SMS (i.e. an SMS is being sent to the application's server), and the ability to mock an outgoing SMS (i.e. the application's server will send out a text).

Twilio supports mocking an outgoing request via a service they describe as "Magic Number", which means SMS messages can be sent from the "Magic Number" to any specified number, there is no requirement for a live server [6].

The issue with the "Magic Number" is each developer would need to explicitly specify their phone number in the code, so when you ran unit tests, the code would execute and send you a text. This theoretically could be done by setting the developer's personal number as an environment variable in the code, essentially a "secret" that gets added to the *.gitignore* file and wouldn't be checked into the codebase, keeping the developer's phone number private. For each unit test/integration test, this didn't seem to scale well. When the codebase expands and tests increase, the developer would receive an increased amount of SMS messages simultaneously when all test suites are run.

For the second problem, Twilio does not offer the option to mock incoming text messages, developers must send an SMS message to a live number that is associated with the application's Twilio account, which costs $0.005 per text. When scaling to hundreds of tests, this option also doesn't work. The team decided to move forward with the assumption that the Twilio service will work as expected, and what should be tested is the business logic of the application.

5. Conclusion and Significance

TextNet allows any cell phone user with SMS messaging service to access six common types of internet functions by simply texting their queries to a phone number and awaiting a reply. Based on the functional specifications, performance requirements, budget and timelines set out for this project, it can be concluded that TextNet exceeds expectations. TextNet is fast, accurate, robust, scalable and its functionalities can be easily expanded upon in the future. Without TextNet, about 40% of the world's population does not have access to mobile internet services, and many more people living in the developed world simply cannot afford these services. A wide implementation of TextNet would result in 95% of the world having access to mobile internet, which is on par with worldwide SMS coverage. Access to these services would improve quality of life by increasing access to information and connecting communities.

The TextNet service also has great potential to function as a business that sells its functionality as a service for a low monthly rate. We believe this would be feasible as a partnership with cell providers to sell TextNet as an inexpensive add-on to basic SMS plans. With the right implementation, TextNet can provide an inexpensive and accessible solution to mobile internet access while the world's cellular networks continue to mature.

6. Next Steps

In the future, TextNet could be improved by a number of changes with respect to functionality, architecture and development methodology. From a functional standpoint, additional query type support can always be added to TextNet or existing functions can be improved to be faster, more robust, provide better results, or consume less space. An example of an existing function that the team would like to improve is the message parsing function which identifies the type of a given query. Currently, queries must follow a strict template which may not always be intuitive for users. The team would like to improve this by implementing natural language processing to identify the overall intent of a given statement and provide more flexibility for input sequences and languages. From an architecture standpoint, using containerized microservices would improve the readability, scalability and flexibility of TextNet. By splitting features, developers would be looking at smaller code bases, be able to use different technology stacks based on the needs of each microservice and be able to scale features independently depending on traffic.

7. Project Process Analysis

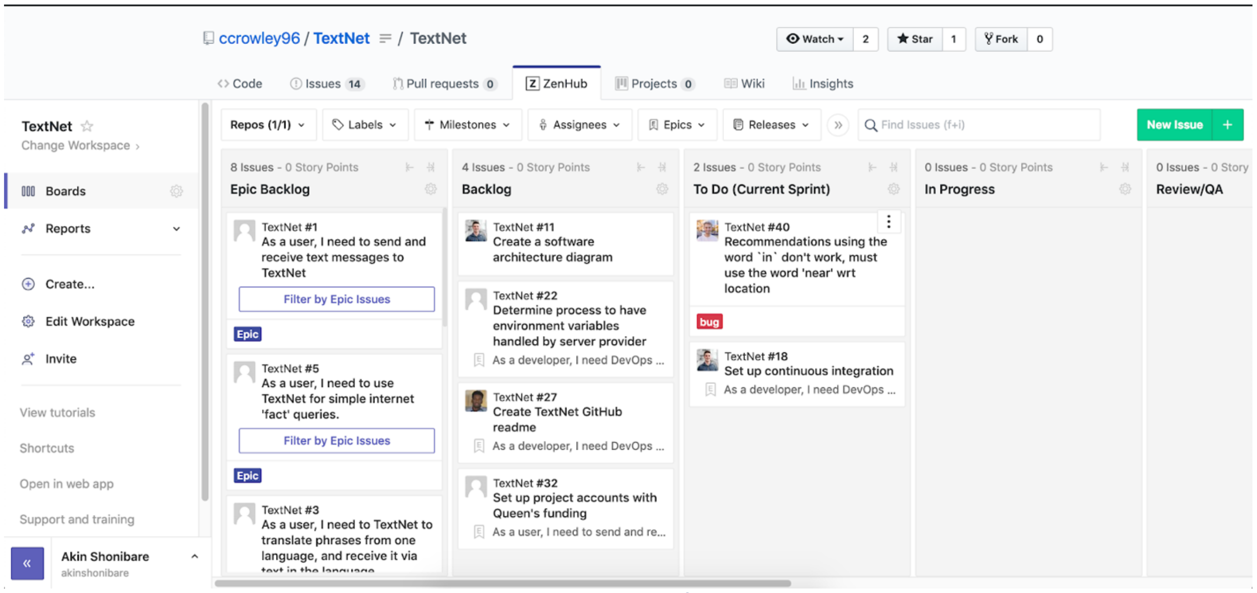
This capstone project went through several phases of development. Initially, during the 490 portion of this course, the team developed an idea called AdByte. AdByte was a data over audio advertising solution which relied on proprietary software from a company named LISNR. The team received written permission to use LISNR's APIs / SDK, but unfortunately, they decided to not support the capstone student account. The team also reached out to the executives of LISNR but received the same response. Because AdByte relied on LISNR technology, the decision was made to pivot and choose a new project.

The new project was TextNet. As described in (*2. Design Process & Key Decisions)*, the design process was broken into components. Considering the entire project as a number of sub-problems helped the team divide work and fulfill the requirements. The team didn't miss any development goals or deadlines, however the project pivot set the development back significantly.

7.1 Project Management

In order to continually meet the project's milestones, the team handled project management by having regular weekly meetings usually after meeting with our teaching assistant along with using ZenHub, the team also had regular Google Hangout calls. ZenHub is a project management and issue tracking tool that integrates natively within GitHub user interface. Tasks and issues were managed through this tool, ensuring agile development within the team. The team chose this tool over popular tools like Jira and Trello, because it is much easier and powerful. The team also used GitHub for version control. Creating feature branches such as: *feature/message-parsing*, or *feature/yelp-integration,* and developing that feature separately from the *master branch* was very important in development. Git workflow aided in code reviews, division of work, and reduction of merge conflicts. ZenHub's integration with GitHub ensured a unified workflow for tasks giving everyone in the team a clear picture about where the team is and what needs to be done. Figure 20 shows an example of our Kanban board, showing our tasks and issues.

For project planning, features were broken into user stories. These stories were further divided into issues, or technical sprint tasks. The team had no strict sprint schedule. Instead, high priority tasks were assigned and implemented first. An example of this is the Twilio messaging architecture; this was a high priority feature that the whole project relied on.



8 Issues - 0 Story Points — Epic Backlog

- TextNet #1 — As a user, I need to send and receive text messages to TextNet — Filter by Epic Issues — Epic
- TextNet #5 — As a user, I need to use TextNet for simple internet 'fact' queries. — Filter by Epic Issues — Epic
- TextNet #3 — As a user, I need to TextNet to translate phrases from one language, and receive it via text in the language

4 Issues - 0 Story Points — Backlog

- TextNet #11 — Create a software architecture diagram
- TextNet #22 — Determine process to have environment variables handled by server provider — As a developer, I need DevOps ...
- TextNet #27 — Create TextNet GitHub readme — As a developer, I need DevOps ...
- TextNet #32 — Set up project accounts with Queen's funding — As a user, I need to send and re...

2 Issues - 0 Story Points — To Do (Current Sprint)

- TextNet #40 — Recommendations using the word `in` don't work, must use the word 'near' wrt location — bug
- TextNet #18 — Set up continuous integration — As a developer, I need DevOps ...

0 Issues - 0 Story Points — In Progress

0 Issues - 0 Story — Review/QA

Figure 20: Issue tracking application used for Agile management - ZenHub

Using these tools aided to the success in meeting our major deadlines and milestones and due to the team's strong prior experience in JavaScript the team was always ahead of our milestones, this meant whenever an issue came up the team was not behind schedule.

Table 1: Final Budget

Service	Price ($)	Paid by Queens	Paid by Team
Twilio Programmable SMS	20	0	20
Google Translate API	0 (FREE TRIAL)	0	0
Google Directions API	0 (FREE TRIAL)	0	0
Wolfram Alpha API	0 (FREE TIER AT 2000 CALLS PER DAY LIMIT)	0	0
Yelp API	0 (FREE TIER TRIAL AT 5000 CALLS LIMIT PER DAY)	0	0
CNN API	0 (FREE TIER)	0	0

8. References

[1] E. Boyle, "UN declares online freedom to be a human right that must be protected," Independent, 5 July 2016. [Online]. Available: https://www.independent.co.uk/life-style/gadgets-and-tech/un-declares-online-freedom-to-be-a-human-right-that-must-be-protected-a7120186.html.

[2] Open Signal, "Global State of Mobile Networks(August 2016)," Open Signal, Aug 2016. [Online]. Available: https://www.opensignal.com/reports/2016/08/global-state-of-the-mobile-network.

[3] tefficient, "Unlimited pushes data usage to new heights," December 2016. [Online]. Available: http://media.tefficient.com/2016/12/tefficient-industry-analysis-5-2016-mobile-data-usage-and-pricing-1H-2016-ver-2.pdf.

[4] ACORN Canada, "Internet for All," January 2016. [Online]. Available: https://www.acorncanada.org/resource/internet-all.

[5] Codecov, "Codecov," [Online]. Available: https://docs.codecov.io/docs. [Accessed 02 02 2019].

[6] Twilio, "Twilio - Test Credentials," [Online]. Available: https://www.twilio.com/docs/iam/test-credentials. [Accessed 03 12 2018].

[7] Facebook, "Jest," [Online]. Available: Facebook. [Accessed 03 01 2019].

Appendix

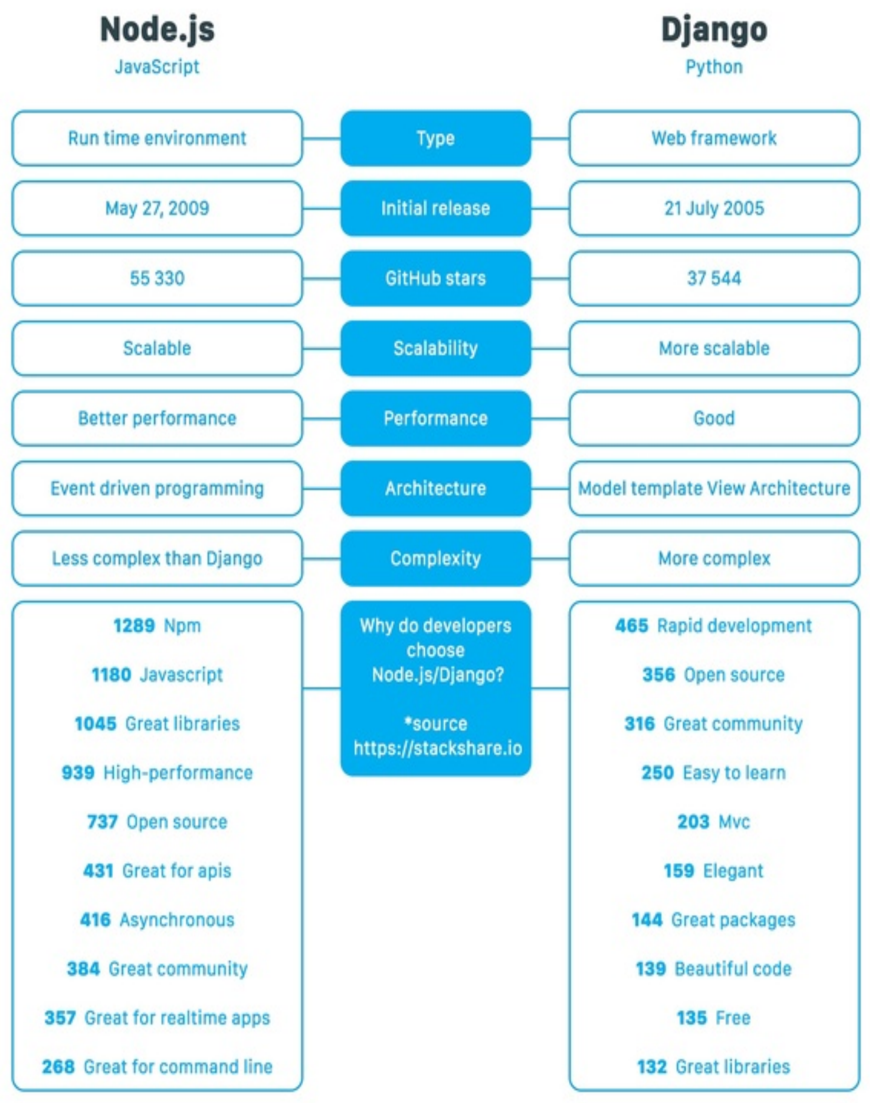
Appendix I - Self-Analysis

Throughout ELEC 490, the team has effectively collaborated and overcome any obstacles that were presented. The team faced the first major challenge immediately at the start of the year when a planned industry partnership was canceled by the client last minute. As a result of this, the team needed to quickly pivot to a new project in time for the first deliverable of the semester. The team worked hard and brainstormed a multitude of ideas which were then refined and quickly yielded the concept for TextNet. The team believes the issue was handled very well, however in the future performance can be improved by not designing systems with crucial external dependencies. Another challenge that the team faced was in ensuring that the code base is concise and organized in a way which is logical and modular. The team was able to quickly produce code that met most functional requirements, however it had many unnecessary redundancies and a confusing file structure. To fix this, the code base was refactored, and the team is very happy with the final result. In the future, the team can be more effective by preparing more detailed plans before splitting up to individually produce components. Another organizational mistake was that the team did not actually collect any money out of the budget from Queen's University budget due to complications with retrieval for online pay-as-you-go websites. As a result, the team paid a small amount of money out of their personal funds to cover these costs. In the future, the team can take more initiative in communicating and problem solving with external groups.

The team had several advantages that led to a successful project. Three members of the team have professional industry experience with software development as both product managers and software developers. This meant that the team understood best practices used by companies like Procter & Gamble and Microsoft and could incorporate these practices through the development lifecycle of the application. All team members have also participated in software focused extra-curriculars such as hackathons which exposed them to technologies and frameworks that aren't taught at Queen's University, such as Node.js, Express, Jest, Codecov, and third-party API use. The team is extremely proud of the cleanliness of the codebase and would encourage anyone who is interested to view more of it on GitHub.

Though there were several issues that arose as a result of the team's weaknesses, the team is proud of the overall effort and how well most aspects of teamwork were carried out. The team was able to consistently follow the project timeline plan to manage and deliver tasks without any critical last-minute issues. Due to the team managing time so effectively, performance issues were identified early and able to be fixed which yielded a finished product that the team believes is close to optimal. Team members all contributed in a meaningful way to the project and the team believes it would be even more effective in future projects as a result of the experience working together.

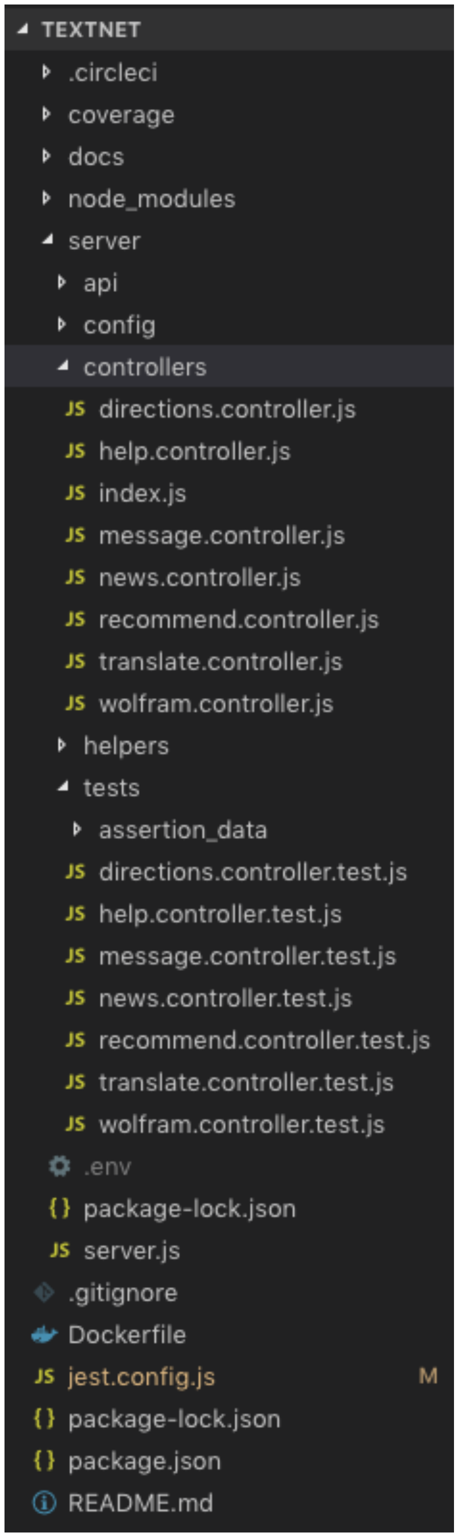
Appendix II – Framework Comparison (Node.js vs Django)



Node.js JavaScript		Django Python
Run time environment	Type	Web framework
May 27, 2009	Initial release	21 July 2005
55 330	GitHub stars	37 544
Scalable	Scalability	More scalable
Better performance	Performance	Good
Event driven programming	Architecture	Model template View Architecture
Less complex than Django	Complexity	More complex
1289 Npm **1180** Javascript **1045** Great libraries **939** High-performance **737** Open source **431** Great for apis **416** Asynchronous **384** Great community **357** Great for realtime apps **268** Great for command line	Why do developers choose Node.js/Django? *source https://stackshare.io	**465** Rapid development **356** Open source **316** Great community **250** Easy to learn **203** Mvc **159** Elegant **144** Great packages **139** Beautiful code **135** Free **132** Great libraries

Figure 21: Comparison of Node.js and Django

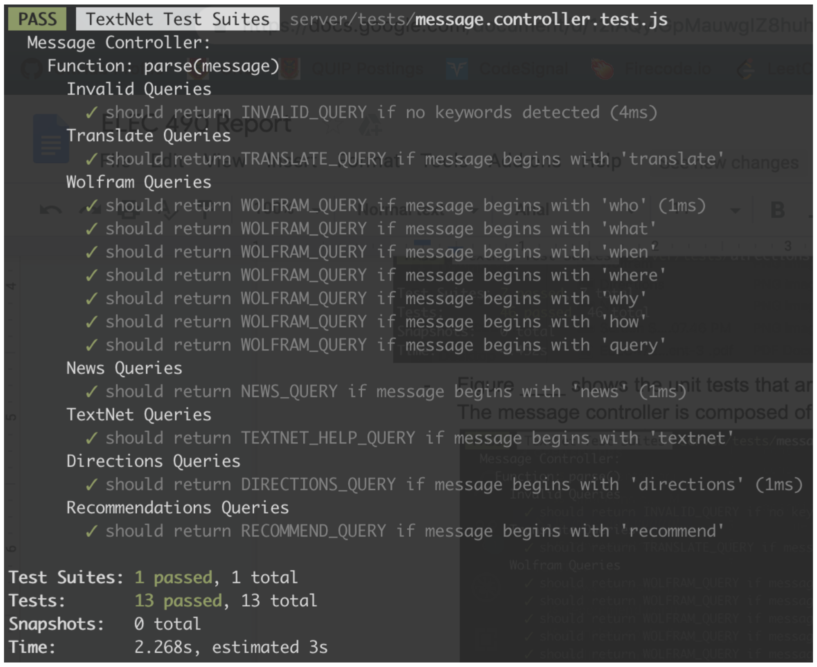
Appendix III - File Structure



```
TEXTNET
  .circleci
  coverage
  docs
  node_modules
  server
    api
    config
    controllers
      directions.controller.js
      help.controller.js
      index.js
      message.controller.js
      news.controller.js
      recommend.controller.js
      translate.controller.js
      wolfram.controller.js
    helpers
    tests
      assertion_data
      directions.controller.test.js
      help.controller.test.js
      message.controller.test.js
      news.controller.test.js
      recommend.controller.test.js
      translate.controller.test.js
      wolfram.controller.test.js
    .env
    package-lock.json
    server.js
  .gitignore
  Dockerfile
  jest.config.js                  M
  package-lock.json
  package.json
  README.md
```

Figure 22: Project Organization and Folder Structure

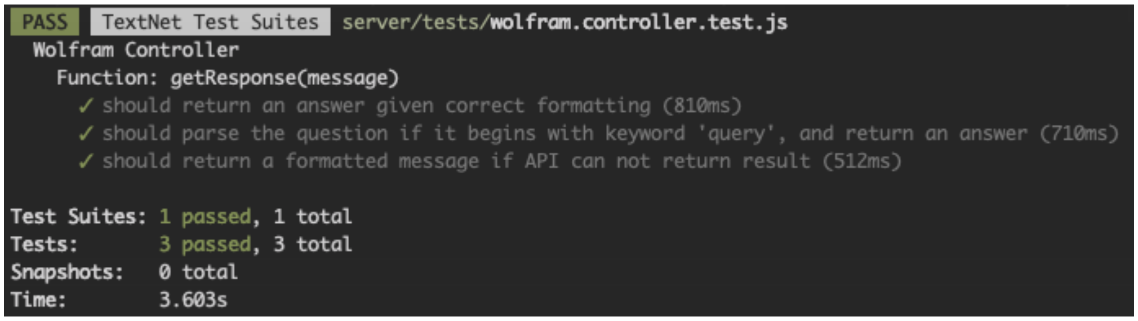
Appendix IV - Breakdown of Test Suites



```
PASS  TextNet Test Suites  server/tests/message.controller.test.js
  Message Controller:
    Function: parse(message)
      Invalid Queries
        ✓ should return INVALID_QUERY if no keywords detected (4ms)
      Translate Queries
        ✓ should return TRANSLATE_QUERY if message begins with 'translate'
      Wolfram Queries
        ✓ should return WOLFRAM_QUERY if message begins with 'who' (1ms)
        ✓ should return WOLFRAM_QUERY if message begins with 'what'
        ✓ should return WOLFRAM_QUERY if message begins with 'when'
        ✓ should return WOLFRAM_QUERY if message begins with 'where'
        ✓ should return WOLFRAM_QUERY if message begins with 'why'
        ✓ should return WOLFRAM_QUERY if message begins with 'how'
        ✓ should return WOLFRAM_QUERY if message begins with 'query'
      News Queries
        ✓ should return NEWS_QUERY if message begins with 'news' (1ms)
      TextNet Queries
        ✓ should return TEXTNET_HELP_QUERY if message begins with 'textnet'
      Directions Queries
        ✓ should return DIRECTIONS_QUERY if message begins with 'directions' (1ms)
      Recommendations Queries
        ✓ should return RECOMMEND_QUERY if message begins with 'recommend'

Test Suites: 1 passed, 1 total
Tests:       13 passed, 13 total
Snapshots:   0 total
Time:        2.268s, estimated 3s
```

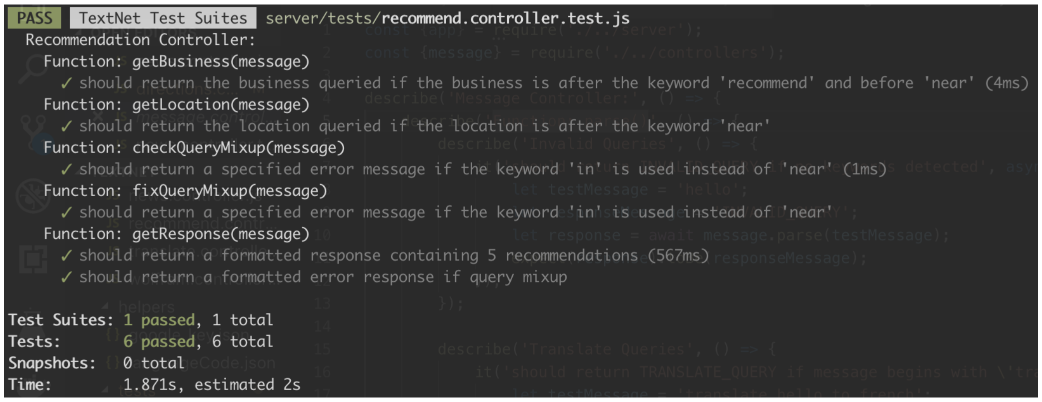
Figure 23: Message Controller Test Overview



```
PASS  TextNet Test Suites  server/tests/wolfram.controller.test.js
  Wolfram Controller
    Function: getResponse(message)
      ✓ should return an answer given correct formatting (810ms)
      ✓ should parse the question if it begins with keyword 'query', and return an answer (710ms)
      ✓ should return a formatted message if API can not return result (512ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        3.603s
```

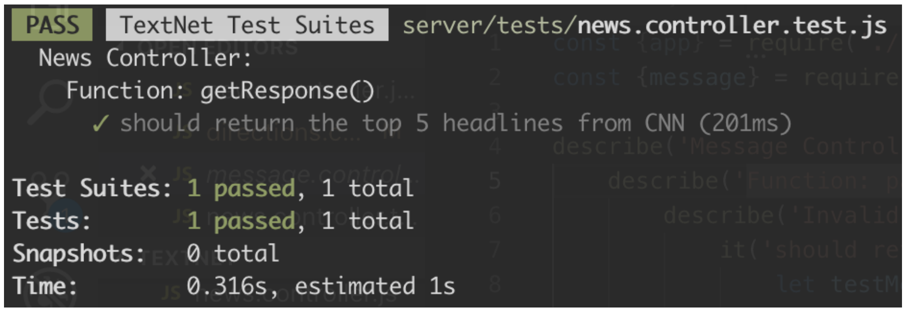
Figure 24: Wolfram Controller Test Overview



```
PASS  TextNet Test Suites  server/tests/recommend.controller.test.js
  Recommendation Controller:
    Function: getBusiness(message)
      ✓ should return the business queried if the business is after the keyword 'recommend' and before 'near' (4ms)
    Function: getLocation(message)
      ✓ should return the location queried if the location is after the keyword 'near'
    Function: checkQueryMixup(message)
      ✓ should return a specified error message if the keyword 'in' is used instead of 'near' (1ms)
    Function: fixQueryMixup(message)
      ✓ should return a specified error message if the keyword 'in' is used instead of 'near'
    Function: getResponse(message)
      ✓ should return a formatted response containing 5 recommendations (567ms)
      ✓ should return a formatted error response if query mixup

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        1.871s, estimated 2s
```

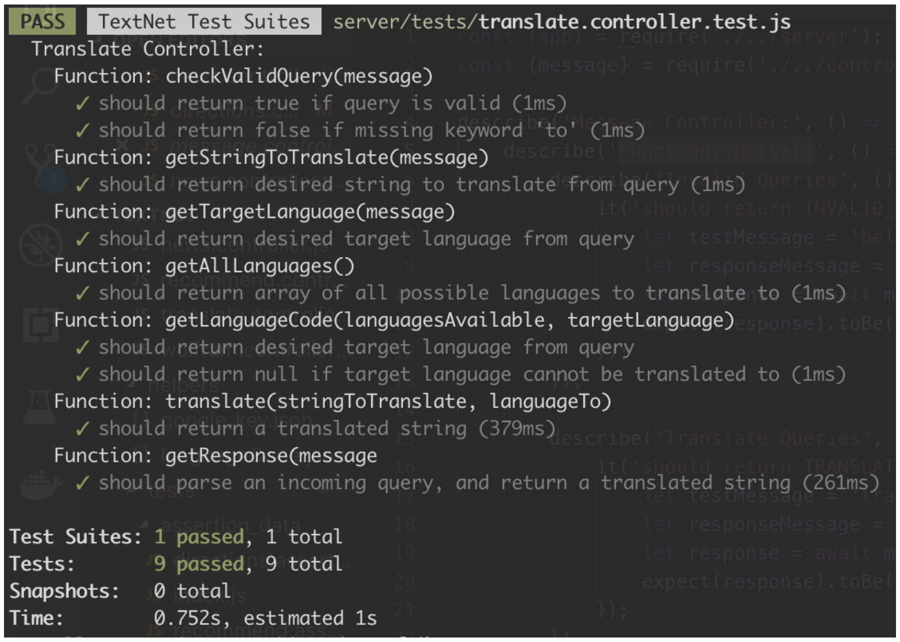
Figure 25: Recommend Controller Test Overview



```
PASS  TextNet Test Suites  server/tests/news.controller.test.js
  News Controller:
    Function: getResponse()
      ✓ should return the top 5 headlines from CNN (201ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.316s, estimated 1s
```

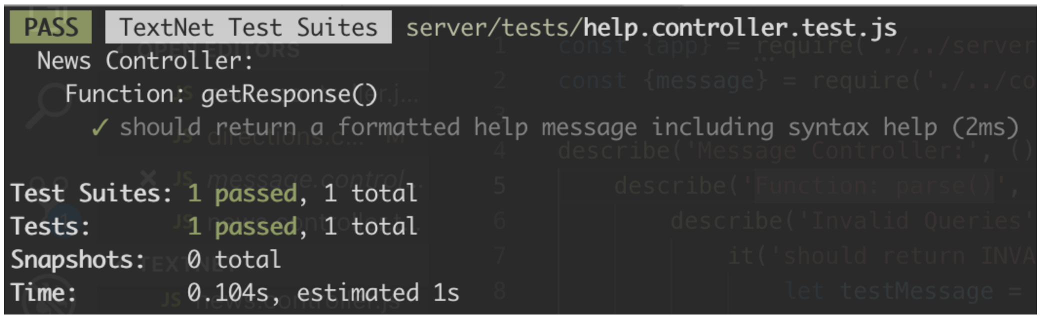
Figure 26: News Controller Test Overview



```
PASS  TextNet Test Suites  server/tests/translate.controller.test.js
  Translate Controller:
    Function: checkValidQuery(message)
      ✓ should return true if query is valid (1ms)
      ✓ should return false if missing keyword 'to' (1ms)
    Function: getStringToTranslate(message)
      ✓ should return desired string to translate from query (1ms)
    Function: getTargetLanguage(message)
      ✓ should return desired target language from query
    Function: getAllLanguages()
      ✓ should return array of all possible languages to translate to (1ms)
    Function: getLanguageCode(languagesAvailable, targetLanguage)
      ✓ should return desired target language from query
      ✓ should return null if target language cannot be translated to (1ms)
    Function: translate(stringToTranslate, languageTo)
      ✓ should return a translated string (379ms)
    Function: getResponse(message
      ✓ should parse an incoming query, and return a translated string (261ms)

Test Suites: 1 passed, 1 total
Tests:       9 passed, 9 total
Snapshots:   0 total
Time:        0.752s, estimated 1s
```

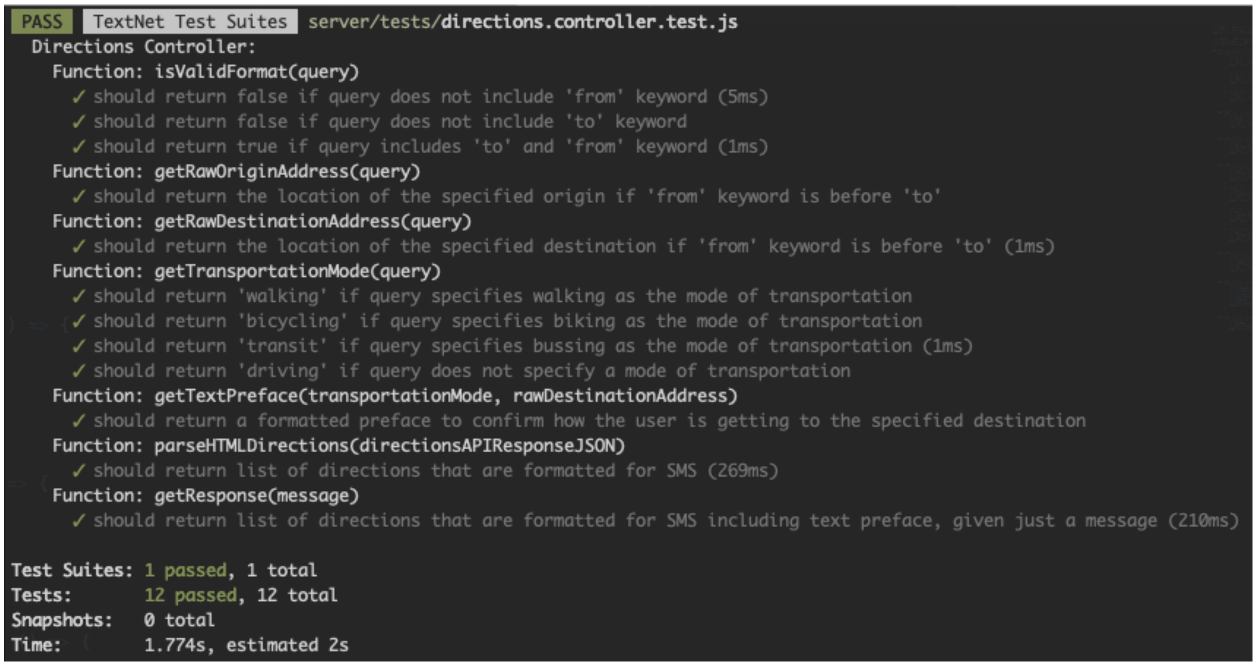
Figure 27: Translate Controller Test Overview



```
PASS  TextNet Test Suites  server/tests/help.controller.test.js
  News Controller:
    Function: getResponse()
      ✓ should return a formatted help message including syntax help (2ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.104s, estimated 1s
```

Figure 28: Help Controller Test Overview



```
PASS  TextNet Test Suites  server/tests/directions.controller.test.js
  Directions Controller:
    Function: isValidFormat(query)
      ✓ should return false if query does not include 'from' keyword (5ms)
      ✓ should return false if query does not include 'to' keyword
      ✓ should return true if query includes 'to' and 'from' keyword (1ms)
    Function: getRawOriginAddress(query)
      ✓ should return the location of the specified origin if 'from' keyword is before 'to'
    Function: getRawDestinationAddress(query)
      ✓ should return the location of the specified destination if 'from' keyword is before 'to' (1ms)
    Function: getTransportationMode(query)
      ✓ should return 'walking' if query specifies walking as the mode of transportation
      ✓ should return 'bicycling' if query specifies biking as the mode of transportation
      ✓ should return 'transit' if query specifies bussing as the mode of transportation (1ms)
      ✓ should return 'driving' if query does not specify a mode of transportation
    Function: getTextPreface(transportationMode, rawDestinationAddress)
      ✓ should return a formatted preface to confirm how the user is getting to the specified destination
    Function: parseHTMLDirections(directionsAPIResponseJSON)
      ✓ should return list of directions that are formatted for SMS (269ms)
    Function: getResponse(message)
      ✓ should return list of directions that are formatted for SMS including text preface, given just a message (210ms)

Test Suites: 1 passed, 1 total
Tests:       12 passed, 12 total
Snapshots:   0 total
Time:        1.774s, estimated 2s
```

Figure 29: Directions Controller Test Overview