

METHODOLOGY OF A SECURE INFORMATION EXCHANGE BETWEEN
A WIFI CAPABLE EMBEDDED COMPUTER AND A MOBILE DEVICE
USING A TCP/IP CONNECTION

BY

JOSEPH DOMENICK CIPOLLINA

MS, Binghamton University, 2012

BS, Clarkson University, 2009

TERMINATION PROJECT

Submitted in partial fulfillment of the requirements for
the degree of Master of Science in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2017

© Copyright by Joseph Domenick Cipollina 2017

All Rights Reserved

Accepted in partial fulfillment of the requirements for
the degree of Master of Science in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2017

December 15, 2017

Professor Leslie Lander, Advisor
Department of Computer Science, Binghamton University

ABSTRACT

This method is used to effectively share live updating information from the vehicle embedded computer to a mobile device. The method is effective by adding a light weight server application to the vehicle embedded computer and using a client mobile application capable of running on both iOS and Android. The mobile application, written in JavaScript and HTML, leveraged the cross-platform capability of Cordova in Visual Studio Community 2015. The light weight server was created using the open source Node.js program http-server. The vehicle embedded computer updates a JSON file with a program written in JavaScript to update the data values. This program can be used as a skeleton for any kind of data desired in future applications. At this time, the security comes from encryption of the private network at the facility where the application and vehicle are used. Additional security measures were explored but not fully implemented, such as a user authentication and SSL.

Table of Contents

ABSTRACT.....	vi
Table of Contents.....	vii
List of Figures	viii
Chapter 1: Introduction	1
1.1 Goals	1
1.2 Overview	1
Chapter 2: WIFI Capable Embedded Computer.....	2
2.1 Embedded Computer Selection and Setup.....	2
2.2 Creating a Light Weight Server	3
2.3 Updating Data Through Node.js	5
2.4 Security and SSL (Secure Socket Layer).....	5
Chapter 3: Cross-Platform Mobile Application.....	7
3.1 Integrated Development Environment (IDE)	7
3.2 Mobile Application Solution	8
3.3 Ripple Simulator.....	10
3.4 Deploy to Device	12
Chapter 4: Conclusion and Further Research	14
Appendix	16
A.1 – Setting up Debian Jessie and Node.js on a Raspberry Pi	16
A.2 – Visual Studio Community 2015 Setup.....	18
A.3 – Weather App GitHub Repository Link.....	20
A.4 – Project App GitHub Repository Link.....	20
A.5 – Google Share Drive.....	20
A.6 – Code for updatetime_cron.js	21
A.7 – Visual Studio Community Bug.....	22
References	23

List of Figures

Figure 1: HTTP GET response for lightweight sever public folder.	3
Figure 2: HTTP GET response for time.json file.	4
Figure 3: Server Log of HTTP GET Requests	4
Figure 4: http-server running with SSL being accessed	6
Figure 5: Ripple Simulator running the project application on an iphone5	10
Figure 6: Ripple simulator running the project application on a galaxy nexus.....	11
Figure 7: Ripple Simulator HTTP Requests Logged on Server	11
Figure 8: Project Application Screen Shots on Samsung Galaxy S5	12
Figure 9: Project Embedded Computer Server Log from Development Phone HTTP Requests....	13
Figure 10: Features Selected with Visual Studio Installation.....	19

Chapter 1: Introduction

The motivation for this project was to create a template for secure information exchange between a WIFI capable vehicle embedded computer and a mobile device. The use case is for technicians, operators, or engineers to be able to receive data from one of many possible vehicle embedded computers on their mobile device through the facility secure WIFI network supplied by the data logger base station.

1.1 Goals

- One common language across both the server and client, JavaScript
- The embedded computer needed to run either Debian Wheezy or Debian Jessie on an embedded microprocessor
- The mobile application would be available for both iOS and Android with one source code
- TCP/IP communication over the WIFI network at a facility
- Live updating data without human intervention
- Security and encryption of data

1.2 Overview

The following is divided up into three chapters, where the WIFI capable embedded computer goes into research and technical details on the emulated computer acting as the data logger. Chapter 3, cross-platform mobile application, describes the research and technical details on the cross-platform mobile application that could be installed on user phones. Conclusions and further research describes in detail the conclusions of how the project has met the goals above and gives suggestions for further research to improve the project.

Chapter 2: WIFI Capable Embedded Computer

The methodology for creating the server-side application was to first find a device able to meet the goals of the project. The device used as the WIFI capable embedded computer would need to run a light weight server and an application to update data shared by the server to simulate live data. The use case may have the WIFI network secure, but additional levels of security would be beneficial such as using SSL.

2.1 Embedded Computer Selection and Setup

The target vehicle embedded computer is running Debian Wheezy or Debian Jessie, so the development embedded computer needed to be capable of running this operating system to best emulate the use case environment. To simulate the target use case, the development embedded computer needed to be capable of being on the same network as the mobile device. Two cases were considered:

- Computers available in the Computer Science Computer Lab
- Raspberry Pi 3

Both the computers in the lab and the Raspberry Pi 3 were capable of running Debian Wheezy or Jessie. Both would require additional hardware to be capable of being on the same network. The computers in the lab would require an adaptor to connect to a mobile hotspot as to not have inference from the Binghamton University network. The Raspberry Pi 3 would need to be procured in general. The advantage of the Raspberry Pi 3 was the ability to leverage online community support and tutorials to implement the WIFI capable embedded computer.

The process of purchasing the Raspberry Pi 3 and setting it up with Raspbian Jessie, a version of Debian Jessie optimized for the Raspberry Pi, is outlined online in a tutorial [1]. Differences from the tutorial to the project and a high-level summary of the tutorial is available in Appendix A.1.

2.2 Creating a Light Weight Server

The method to create the light weight server was based on a tutorial from online [2]. The light weight server software is http-server from npm (Node Package Manager) [3]. Node package manager is an open source archive of JavaScript software. The default folder location was utilized, “public” in the directory it is started. The public folder was placed in the user home directory, so to run the server the command below needs to be ran in the user home directory. The default port is 8080, but normal http requests go through port 80, so on startup the port parameter is used.

```
$ sudo http-server -p 80
```

The server looks for an index.html file in the public folder by default, but if it is not present it will share the public folder. If the application is not able to find the public folder, the default, it will instead share the directory it was ran in. To ensure the http-server only shares the public folder, run it in the parent folder of the “public” folder. The HTTP GET request for the public folder through a chrome browser is shown in Figure 1.

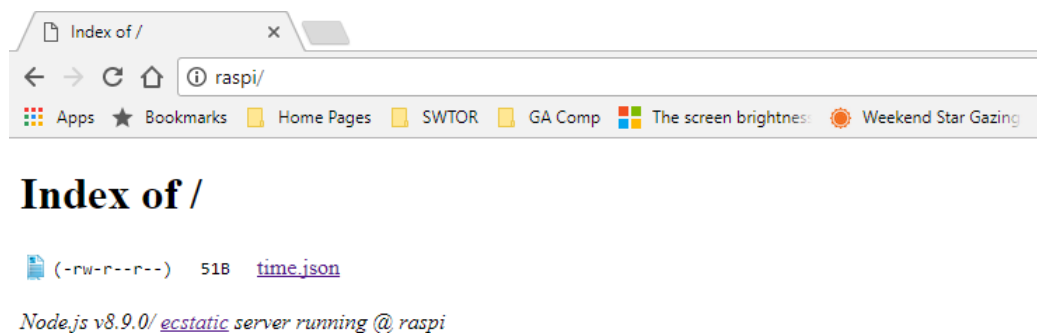


Figure 1: HTTP GET response for lightweight sever public folder.

For this project, the public folder has no html files but instead JSON files. Each JSON file would be specific to the type of data it lists. The live data that is sent for a demo of the project is the time (hour, minute, second), stored in a JSON file called time.json and is shared through the HTTP GET requests. The HTTP GET request for the time.json file through a chrome browser is shown in Figure 2. The log for these requests from the server is shown in Figure 3.

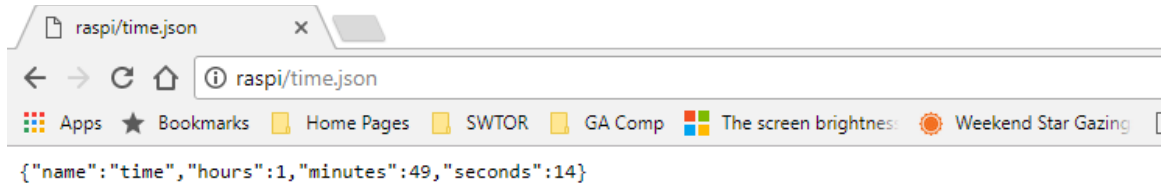


Figure 2: HTTP GET response for time.json file.

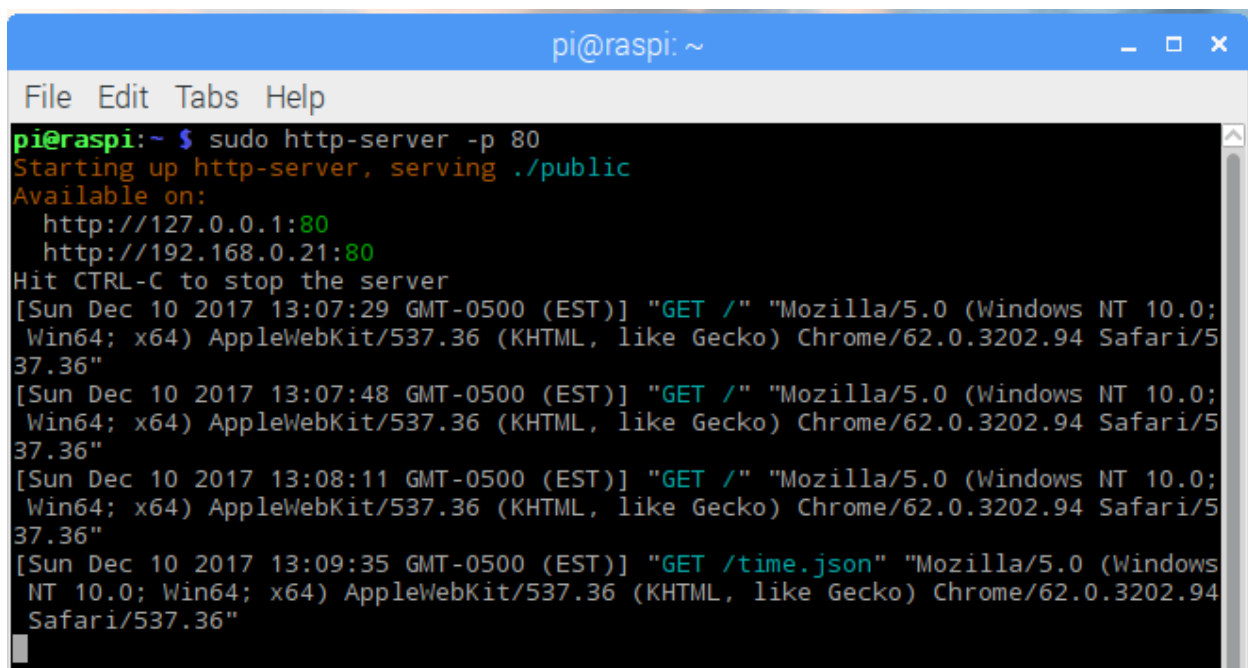


Figure 3: Server Log of HTTP GET Requests

The public folder and the time.json file are available in the latest Raspberry Pi snapshot on the Google drive, Appendix A.5.

2.3 Updating Data Through Node.js

To achieve live updating data the embedded computer needed the shared data, time (hour, minute, second) stored in time.json, to be updating. This action was done with a simple Node.js application created to update the time.json file with the current time every two (2) seconds. The periodic change to the time.json file was achieved using cron from npm [4]. Updating the JSON file was done with JSON.stringify and fs.write, both of which are standard built-in objects of Node.js [5]. The relationship of the application and JSON file are hardcoded into the application based on relative location. To ensure the application can find the JSON file the application needs to be in a subfolder of the user home directory. For the project “apps” was used. Run this application with the following command in the apps folder under the user home directory:

```
$ node updatetime_cron.js
```

The code for this application is available in the latest Raspberry Pi snapshot on the Google drive, Appendix A.5, or can be directly viewed in Appendix A.6.

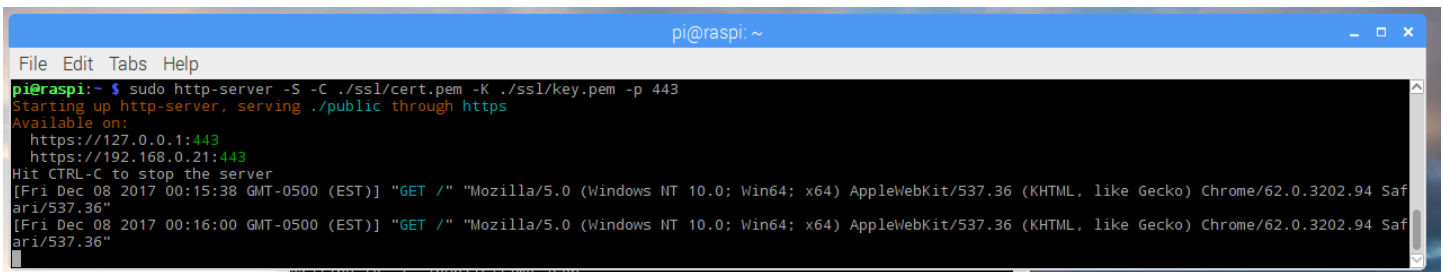
2.4 Security and SSL (Secure Socket Layer)

The http-server application is capable of SSL and the feature was implemented for self-signed certificates using the OpenSSL toolkit. The http-server requires a certificate and key stored in an cert.pem and key.pem file, respectively [3]. To create these files a tutorial online gives the process for both authority-signed and self-signed certificate requests [6]. While answering the question from the toolset the important answer is the common name and was answered with “raspi”, the device name. During testing, this did not seem as important when performing the HTTP request, because the result was the same using <https://raspi> and <https://192.168.0.21>. The cert.pem and key.pem files were placed in a folder called “ssl” under the user directory. The default port for http-server is 8080 and for a proper HTTPS request, it

requires port 443. To start the http-server application, with SSL and on port 443, the following command should be run in the user folder:

```
$ sudo http-server -S -C ./ssl/cert.pem -K ./ssl/key.pem -p 443
```

The http-server application running with SSL and being accessed by a chrome browser can be seen in Figure 4. Copies of the cert.pem and key.pem are available in the latest Raspberry Pi snapshot on the Google drive, Appendix A.5.



```
pi@raspi: ~  
File Edit Tabs Help  
pi@raspi:~$ sudo http-server -S -C ./ssl/cert.pem -K ./ssl/key.pem -p 443  
Starting up http-server, serving ./public through https  
Available on:  
  https://127.0.0.1:443  
  https://192.168.0.21:443  
Hit CTRL-C to stop the server  
[Fri Dec 08 2017 00:15:38 GMT-0500 (EST)] "GET /" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36"  
[Fri Dec 08 2017 00:16:00 GMT-0500 (EST)] "GET /" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36"
```

Figure 4: http-server running with SSL being accessed

Using the self-signed certificate is suited for the project use case, because the server is not desired to be accessible outside of the facility network. Yet, an Android OS is not as accepting of a self-signed certificate as a browser and the application needs to verify the certificate through a method such as certificate pinning [7]. Certificate pinning can also prevent man-in-the-middle attacks [8]. The certificate can be pinned as long as the production source code is not made public and has a different certificate and key from the project. Plugins are available to Visual Studio Apache Cordova for pinned certificate checking [9].

Another lightweight server application was researched for additional security. The json-server is based on Express.js and uses the middleware web framework that allows for a series of function calls, which allows for easy expansion of the application for user authentication [10][11].

Chapter 3: Cross-Platform Mobile Application

The methodology for creating the cross-platform mobile application was to first find and install an integrated development environment (IDE) that would support the goals of the project. A tutorial for the IDE was then leveraged to create the desired application for the project. The tutorial also gave instructions on how to use the Ripple Simulator for both Android and iOS. Another tutorial was used to deploy the completed application to the mobile device.

3.1 Integrated Development Environment (IDE)

Based on the goal of one set of source code for both iOS and Android, the mobile application needed to be cross-platform. Also, based on the requirement of programming in JavaScript the cross-platform had to be predominately written in JavaScript. Through research a few integrated development environments (IDEs) were found that supported these requirements:

- Meteor [12]
- Apache Cordova through Visual Studio [13]

Visual Studio was picked for the larger resources of open-source libraries, plug-ins, and NuGet Packages. Also, experience with the IDE and availability of Visual Studio Community 2015 was weighted as an advantage. The general setup of Visual Studios and toolsets are available in Appendix A.2.

During the project a bug in Visual Studios became apparent, where files from the solution would be removed upon saving. This bug in Visual Studios was not confined this project and additional details, including potential fixes and workarounds can be found in Appendix A.7.

3.2 Mobile Application Solution

The project mobile application was based on a tutorial for a weather app [15]. The tutorial was followed fully as a first attempt, with the exception of emulation due to not having the necessary hardware or supporting programs. The weather application source code is available through the link in Appendix A.3 and the project application source code is available through the link in Appendix A.4. The project application was created by selectively following the weather app tutorial.

When creating the project application, the Apache Cordova Apps, JavaScript template from Visual Studio was used. The merges folder stores files that are specific to Android or iOS and the www folder stores all common files between the two operating systems. The project uses the same jQuery 3.2.1 and jQuery.Mobile 1.4.5 from NuGet as the weather app. Files created for these programs needed to be moved into the www directory, same as the weather app. The index.html and index.css in the www directory were edited the same way to utilize the same styling and add the use of specific JavaScript files. JavaScript files were named differently to reflect the new application, but the code for performing the HTTP GET request was very similar. In the data.js file, instead of weather.js, the functions were renamed, queryString was edited, and the JSON parsing function was updated to reflect the new application. In the index.html file CSP had to be removed due to the project use case having each WIFI capable embedded computer being a different domain to access. At this point the program was able to perform an HTTP GET request, receive, and parse the data from the raspberry pi with commit 4dcffd7 in the project application repository, Appendix A.4.

With the ability to perform an HTTP GET request and parse the JSON data, the JavaScript function, `getDataFromDataLogger()`, was updated to perform the request periodically

using the setInterval object [16]. Since this function would only connect to the raspberry pi, another function was created to disconnect from it, using the clearInterval object. For the disconnect function to work the variable referencing the setInterval needed to be global. Each function had their own button and each button would be hidden or shown depending on the state of the application. At this point the program would periodically loop through the code to perform the HTTP GET request, but the data would not update from the initial set with commit 2d8f060 in the project application repository, Appendix A.4.

This issue was initially classified as keeping stale data because the server would not show additional HTTP GET requests, but the code would loop through the function periodically as expected. This stale data would continue through closing and reopening the application. Removing the periodic part of the function and making the request manual did not fix the issue of stale data. Yet, a browser was still able to get updated data consistently. These behaviors required narrowing down where the issue lied. During the debug process Wireshark was used to snoop on the network and see if the HTTP GET requests were actually being sent and it confirmed that the requests were not being sent from the computer. Since the weather app was able to manually get live data consistently the weather app was run in debug mode and just before performing the request the queryString value would be updated to the raspberry pi. It was found that, the program would show http error 404 multiple times but once the data was found the application would stop sending HTTP requests for the same queryString value. The function used to perform the HTTP request (.getJSON), was researched. Research found that it could cache the data and to disable it ajax had to be setup to never cache [17]. Once ajax was setup to not cache globally, the application worked as expected with one (1) second periodic HTTP GET requests to the embedded computer server and live data updating on the application screen

without human intervention. The latest version of the project application has this update and is available in the project application repository, Appendix A.4.

3.3 Ripple Simulator

The process for using the Visual Studio Ripple simulator for Android and iOS is covered in the weather app tutorial [15]. The project application performs periodic HTTP request on both an iOS device, as shown in Figure 5, and Android device, as shown in Figure 6.

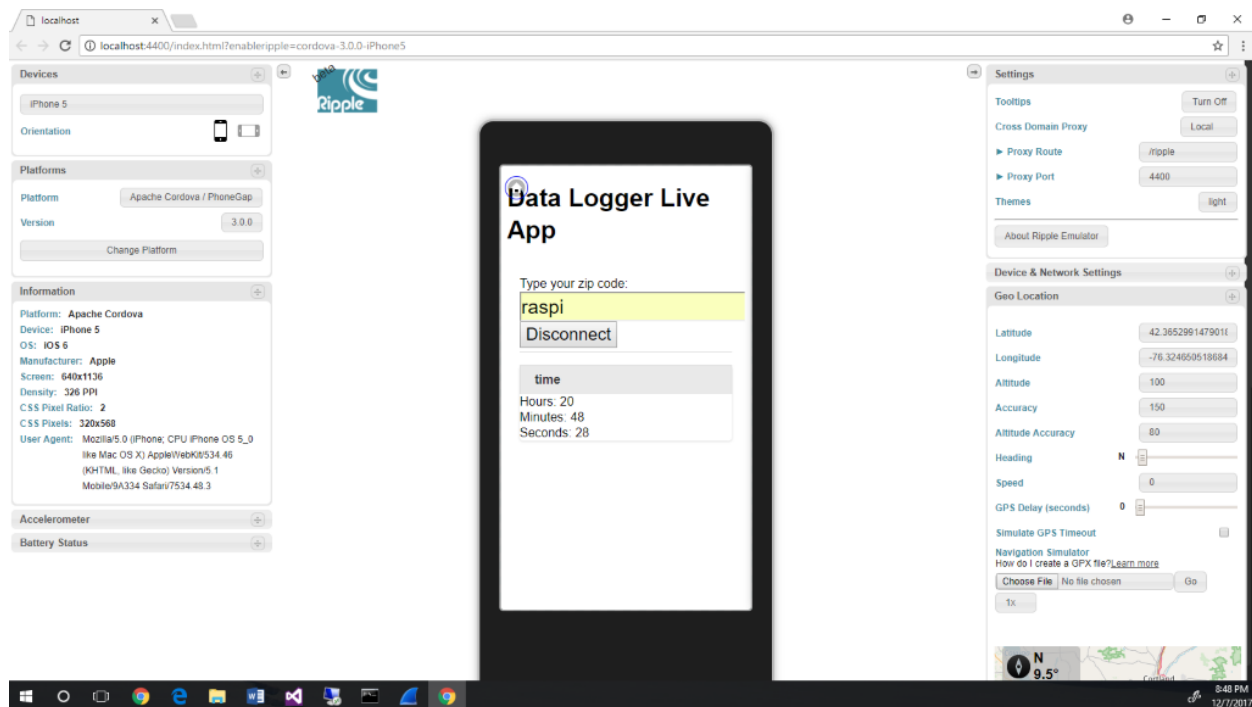


Figure 5: Ripple Simulator running the project application on an iphone5

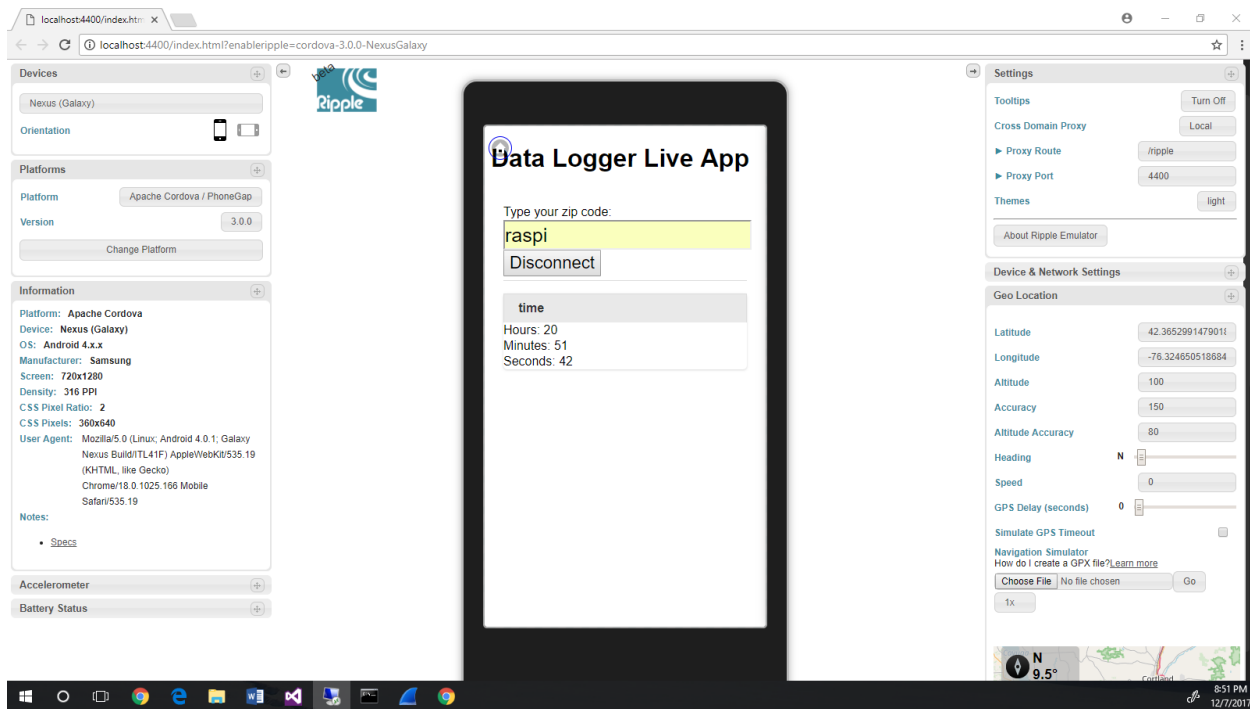


Figure 6: Ripple simulator running the project application on a galaxy nexus

While simulating the application it was noticed that the installation of Samba on the Raspberry pi allowed for the Ripple Simulator to perform an HTTP call to the Raspberry Pi using its device name in addition to the IP address. The requests shown in Figure 6 were logged by the server and can be seen in Figure 7 as one (1) second period requests, as expected.

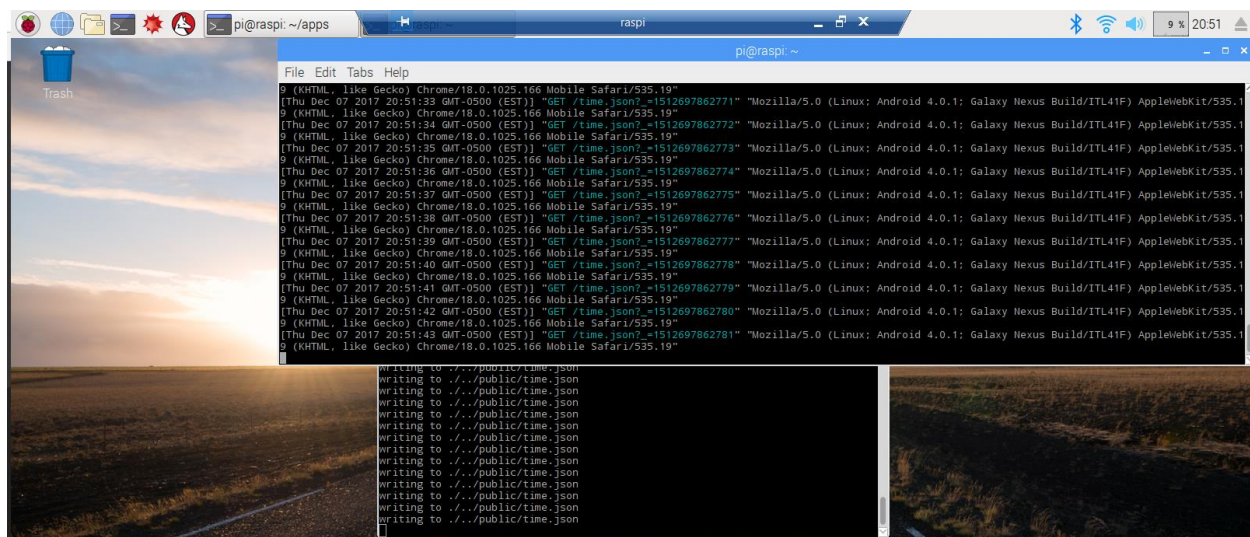


Figure 7: Ripple Simulator HTTP Requests Logged on Server

3.4 Deploy to Device

The project application is deployed to the development phone, a Samsung Galaxy S5 with Android 6.0.1, with ease using Visual Studio Community 2015 and the process outline at the end of an online tutorial [18]. Deployment to a phone with Android 6.0.1 requires the Android SDK (API Level 23) support files. A Samsung Galaxy S5, model number SM-G900V, required Samsung USB driver version 1.5.45.00 [19]. For this project an iOS phone was not available without purchase and deployment for an iOS phone requires a Mac computer, physical or cloud hosted [20].

The application deployed to the development phone without issue and can be seen in Figure 8 working with the embedded computer's IP address, but not with the device name. At the same time the log from the server can be seen in Figure 9. The application likely does not recognize the device name as the domain of the embedded computer's server due to being Android and not Windows. This shows how simulation can only go so far.

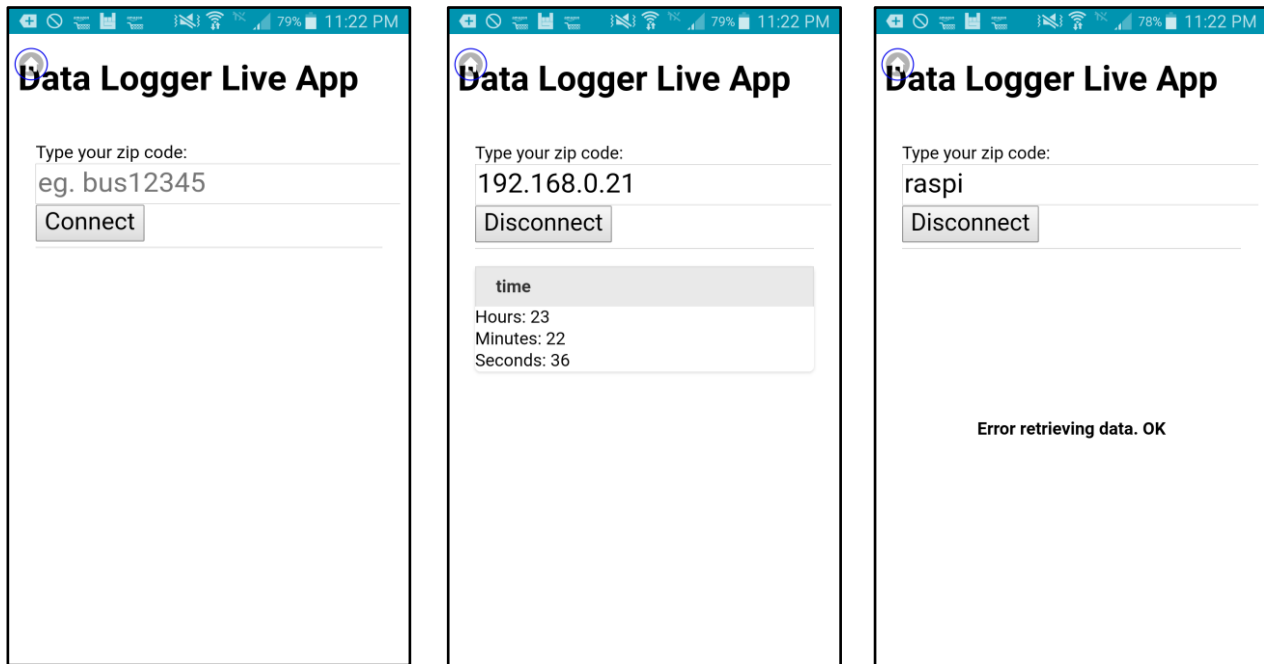


Figure 8: Project Application Screen Shots on Samsung Galaxy S5

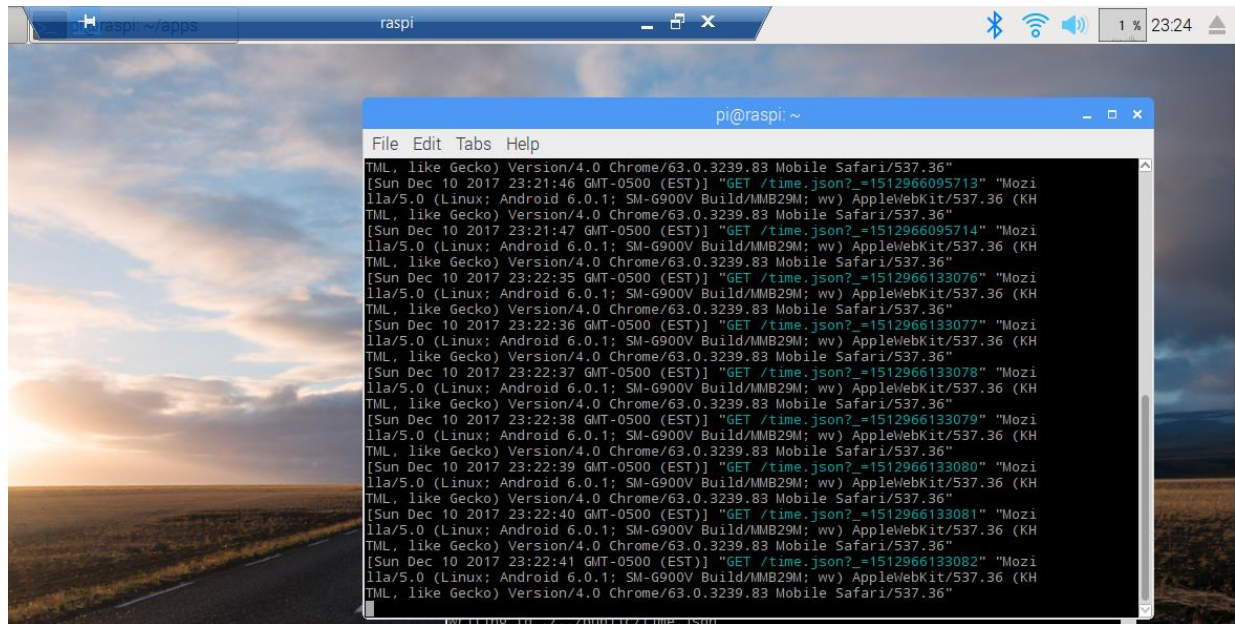


Figure 9: Project Embedded Computer Server Log from Development Phone HTTP Requests

After deployment the application is installed and the phone doesn't need to be connected to the computer unless debug is desired. A video of the deployed application running without the cable and receiving live updating data without human intervention is available on the Google drive, Appendix A.5, named [20171128_000156_App_Deployed_Live_Data.mp4](#).

Chapter 4: Conclusion and Further Research

The mobile application and embedded computer were able to meet all goals, outlined in chapter 1, with varying levels of success. The Raspberry Pi, WIFI capable embedded computer, applications, both http-server and updatetime_cron, are written in JavaScript. The cross-platform mobile application is written in JavaScript, excluding the front end being written in HTML. The WIFI capable embedded computer runs Raspbian Jessie V1.3, which is Debian Jessie optimized for the Raspberry Pi. The mobile application is able to run on Android 6.0.1 and receive updating live data from the WIFI capable embedded computer without issue. The mobile application is sending a HTTP request and the embedded computer is responding following TCP and using IPv4. The project security and encryption of the communication is only as safe as the facility network security and encryption. To improve the security and encryption SSL was partially implemented and a plan going forward was developed.

Further research could include additional security, additional encryption, connecting to the embedded computer via its device name, and deployment on an iOS mobile device. Additional security for the data can be achieved by changing the server application to json-server and expanding the program to check for a user authentication with each the HTTP request, leveraging the middleware web framework. Additional encryption could be added to protect the authentication and actual data with SSL, but would need to implement a plugin capable of a dynamic domain for multiple servers with the same certificate and key to meet the project use case. The application running through the Ripple Simulator is capable of sending the HTTP request based on the embedded computer device name, but not when running on the Android

mobile device. The necessary plugin or code for the application to alleviate this could be integrated into the application. Deployment on an iOS mobile device could be done with a cloud hosted compiler, such as Travis CI, or a Mac as explained in the tutorial [20].

Appendix

A.1 – Setting up Debian Jessie and Node.js on a Raspberry Pi

Setting up a Raspberry Pi 3 with Raspbian Jessie, a version of Debian Jessie optimized for the Raspberry Pi, is written out in a tutorial online [1]. Following the tutorial gave the Raspberry Pi 3 the ability to have a wireless setup, windows file share, and Node.js, beyond the normal configuration of the embedded computer. There were a few differences between the tutorial and the project.

A difference early on between the tutorial and this project was that the Raspberry Pi came with NOOBS (New Out Of the Box Software), which allowed the SD Card to not need imaging via another computer, similar to having a boot loader. During the project the latest version of Raspbian was Stretch, one version newer than Jessie, but NOOBS came preloaded with Raspbian Jessie version 1.2. If future versions did not have NOOBS with Raspbian Jessie, an archived version of Raspbian Jessie would have to be written as an image to the SD card as described in the tutorial. Another difference was during the configuration setup, where I didn't have an option to expand the file system, but instead already had more than 20 GB of free space seen by the operating system.

When updating Raspbian Jessie, it went to Release 1.3 and was used for the rest of the project.

The wireless setup, excluding the power cord, gave the ability to work on the Raspberry Pi 3 without an HDMI display cable, USB keyboard, or USB mouse. Note that you would still need a keyboard, mouse, and HDMI display to perform the initial setup and connect to your WIFI. The basic concept was to create remote desktop access from devices on your wireless

router. It was suggested to use xrdp over the preinstalled RealVNC because of the tediousness of manually adjusting screen sizes. Prior to installing xrdp, RealVNC first needed to be removed by installing tightvncserver. After removing RealVNC and installing xrdp the Raspberry Pi was capable of having remote desktop access using the IP address.

Being that most networks give IP address dynamically and avoid a first step of determining the IP address the device name of the Raspberry Pi 3 could be used with windows remote desktop. To accomplish this, Samba needed to be installed so that the Raspberry Pi would follow the necessary protocol for device name based remote desktop access. One additional step that was needed, but not listed in the tutorial, was to make the network that the Raspberry Pi is connected to and the computer trying to access it through have the wireless network as a home network setting, so that the Windows computer is discoverable. Samba also gives the capability for file sharing folders.

When installing Node.js, it was important to ensure that the processor was ARMv7 or ARMv8 for the particular version of Node.js suggested. The project used an archived version of Node.js, 8.9.0 instead of the latest. The installation was also specific to Debian. More information on this version of Node.js is available at:

- https://deb.nodesource.com/setup_8.x

Archived versions of Raspbian are available at:

- <http://downloads.raspberrypi.org/raspbian/images/>

The SD card was backed up as an image out of the box, NOOBS preloaded with Raspbian Jessie, using Win32 Disk Imager and is available on the Google Share Drive, Appendix A.5, folder under Raspberry Pi.

A.2 – Visual Studio Community 2015 Setup

Prior to installation of Visual Studio Community 2015, ensure you have installed Java.

The version used here was JDK1.8.0_151-windows-x64. This is required to install Android SDK files. If you receive an error for installing Android SDK files, you need to install Java.

The version of the IDE and toolsets used in the project are given in Table 1. Apache Cordova uses Node.js and the version was V0.12 for the windows computer with Visual Studio.

Table 1: Visual Studio and Toolset Versions

Microsoft Visual Studio Community 2015	Version 14.0.25431.01 Update 3
Microsoft .NET Framework	Version 4.7.02556
Visual Basic 2015	00322-20000-00000-AA320
Visual C# 2015	00322-20000-00000-AA320
Visual C++ 2015	00322-20000-00000-AA320
Windows Phone SDK 8.0 - ENU	00322-20000-00000-AA320
Application Insights Tools for Visual Studio Package	7.0.20622.1
ASP.NET and Web Tools 2015.1 (Beta8)	14.1.11107.0
ASP.NET Web Frameworks and Tools 2012.2	4.1.41102.0
ASP.NET Web Frameworks and Tools 2013	5.2.40314.0
Clang with Microsoft CodeGen	14.0.25516
Command Bus, Event Stream and Async Manager	Merq
Common Azure Tools	1.8
GitHub.VisualStudio	2.3.6.391
JavaScript Language Service	2.0
JavaScript Project System	2.0
KofePackagePackage Extension	1.0
Microsoft Azure Mobile Services Tools	1.4
Microsoft MI-Based Debugger	1.0
NuGet Package Manager	3.4.4
PreEmptive Analytics Visualizer	1.2
SQL Server Data Tools	14.0.60519.0
TypeScript	1.8.36.0
Visual C++ for Cross Platform Mobile Development (Android)	14.0.25401.00
Visual C++ for Cross Platform Mobile Development (iOS)	14.0.25401.00
Visual Studio Tools for Apache Cordova	Update 10
Xamarin	4.2.1.62 (680125b)
Xamarin.Android	7.0.2.37 (ce955cc)
Xamarin.iOS	10.2.1.5 (44931ae)

The installation file for Visual Studio Community 2015 was received from MSDN Alliance from a Binghamton University Student account. It is also available from Microsoft [14].

The features selected during setup or later modified to are shown in Figure 10. Note that Android SDK setup (API Level 23) was installed manually through the SDK manager application. Because of manual installation visual studios fails to install it normally and will actually remove supporting files. If not installed through the SDK manager (and Java is installed), it is expected that this feature can be installed in the initial installation without problems.

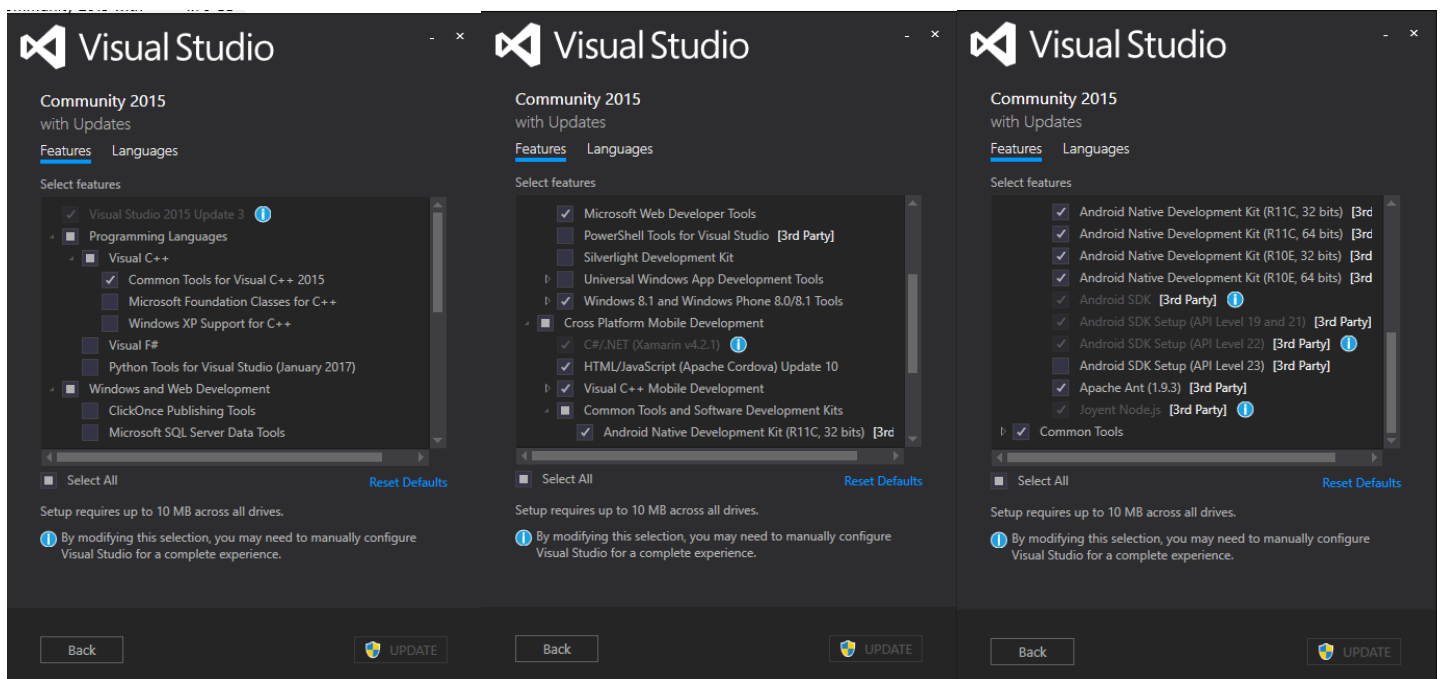


Figure 10: Features Selected with Visual Studio Installation

A.3 – Weather App GitHub Repository Link

<https://github.com/jcipoll1/CrossPlatformPhoneAppV02>

A.4 – Project App GitHub Repository Link

<https://github.com/jcipoll1/DataLoggerLiveApp>

A.5 – Google Share Drive

<https://drive.google.com/open?id=0B32N9zg2MPSULUZCMnRhX0JKTWM>

A.6 – Code for updatetime_cron.js

```
var fs = require('fs');
var filename = '../public/time.json';
var file = require(filename);
var cron = require('cron');

var d = new Date();

var jobupdatetime = new cron.CronJob({
  cronTime: '*/2 * * * *',
  onTick: function () {
    d = new Date();
    file.name = 'time'
    file.hours = d.getHours();
    file.minutes = d.getMinutes();
    file.seconds = d.getSeconds();

    //Input data
    fs.writeFile(filename, JSON.stringify(file), function (err) {
      if (err) return console.log(err);
      console.log('writing to ' + filename);
    });
  },
  start: false
});

jobupdatetime.start();
```

A.7 – Visual Studio Community Bug

After a Windows update a bug in Visual Studio was found that would cause the active file on screen to be removed from the solution if the project was saved, even without an edit to be saved. A stackoverflow post was made that described the same behaviors of my Visual Studio solution and a solution was given near the end of the project [21]. Repairing and modifying Visual Studio did not fix the problem and the solution given by the stackoverflow user requires a restore point prior to the update, which is not available. Until another solution can be found the process that was followed was:

- Make all edits and Save
- Manually add file to solution (right click on folder where it belongs)
- Attempt simulation. If pulled into simulation stop here, else continue.
- Commit and Push to GitHub
- Zip and delete file directory (or just use new one in later step)
- Go to the repository at GitHub in a browser
- In the browser, clone with visual studio option
- Save locally
- Reopen in VS administrator mode and perform either ripple or deploy

References

- [1] D. Johnson, "Guide to Installing Node.js on a Raspberry Pi | thisDaveJ", Thisdavej.com, 2017. [Online]. Available: <http://thisdavej.com/beginners-guide-to-installing-node-js-on-a-raspberry-pi/>. [Accessed: 30- Oct- 2017].
- [2] D. Johnson, "Create a Web Server in Node.js without any Code | thisDaveJ", Thisdavej.com, 2017. [Online]. Available: <http://thisdavej.com/create-a-web-server-in-node-without-any-code/>. [Accessed: 30- Oct- 2017].
- [3] C. Robbins, "http-server", npm, 2017. [Online]. Available: <https://www.npmjs.com/package/http-server>. [Accessed: 06- Nov- 2017].
- [4] N. Campbell, "cron", npm, 2017. [Online]. Available: <https://www.npmjs.com/package/cron>. [Accessed: 12- Nov- 2017].
- [5] S. Krasnianski, "How to update a value in a json file and save it through node.js", Stackoverflow.com, 2017. [Online]. Available: <https://stackoverflow.com/questions/10685998/how-to-update-a-value-in-a-json-file-and-save-it-through-node-js>. [Accessed: 12- Nov- 2017].
- [6] M. Kosh, "How to SSL - OpenSSL tips and common commands", How2ssl.com, 2017. [Online]. Available: http://how2ssl.com/articles/openssl_commands_and_tips/. [Accessed: 12- Nov- 2017].
- [7] "Android security - Implementation of Self-signed SSL certificate for your App. - CodeProject", Codeproject.com, 2017. [Online]. Available: <https://www.codeproject.com/Articles/826045/Android-security-Implementation-of-Self-signed-SSL>. [Accessed: 28- Nov- 2017].

- [8] R. Dasgupta, "Certificate pinning for Android and iOS: Mobile man-in-the-middle...", NowSecure, 2017. [Online]. Available:
<https://www.nowsecure.com/blog/2017/06/15/certificate-pinning-for-android-and-ios-mobile-man-in-the-middle-attack-prevention/>. [Accessed: 28- Nov- 2017].
- [9] C. Lantz, "Securely transmit data", Taco.visualstudio.com, 2017. [Online]. Available:
<http://taco.visualstudio.com/en-us/docs/cordova-security-xmit/#certificate-pinning>.
[Accessed: 08- Dec- 2017].
- [10] "typicode/json-server", GitHub, 2017. [Online]. Available:
<https://github.com/typicode/json-server>. [Accessed: 12- Nov- 2017].
- [11] "Using Express middleware", Expressjs.com, 2017. [Online]. Available:
<http://expressjs.com/en/guide/using-middleware.html>. [Accessed: 12- Nov- 2017].
- [12] "Build Apps with JavaScript | Meteor", Meteor.com, 2016. [Online]. Available:
<https://www.meteor.com/>. [Accessed: 08- May- 2016].
- [13] "Cross-Platform Mobile Development in Visual Studio", Msdn.microsoft.com, 2017.
[Online]. Available: <https://msdn.microsoft.com/en-us/library/dn771552.aspx>. [Accessed:
02- Oct- 2017].
- [14] L. Webster, "Download Older Visual Studio Software | Visual Studio", Visual Studio,
2017. [Online]. Available: <https://www.visualstudio.com/vs/older-downloads/>.
[Accessed: 07- Dec- 2017].
- [15] J. Wargo, "Get started with Visual Studio Tools for Apache Cordova | Cordova",
Taco.visualstudio.com, 2017. [Online]. Available: <http://taco.visualstudio.com/en-us/docs/get-started-first-mobile-app/>. [Accessed: 29- Oct- 2017].

- [16] "Is there any way to call a function periodically in JavaScript?", Stackoverflow.com, 2017. [Online]. Available: <https://stackoverflow.com/questions/1224463/is-there-any-way-to-call-a-function-periodically-in-javascript>. [Accessed: 17- Nov- 2017].
- [17] C. Salvado, "jQuery \$.getJSON works only once for each control. Doesn't reach the server again", Stackoverflow.com, 2017. [Online]. Available: <https://stackoverflow.com/questions/1345277/jquery-getjson-works-only-once-for-each-control-doesnt-reach-the-server-again>. [Accessed: 27- Nov- 2017].
- [18] M. Jones, "Run Your Apache Cordova App on Android | Cordova", Taco.visualstudio.com, 2017. [Online]. Available: <http://taco.visualstudio.com/en-us/docs/run-app-apache/>. [Accessed: 31- Oct- 2017].
- [19] "Install OEM USB Drivers | Android Studio", Developer.android.com, 2017. [Online]. Available: <https://developer.android.com/studio/run/oem-usb.html#Drivers>. [Accessed: 01- Nov- 2017].
- [20] J. Matthiesen, "Setup guide: Target iOS mobile devices in a Visual Studio Tools for Apache Cordova project | Cordova", Taco.visualstudio.com, 2017. [Online]. Available: <http://taco.visualstudio.com/en-us/docs/ios-guide/>. [Accessed: 29- Oct- 2017].
- [21] "files disappear from VS 2015 solution after saving", Stackoverflow.com, 2017. [Online]. Available: <https://stackoverflow.com/questions/47208157/files-disappear-from-vs-2015-solution-after-saving>. [Accessed: 10- Dec- 2017].