PropNets Visualizer: A web based tool to visualize information propagation in networks

G. V. M. P. A. Fernando
Department of Computer Science and Engineering, University of Moratuwa, Sri Lanka
pubudu.12@cse.mrt.ac.lk

***Abstract* - Data visualization and analysis has become vital in both knowledge discovery and presenting and explaining that knowledge to others. As such, the goal of this project was to implement a web based tool for the researchers studying propagation of information using network-based models. The tool was designed with ease of use and extensibility in mind and as such, it is a simple yet powerful tool which allows researchers to generate dynamic visualizations of network flow data they are working with. It was developed as a Polymer project with the intention of publishing it as a reusable web component. The end result of this is a simple but powerful visualization tool for researchers and an easy to integrate web component for the developers of geographic information systems (GIS).**

Keywords – Data visualization; Web component; GIS; Flow networks

I. INTRODUCTION

In recent times, data visualization has gathered increased popularity as an effective means of extracting and conveying information from large datasets. There are many forms of data visualization such as charts, graphs, timelines, infographics and map visualizations. As computers grew in processing power and over the years, the sophistication of these visualizations have also grown accordingly as developers were able to develop more complex visualization libraries and programs.

This paper explains the research and implementation of a web based tool for visualization of information/disease propagation in networks. The motivation for the project was the lack of a simple and intuitive tool for the use of researchers studying propagation of information and disease in network based models. The main purpose of the tool is to aid researchers in studying propagation of information, disease etc. using network based models. The spread of news, ideas and contagious diseases within a population is influenced by the level of interaction among individuals. Epidemiologists and scientists researching information diffusion within populations use network-based models to understand the rate and nature of propagation of diseases and information. Network visualizations of disease propagation particularly when overlaid on a map provide the ability to simulate scenarios and identify vulnerable regions, areas that serve as reservoirs of disease vectors etc.

The project itself was developed as a Google Polymer app [1], using Yeoman [2] to scaffold it. The core libraries used were D3 [3], ShapefileJS [4], jQuery [5], and Google Maps API [6].

II. LITERATURE REVIEW

When it comes to geographic data visualization, there is a number of concepts such as projections, Chloropeth Maps, use of colours, data formats to be considered.

Generally, a computer projects on a 2D plane. This gives rise to the problem of mapping the points on a 3D sphere, to the 2D plane. Many methods have devised by mathematicians and cartographers that can be categorized into 3 main families of projections [7]. First, a cylindrical projection is obtained by wrapping a cylinder around a globe which represents the Earth and projecting the image of the globe to the cylindrical surface. The map projection can then be obtained by unwrapping this cylindrical projection. Conic projections can be produced by projecting the globe onto a cone placed over the globe. Finally, Azimuthal projections can be produced by projecting the globe onto a plane [7].

In Chloropeth Maps, the data is visualized using colour intensities. Various intensities of a colour can be used to represent different ranges of the data. A geographic area can be assigned a particular intensity depending on the quantity that needs to be represented by that area. [8]

When using colours to encode quantitative data, the use of colours with different hues should be avoided as this leads to problems when ordering them from least to greatest. Different hues should be used when separating items on the map into different groups [8]. Color Brewer project [9] contains an open source set of palettes of different hues and intensities which can be used in visualizations.

When we consider the data formats the main source of maps required for the tool are Shape files, which is a popular geospatial vector data format for geographical information systems. Shape files store geometries of features as shapes comprising of sets of vector coordinates [10].

One of the most popular choices for data visualization is the D3 library. It is written in JavaScript and provides a vast array of visualization options. For visualizing maps, D3 has support for

12 types of map projections belonging to all 3 of the projection families mentioned above [11]. There are many visualization libraries based on D3.With advancements made in the web technologies, there are many app, services and libraries providing the capability to visualize data overlaid on maps.

III. System Models

A. System Requirement

There were five main functionalities required of the tool. Map upload, data upload, rendering the visualization, visualization options, visualization playback. The user should be given the option to upload the dataset to be visualized and a Shape file corresponding to the data being visualized. The user should be able to feed both, the network layout and the network flow data required for the visualization. Control over the styling and other options and parameters related to the visualization should be available for the user. After the necessary parameters are set and the data is provided, the user should be able to playback the dynamic visualization.

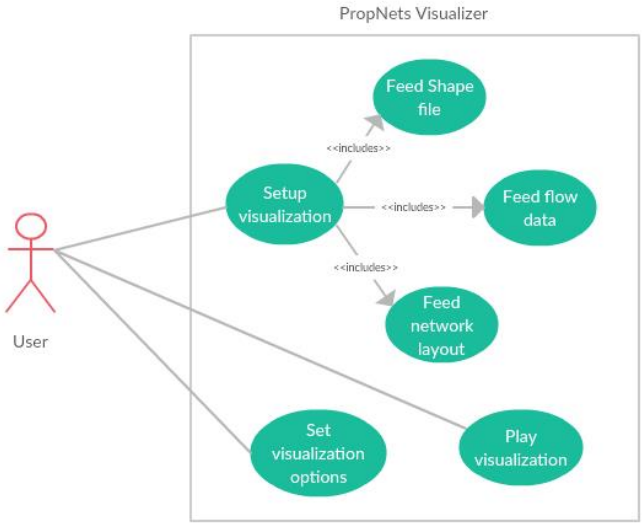

Fig. 1. Main usecase diagram

As shown in Fig. 1, there are 3 key use cases in the tool. Setting up the visualization includes all the necessary steps needed to get a minimal visualization out of the tool. These steps are the feeding of the 3 key data files into the tool: the Shape file, the network layout and the network flow data. Set visualization options use case deals with all the visual and animation parameters of the visualization. This use case is optional since the minimal visualization will consist of default visualization options. However, it gives great flexibility to the users in customizing the visualization to fit to their needs. Play visualization use case is a vital use case since this is the use case which will render the visualization, using all of the data and parameters set by the other use cases.

When considering the non-functional requirements, usability is the main focus for this tool. It should be easy to learn and use without minimal or no documentation of the tool. The user should also be able to fully grasp the tool and make use of it within a very short period of time, just by trying out the various features provided.

The tool is expected to visualize large volumes of flow data of networks up to ~3000 nodes. Therefore, it should be scalable and should have the capacity to handle the data without crashing the app or the browser. It should also be responsive regardless of the amount of data it handles. Standard operations such as visualization playback, panning, zooming and styling should function with minimal lag in the visualization.

Apart from the above, the tool should also be extensible, allowing more features to be added easily, in the future; especially the addition of visualization options. There were design constraints imposed by a stakeholder, LIRNEasia, as well. The expectations were that the tool be web based and use the D3 library for visualizing the data. They also expected Shape files to be the primary source of maps for the tool and Google Maps to be the secondary source.

B. System Design

The logical view of the main architecture of the tool is shown in Fig. 2. It follows a component based architecture, considering the extensibility requirement imposed by the client. The tool can be divided in to three distinct layers. The view layer contains all the presentation components of the tool. It is mainly driven by the visualization engine and will render the visualization data provided by the visualization engine. It also provides the UI needed for the File Handler layer.

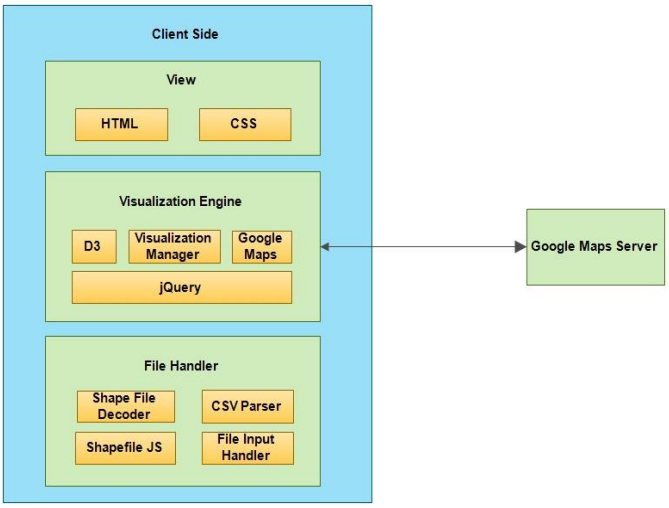

Fig. 2. Logical view

The map, network and flow data needed for the visualization engine is provided by the file handling layer. This layer makes use of the ShapefileJS library [4] to parse the Shape file and the

parsing process is managed by the Shape File Decoder component. CSV parser handles the parsing of the network and flow data files. It makes use of the parsing function provided in D3 [3, p. 3] to accomplish this. The File Input Handler controls the file input components in the view layer and reads and directs the different input files to their appropriate components for parsing.

The main layer of the tool is the visualization engine. It manages all the parsed data and renders visualizations based on that data. If the user has enabled the Google Maps overlay, it will make use of the Google Maps API as well and make requests to the Google Maps server accordingly. The Visualization Manager manages the visualizations by utilizing the 3rd party libraries D3, jQuery and Google Maps. All the visualization options are also handled through the Visualization Manager.

Fig. 3, shows an overview of the main sequence of actions that takes place when a visualization is being rendered. The sequence of actions in order are: feed Shape file, feed network and flow data, generate visualization. Here, the visualizations returned both when a Shape file is uploaded and when the CSV files are uploaded. This is because the basic map is rendered on the browser after the Shape file is fed into the tool. However, visualizations cannot take place when either or both of the 2 CSV files containing the visualization data are missing.

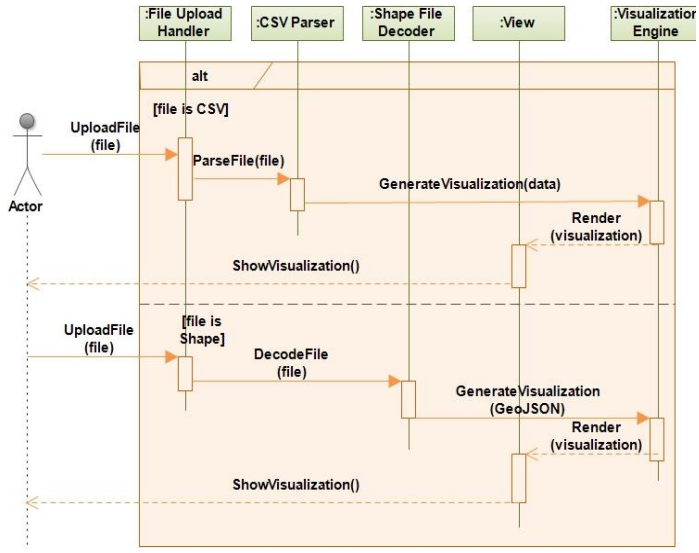

Fig. 3. Sequence diagram

IV. System Implementation

C. Implementation Procedure

Fig. 4, shows the flow of activities that take place when visualizing a set of data.

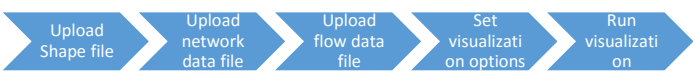

Fig. 4. Basic flow of activities

The development of the tool was broken down into two development iterations. The first iteration concentrated on adding all the basic functionality needed for the tool. This mainly included the file input mechanism, Shape file decoding process, CSV file parsing and rendering the Shape file map. During the second iteration, all the visualization options were added, including the Google Map overlay option. The tool was tested and built during the transition phase and it was deployed to Github pages. It is available at http://pubudu91.github.io.

The focus was on making the tool an independent web component adhering to the web component specification of W3C [12, 13, 14]. Publishing it as a web component would enable developers interested in the tool to incorporate it to their web apps easily. However, given the time constraints for the project, it was decided to develop the tool incorporating the functionalities required along with room to transform it into a web component standards compliant component. Given this goal, it was decided to develop the tool as a Google Polymer app in order to ensure a smooth transition of the tool from a standalone web app to a web component in future development.
.

Polymer is a library developed on top of the web components standards with the aim of making it easier for developers to build great reusable components for the modern web [15]. Given below is a logical view of Polymer.

Yeoman was used as a scaffolding tool and the Yeoman generator for Polymer [16] was used to scaffold the app. Yeoman scaffolds the app, complete with a build configuration and pulls in any package manager dependencies that is required for the project [2]. For polymer, the build system provided is Gulp [17] and the package manager provided is Bower [18].

For implementing the visualization engine, it was decided to use D3 [3, p. 3], a powerful and flexible JavaScript library dedicated for allowing developers to create stunning visualizations. Although this was a design constraint imposed by the client, it didn't cause any problems since D3 would have been selected even if there was not any constraint.

Since ease of use was a major concern, it was decided to use ShapfileJS [4] to decode the Shape file to GeoJSON, although it would have considerable simplified the development of the tool if the user was prompted to convert the Shape file to TopoJSON [19] prior to feeding it to the tool. TopoJSON is an extension to GeoJSON, which encodes topology. Since TopoJSON stores geometries as line segments called arcs, instead of as a collection of discrete points, the resulting TopoJSON file is considerably smaller (80+ % reduction in size is possible) [19]. However, currently there are no client side JavaScript libraries which are capable of converting Shape files

to TopoJSON. Therefore, it was decided to go ahead with converting Shape files to GeoJSON format in order to preserve simplicity of the app.

Parsing of the CSV files was made easy by the CSV file parsing method provided in D3 itself. The implemented parser only had to prepare and format the data returned by the method to the format required by the visualization engine. For the Google Maps overlay, the custom overlay option available in the Google Maps API was used. This makes use of the overlay layer in the Google Maps by appending an SVG tag to it and drawing the Shape file map inside it.

D. Materials

Random data was generated using Mockaroo [20], an online tool for generating random data. The data were generated for the two CSV files required: the network layout file and the flow data file. The CSV file specification for the two files: Network layout file for location1, location2, weight and Flow data file for source, source_latitude, source_longitude, destination, destination_latitude, destination_longitude, timestamp, source_infected, destination_infected. For the flow data, for frequent testing, a file with 100 records was used while for testing the performance, a file with 1000 records was used.

E. The Algorithm

Fig. 5, shows algorithm of: the visualize function which handles the animating of the visualization. The function calls made are to other functions defined in the source file and they are for accomplishing the tasks indicated by their names.

```
function visualize(flowdata, csvdata,
g,baseMapColour) {
   g.removeAllLines()
   baseTime = flowdata[0].timestamp
   maxTime = lastRecordOf(flowdata)
   maxWeight = getMaxWeight()
   gradientMapper = getGradientMapperFunction([0,
maxWeight], colourGradient)
   addMapLegend()
   for each element x in flowdata {
     relativeTime = x.timestamp - baseTime
     src = getLocationObjectByName(x.source)
     dest = getLocationObjectByName(x.destination)
     connectLocations(src, dest, relativeTime /
(maxTime - baseTime) * totalDuration)
     if (x.source_infected)
        colour = gradientMapper(weight(x.source,
x.destination))
     else
        colour = baseMapColour
     animateRegionColouring(x.source, colour,
relativeTime, maxTime, baseTime)
     if (x.destination_infected)
         colour = gradientMapper(weight(x.source,
x.destination))
      else
         colour = baseMapColour
      animateRegionColouring(x.destination, colour,
relativeTime, maxTime, baseTime)
   }
}
```

Fig. 5. Algorithm for the animation of the visualization

This function proceeds with a visualization if all the necessary data has been set. It first removes any remaining components from a previous visualization if they are present in the SVG [21] element and resets the values to the default values. Then it keeps track of the timestamps of the first and the last records of the flow data set. The timestamp of the first record is used as the base time relative to which all the time durations are calculated for the various visualization options. The relative time for a particular record *x* is computed as t(*x*) = (time of occurrence of x – base time).

The maximum weight between any given two nodes in the network layout is needed to calculate the quantitative range of sizes depicted by a single colour in the gradient. This is calculated by dividing the maximum weight by 9 (the number of different intensities in the colour gradient). This is vital in order to create the legend for the visualization and to decide the colour for a particular region in the map, based on the weight of the infection in that particular region.

The **getGradientMapperFunction()** function returns a gradient mapper based on the range provided and the colour gradient. Then, this function is used to calculate the appropriate colour for each region by passing the weight of each region to this gradient mapper function. The **animateRegionColouring()** function handles the animation of colour change of a region.

The call to the **connectLocations()** function handles the animation of the propagation between 2 regions. This is achieved using a delay calculated using the relative times based on the base time and the total duration the user wishes the animation to last. If the time of occurrence of the record *x* is given by t(*x*), the delay for a record *x* can be calculated as follows.

$$delay = \frac{t(x) - start\ time}{end\ time - start\ time} \times total\ duration$$

Where, Start time is the time of occurrence of the first record in the data set; End time is the time of occurrence of the last record in the data set and Total duration is the User specified duration of the animation

F. Main Interfaces

The interface given in Fig. 6 shows the data file input tab of the menu; Fig. 7, shows the visualization options tab of the menu. Fig. 8, shows an instance in time when running the visualization with the Google Maps overlay enabled.

V. System Testing and Analysis

During development, each of the features was tested using the test material mentioned in section 4-B on Google Chrome browser. During the transition phase, unit testing and automated

end to end testing was carried out on Firefox browser. Several techniques were considered for methodical testing of the tool.

Unit testing was done using QUnit. Tests were written only for the maximum weight returning function mentioned in section 4-C. Unit test coverage is low because the tool was designed in the component based approach and as such, it does not provide much public interfaces. For end – end testing, NightwatchJS was used. It is a JavaScript library which makes use of Selenium to automate the testing. The basic flow of activities which takes place from the start of page load to visualization was tested using this approach. Considering the results of these tests, it can be concluded that the behaviour of the tool is as expected.

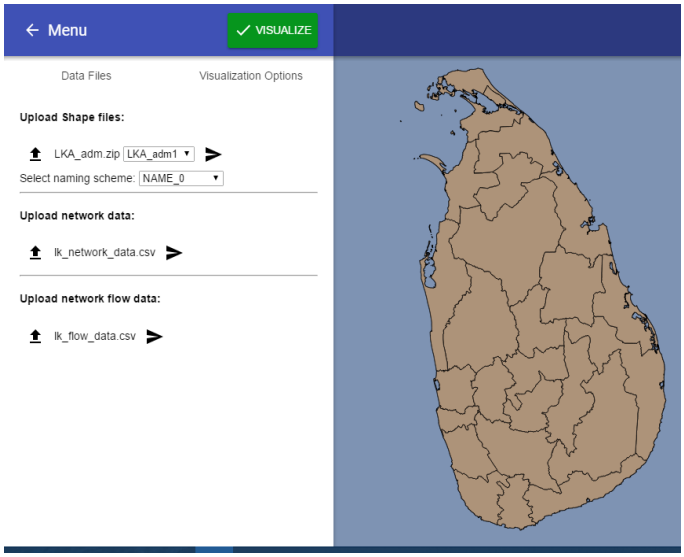

Fig. 6. The interface of the data files menu tab

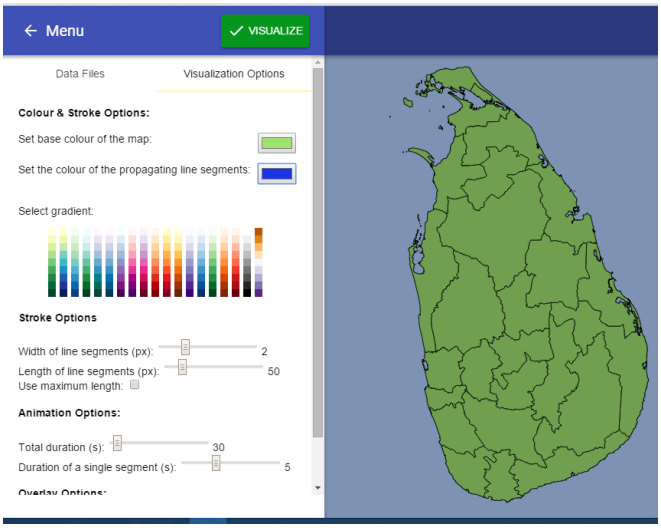

Fig. 7. The interface of the visualization options menu tab

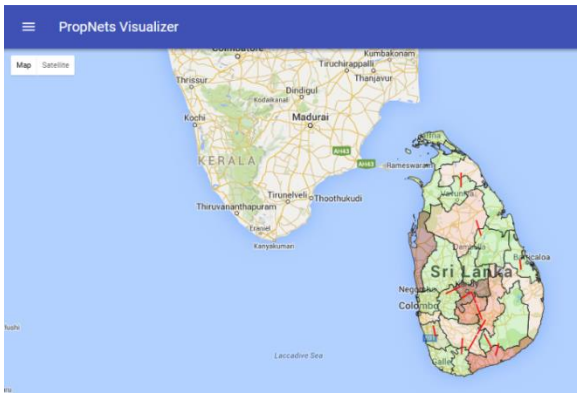

Fig.8. The interface during the visualization with Google Maps overlay enabled

For performance testing, the time taken for the visualization to playback after clicking the visualize button was recorded for flow data of varying sizes. Performance is a concern for the tool since it is expected to deal with large sets of data. After initiating the visualization, there tends to be a small lag while all the computations take place prior to the visualization. For smaller test data, with 100 rows of data, the lag was about ~3 seconds. But for the data file with 1000 rows of data, the lag was unacceptable (~30 seconds). However, the tool was capable of handling the visualizations without crashing the browser, even for the 1000 record input file.

The tool was tested for security using OWASP ZAP [22], a web application security tester. Security concerns are minimal for this app since all the processing take place on the client side. Furthermore, the tool does not require any sort of user details for it to function. Once the app is loaded, there is no interaction with the server. When a set of attacks were mounted on the app using OWASP ZAP, apart from 3 minor warnings, there were no security issues detected.

VI. Conclusion and Future Work

This paper describes the design and implementation details of PropNets Visualizer: a web based tool for visualizing propagation of information/disease in network based models. The tool allows researchers studying propagation of news, information, disease etc. an easy to use, intuitive, simple tool to visualize the network flow data in a dynamic manner. This enables the researchers to identify any patterns, recognize vulnerable regions which act as reservoirs of disease vectors.

The tool allows the user to upload a map (in the form of a Shape file), network model data and flow data and visualize the dynamic flow of the data, overlaid on the map. It provides a variety of visualization options and parameters to customize the visualization as the user sees fit. An optional Google Maps layer can be overlaid should the user wish to do so. The tool was developed using Google Polymer, with the aim of developing it

into a reusable web component. For data visualization, the D3 library was used.

The tool can be extended in many ways, such as refining the functionality of the Google Maps overlay option; Option to overlay several Shape files (i.e: add a road map Shape file on top of the administrative boundaries map); Reduce the latency experienced while waiting for the visualization playback to begin and provide better support for large data sets; A mechanism to play, pause and seek the animation; Further refine the UI and add more visualization options; Re-evaluating the code and refactoring it to get rid of any bad coding practices and transform it to adhere to the web components specification; Break down the main JS library of the app into separate logical source code files and make use of RequireJS to handle the JS file and module loading; Try and incorporate Mapshaper [23] to the tool. Mapshaper provides a lot more options for handling Shape files than the library we are currently using for reading Shape files. And the author of the library claims that it is capable of handling Shape files as large as 1GB. The tool has the potential to keep evolving with time into a more powerful visualization tool for researchers studying propagation of information and diseases.

REFERENCES

[1] "Welcome - Polymer 1.0." [Online]. Available: https://www.polymer-project.org/1.0/. [Accessed: 20-Oct-2015].

[2] "The web's scaffolding tool for modern webapps | Yeoman." [Online]. Available: http://yeoman.io/. [Accessed: 18-Sep-2015].

[3] "D3.js - Data-Driven Documents." [Online]. Available: http://d3js.org/. [Accessed: 20-Oct-2015].

[4] "calvinmetcalf/shapefile-js," *GitHub*. [Online]. Available: https://github.com/calvinmetcalf/shapefile-js. [Accessed: 20-Oct-2015].

[5] "jQuery." [Online]. Available: https://jquery.com/. [Accessed: 20-Oct-2015].

[6] "Google Maps APIs for Web," *Google Developers*. [Online]. Available: https://developers.google.com/maps/web/. [Accessed: 20-Oct-2015].

[7] "The Three Main Families of Map Projections - MATLAB & Simulink - MathWorks India." [Online]. Available: http://in.mathworks.com/help/map/the-three-main-families-of-map-projections.html. [Accessed: 17-Sep-2015].

[8] S. Few, "Introduction to geographical data visualization," *Vis. Bus. Intell. Newsl.*, pp. 1–11, 2009.

[9] "ColorBrewer: Color Advice for Maps." [Online]. Available: http://colorbrewer2.org/#. [Accessed: 27-Oct-2015].

[10] "ESRI Shapefile Technical Description." Environmental Systems Research Institute (ESRI), Jul-1998.

[11] "Geo Projections · mbostock/d3 Wiki." [Online]. Available: https://github.com/mbostock/d3/wiki/Geo-Projections. [Accessed: 17-Sep-2015].

[12] "Shadow DOM." [Online]. Available: http://w3c.github.io/webcomponents/spec/shadow/. [Accessed: 18-Sep-2015].

[13] "Custom Elements." [Online]. Available: http://w3c.github.io/webcomponents/spec/custom/. [Accessed: 18-Sep-2015].

[14] "HTML Imports." [Online]. Available: http://w3c.github.io/webcomponents/spec/imports/. [Accessed: 18-Sep-2015].

[15] "What is Polymer? - Polymer 1.0." [Online]. Available: https://www.polymer-project.org/1.0/docs/start/what-is-polymer.html. [Accessed: 18-Sep-2015].

[16] "yeoman/generator-polymer," *GitHub*. [Online]. Available: https://github.com/yeoman/generator-polymer. [Accessed: 26-Oct-2015].

[17] "gulp.js - the streaming build system." [Online]. Available: http://gulpjs.com/. [Accessed: 26-Oct-2015].

[18] "Bower." [Online]. Available: http://bower.io/. [Accessed: 26-Oct-2015].

[19] "mbostock/topojson," *GitHub*. [Online]. Available: https://github.com/mbostock/topojson. [Accessed: 18-Sep-2015].

[20] "Mockaroo - Random Data Generator | CSV / JSON / SQL / Excel." [Online]. Available: https://www.mockaroo.com/. [Accessed: 26-Oct-2015].

[21] "W3C SVG Working Group." [Online]. Available: http://www.w3.org/Graphics/SVG/. [Accessed: 27-Oct-2015].

[22] "OWASP Zed Attack Proxy Project - OWASP." [Online]. Available: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project. [Accessed: 26-Oct-2015].

[23] "mbloch/mapshaper." [Online]. Available: https://github.com/mbloch/mapshaper. [Accessed: 18-Sep-2015].