**CHAPTER 4**

**DEVELOPMENT**

This chapter provides the development process of this project which includes the single hexagon tile setup, JSON data file setup, hexagon tile plotting setup and the event listener setup. This chapter explains each process through programming codes and the challenges faced during the development of the system are also explained.

**4.1 Introduction**

The main functionality for this system is to convert the election data into a hexagon tile grid map visualization. The election data are manually stored into a JSON file to make the retrieval of the data as smooth as possible.

The system is developed by using a combination of HTML, Cascading Stylesheet (CSS), Javascript (JS) and Data Driven Document (D3.js) together with D3.js hexagon plugin, hexbin. The HTML and CSS were used to setup the foundation of the User Interface (UI) of the system while JS and D3.js were used to implement the algorithm flow of the system such as creating and populating the hexagon based on the coordinate data stored in JSON file. The hexbin plugin of D3.js were used to help creating the hexagon tile for a better coordination. All the output seen in the HTML document are in Simple Vector Graphic (SVG) format produced by D3.js. The system also comes with an administrator part that were developed using Hypertext Preprocessor (PHP) scripting language and it is used mainly for data entry.

**4.2 System Modelling**

This project consists of two types of files, namely the HTML file and the JSON data file. The HTML file contains the layout of the system UI and the logic behind the hexagon tile grid map. The logic uses D3.js, which is a Javascript library used in the same HTML file through the <script> tag. JSON files contains the data that are needed to process and populate into the system UI. All files for administration system purposes are stored in “admin” folder. The file structure is shown in Figure 4.2:

|  |
| --- |
| **Figure 4.1** File structure of the system |

Based on Figure 4.1, the files that are in the “json” folder contains the JSON data file that stores the data needed to run the system. Files in “lib” folder contain all the libraries and dependencies that are used in the system. The file d3.min.js is the core library for D3.js, d3-hexbin.min.js contains the extended plugin of d3.js used to draw the hexagon and jquery-3.3.1.min.js contains the core library for jQuery that will be used in the system. All HTML files stored in “pages” folder is the main folder for the system that contains the HTML document for the UI and the Javascript for logical operation of the system. The structure of the “index.html” file which is the main file in this system is shown in Figure 4.2:

|  |
| --- |
| **Figure 4.2** Main file coding structure, parlimen-v2.html |

Based on Figure 4.2, there are two main sections in the file. These sections are the HTML document section and Javascript section. The HTML document section contains the division that shows the data visualization. The Javascript section contains the import section and the script section. The import section is where all dependencies and libraries such as Jquery, D3.js and hexbin plugin are imported into the HTML document and the script section contains the logical part of the whole system.

Furthermore, the “admin” folder stores all files related to the administration system and there are 3 files for administration systems based on three processes for data entry. The “admin-demography.php” file were used for demographic data entry, “admin-parliament.php” file were used for parliament data entry and “admin-election.php” were used for actual election data entry.

**4.2.1 Single Hexagon Tile Setup**

This project uses the hexbin plugin for D3.js to draw the hexagon tile. Its ability to easily create a single hexagon tile is one of the reasons why this plugin was chosen in the first place. Hexbin plugin will produce a SVG path based on the developer’s configuration. The SVG path will eventually create a hexagon tile. The plugin generates the hexagon from the perspective of a circle. Therefore, the main parameter that the developer had to include is the radius of the hexagon tile and it is considered as the size of the hexagon. After that, the developer can plot the hexagon anywhere inside the SVG plain. Figure 4.3 shows the code for single hexagon tile setup:

|  |
| --- |
| **Figure 4.3** Coding for a single hexagon |

Figure 4.3 shows the code to draw a single hexagon using the D3.js library and hexbin plugin. In variable declaration section (from line 21 until 27), *coordinate* variable stores a list of coordinates (in this case, one coordinate) that are used to plot the hexagon. The variable *svg* stores the instance of the <svg> in the <div> element with id of “chart” and variable *hexbin* stores the instance of the hexbin plugin with the hexagon radius being declared as “10”. This means that the size of the hexagon is drawn based on the circle radius of 10 pixels. However, the size of hexagon is adjustable according to the development need. The developer can increase the radius to enlarge the size of hexagon and vice versa.

In the process section (from line 29 until 37), the real operation of the hexagon drawing takes place. Firstly, line 30 shows a D3.js selection function *.selectAll()*, which selects all of the occurrence of <g> tag in the HTML document during execution. On line 31 and 32, for each data stored in the *coordinate* variable, the system will create a new instance of <path> tag inside the <g> tag. On line 33 until 35, the attribute of the <path> tag are initialized by using the *.attr()* function.

Most of the function in D3.js requires 2 parameters, name of the attribute and its value. The <path> tag requires the developer to include one main attribute, the *d* attribute. The *d* is used to draw a path by using the <path> tag in SVG plane (inside <svg> tag). The value for *d* attribute is a set of strings that determine the command for the drawing of the <path>. The value included for *d* is the coordinate command that includes the coordinate to which the hexagon will be plotted (as stored in the *coordinate* variable) and the shape drawing command the hexbin plugin generates by using the *.hexagon()* function. On line 36 and 37, the styling of the hexagon are stated using the *.style()* function. In this case, the style attribute that are being initialized is *stroke,* the border line color of the hexagon and *fill,* the color inside the hexagon. Developer can choose any color for the *stroke* and the *fill* attribute. The function will produce the output as shown in Figure 4.4:

|  |
| --- |
| **Figure 4.4** Single hexagon output |

**4.2.2 JSON Data File Setup**

This project stores all related data in JSON file format. There are 4 main JSON data files namely “setting.json”, “parliament.json”, “election.json” and “demography.json”. Data stored in “setting.json” are used to store the value for the main setting of the system. The “parliament.json” file stores data for parliament setting data by states. The “election.json” file stores the data for election result for each parliaments. Lastly, the “demography.json” file stores the demographic data for each states and its parliaments.

There are two format of JSON data file which is array format and Javascript object format. The array format are data are stored as array in JSON file and they are represented as data inside a pair of square brackets ([]). The Javascript object format are data stored as object in JSON file and their data can be accessed using the dot (.) operator which is the same as accessing object attribute for Javascript. The data stored in Javascript object format are located inside a pair of curly braces ({}).

Data in “setting.json” file includes settings for the SVG elements, settings for hexagon (which is the hecagon radius), the settings for tooltip functionality and lists of colours used in the system including political party colour and colours for each states. Most of these settings are used to create the system user interface. Figure 4.5 show the content of “setting.json” data file:

|  |
| --- |
| **Figure 4.5** Data in “setting.json” file |

Data in “parliament.json” file include settings for each hexagon that represents each parliament by states. The data includes the state name list of parliaments in the state. For each of the parliaments, it consists of the parliament code, parliament name and the coordinate of the hexagon that represents the parliament. The parliament code represents the code that will be used as a key to find the election data related to the parliament in “election.json” file. The hexagon coordinate ise divided into *x* and *y* coordinates and this coordinates will be used to populate hexagon that are related to the particular parliament. Figure 4.6 shows the content of “parliament.json” data file:

|  |
| --- |
| **Figure 4.6** Data in “parliament.json” |

Data in “election.json” consist of the actual election data that are related to each of the parliaments. The data includes a list of states that stores the state name and a list of election result for each of the parliaments in that states. Each result consists of the parliament code, total voters, total votes for each political party and the information about the winning party and candidate. Figure 4.7 shows the content of “election.json” file:

|  |
| --- |
| **Figure 4.7** Data in “election.json” |

Data in “demographic.json” consist of the demographic information for both state and its parliament. The demographic data for the states consist of the gender information, which tells how much male and female are in that state. The demographic data for the parliaments consist of the ethnic or race information which tell how much a certain ethnic are in that parliament. Figure 4.8 shows the content of “demographic.json” data file;

|  |
| --- |
| **Figure 4.8** Data in “demographic.json” |

D3.js comes with the functionality to read JSON data from external file. D3.js reads JSON data from external files by *d3.json()* function. The code for reading the json files are shown in Figure 4.9:

|  |
| --- |
| **Figure 4.9** Code snippet for *d3.json()* function |

The function for *d3.json()* requires two parameters which is a string of path for JSON file to be read and callback function after the function executes. The callback function can include one parameter which represents the data captured in the JSON file. By using this function, all the data stored in the external JSON files are stored in the parameter of the callback function. For example, in Figure 4.8, on line 78, all of the data retrieved in the “parliament.json” file will be stored into *parliamentList* variable. Then, the data can be accessed by using the same Javascript method of accessing object or array data depending on the data stored in the JSON file. For example, on line 86 and 87, the data stored in “parliament.json” are accessed using the *parliamentList* variable. The accessing of the data need to be in Javascript array format because the data stored for a list of parliaments in “parliament.json” is in array form. Then the coordinate of the parliament can be accessed using the dot (.) operator (same as accessing the attribute of Javascript object) because the coordinate data are stored as Javascript object.

**4.2.3 Hexagon Tile Grid Map Plotting**

This project implements the pointy-top hexagon because the hexbin plugin are not developed with a flat-top hexagon in mind. However, by using hexbin plugin, flat-top hexagon can still be created nonetheless just by transforming (rotate) a single hexagon, but there are some problems related with the rotation of the hexagon. The problems are explained in detail discussion sub-topic of Chapter 5. Moreover, this project implements the offset coordinate system which uses the classical 2-dimensional coordinate system (x-axis and y-axis). The reason why this project uses the offset coordinate system is it is one of the popular coordinate system for 2-dimensional visualization and hexbin plugin implements it.

Hexbin plugin implements the offset coordinate system and requires the developer to only input the list of coordinates for each hexagon. From the coordinate, hexbin will create a hexagon path and place the hexagon on the specified coordinate. Furthermore, hexbin will also automatically place a partially overlapping hexagon side by side or completely overlap them. So, there will be no partially overlapped hexagon in the visualization

The process of populating the hexagon in the visualization is the system will retrieve the parliament coordinate data from JSON file first. Then, it will process the retrieved data and store the data into a single variable so that it is able to pass on to the hexbin plugin. The hexbin plugin will take a single value that consist lists of coordinates of the parliaments and produce a path to place in the SVG DOM element.

The parliament coordinate data are retrieved from the “parliament.json” data file and this will result in multiple data that are related to the parliament settings. However, some of the data are not needed for this process because at this stage, the system will only need the coordinate of the parliament to populate the hexagon. The system will filter the retrieved data so that only parliament coordinate is retrieved and save the data into a separate variable to be used in the next stage. Figure 4.10 shows the coding on how to retrieved the data in “parliament.json” and filter the data so that only the coordinate data are retrieved and save them in a separate variable:

|  |
| --- |
| **Figure 4.10** Code to store parliament coordinate from “parliament.json” file |

Based on Figure 4.10, on line 78, D3.js uses *d3.json()* function in order to retrieve data in “parliament.json” data file and store the retrieved data in *parliamentList* variable. On line 80 until 91, the data retrieved from “parliament.json” will be filtered and only parliament coordinate data will be taken. The coordinate data are stored in the *parliamentCoordinateList* variable. The coordinate data are retrieved from “parliament.json” by accessing the value of *coordinate.x* and *coordinate.y*.

After the coordinate data has been retrieved, the system will populate the hexagon for each of the specified coordinate. Figure 4.11 shows the code for this process:

|  |
| --- |
| (a)    (b)    (c)  **Figure 4.11** Code to populate hexagon tile based on parliament coordinate |

Based on Figure 4.11, on line 66 until 70, *svg* variable are declared to initialize the width, height & location of the SVG element in the HTML. The width and height of the SVG element are initialized based on the *setting* value retrieved from “setting.json” data file. The *hexbin* variable initializes the hexbin plugin by setup the hexagon radius value retrieved from “setting.json” data file also.

On line 96 until 107, a list of *<g>* tags are initialized in the SVG element to represent states in Malaysia. The *<g>* tags represents a group of *<path>* tags that will be created by hexbin plugin (hexagon paths). Furthermore, on line 97, the list of parliaments that was initialized previously are made as the input for the creation of the *<g>* tags which in turn will initialize all states of Malaysia. This is because the *parliamentList* variable stores a 2-dimensional array that incorporates two things, which is the list of states and a list of parliaments within those states. For this steps, D3.js will only read the list of states and not the list of parliaments within the states to populate the *<g>* tags and it will iterate all of the data to create the *<g>* tags for every states. On line 99 until 107, the attribute and style of each *<g>* tags are initialized for every itteration. The attribute that are initialized is the *id* which is the name of the state itself and the style that are initialized are the *fill* and *stroke* of the hexagons which will set the colour of the hexagon to the colour as in “setting.json” file.

On line 148 until 159, each of the parliaments in the states are initialized based on the data stored in *parliamentCoordinateList*. On line 149, the value of the hexagon path are input together with the parliament coordinate value. The .*hexbin()* function returns a list of hexagon paths that will be used to populate the hexagon. On line 152, each paths of the hexagon of the parliament are initialized and each of them are given the “.parliament” class attribute to make the HTML element selection process easier. On line 154 until 156, the id of each parliaments is initialized which is the parliament code itself. On line 158, the attribute *d* of the *<path>* tag is initialized and it represents the path of the hexagon as well as the coordinate the hexagon should be placed.

From the above code, it will produce the output (without colour) as shown in Figure 4.12:

|  |
| --- |
| **Figure 4.12** Output for the population of hexagon tiles |

**4.2.6 Hexagon Tile Colour Setup**

For each of the hexagon tiles in the map, they are coloured according to the event that occurs and the position that they are representing. There is an event that will cause their colour to change which is when the user clicked on the state. This event will cause the clicked state to be focused by reducing the opacity of the surrounding states and change the colour of each hexagon in that state. The colour change is based on the party that have won. The codes for the colour setup are shown in Figure 4.13:

|  |
| --- |
| (a)    (b)  **Figure 4.13** Code for hexagon tile color setup |

Based on Figure 4.13, line 155 until 160 shows the code for setting up the colour of each states. D3.js provides a simple function for the developer to change the style of the HTML elements dynamically. This function can be accessed by calling *.style()* function. This function requires two parameters namely the properties of the element to be changed and its value. The value of the properties can be processed further by calling anonymous function in the second parameter. For example, on line 155, the *.style()* function are called with the parameter of “fill” and a callback function. The “fill” indicates that the developer wanted to change the “fill” properties of the element (which will fill up the element with the specified colour) and the callback function will return appropriate value for the properties which in this case, it returns a colour code stored in “setting.json” file. In addition, on line 158, the properties “stroke” will determine the element’s stroke colour. These property are used to change the border colour of each of the hexagon.

Furthermore, on line 198 until 213 shows the code for setting up the colour of each individual hexagon tile and this statement are called inside the *.on()* function with the “click” event listener. So, the colour of the hexagon will change when the viewer click the state. On line 198, the stroke colour property of the hexagon are changed into “white” value using the *.style()* function with “stoke” property set as its first parameter. On line 199, the “fill” property of the hexagon are changed using the *.style()* function “fill” property as its first parameter and an anonymous function as its second parameter. The anonymous functions shown in line 199 until 212 are used to set the appropriate colour of the hexagon tile according to which winning parties during the election for that parliament. The output of this process is shown in Figure 4.14:

|  |
| --- |
| **Figure 4.14** Output for hexagon tile colour setup |

**4.2.7 State Labeling Setup**

Each of the states drawn is labelled with their own name. The labelling process occurs after all the hexagon tiles has been plotted into the map because to let the state label to be drawn on top of all the hexagons.

|  |
| --- |
| (a)    (b)  **Figure 4.15** Code for state labeling setup |

Based on Figure 4.15, on line, 510 until 523 is the code for initializing the label text, their attributes and their styling and the remaining lines of codes shown the process for initializing the coordinate a label should be appearing.

Line 509 shows the code for appending a *<text>* tag inside the map *<svg>* tag. This code is to ensure that the labels will not overflow outside of the SVG elements. Line 510 shows the process of initializing the text that will appear inside the state label. This text will be based on the state labels as stored in “parliament.json”. Line 511 and 512 shows the process of initializing the *class* and *id* attribute for each tag and these attributes are used for selection for various purposes such as styling all the labels at once using the CSS inside *<style>* tag. Line 513 until 517 shows the process of initializing the style of each labels such as the *font-weight*, *text-shadow* (used to draw the outline of the text), *fill* (used to colour the text) and *cursor* and *pointer-event* (used to change the behaviour of the mouse pointer so that when it hovers on the label, the mouse pointer icon will not change and clicking on the label will not highlight the label). The remaining lines of code shows the process of initializing the coordinate of the label by its *x* value and *y* value. The result of the labelling code is shown in Figure 4.16:

|  |
| --- |
| **Figure 4.16** Output for state labeling setup |

**4.2.4 Mouse on Hover Tooltip Setup**

This system implements D3.js event listener to make the map interactive. One of the event listener that are used is the “mouse on hover” event. This event will be triggered when the viewer hovers their mouse pointer over the specified elements. The triggered event can be initialized with any action depending on the developer needs. However, for this project, a tooltip will pop up beside the mouse pointer icon when the viewer hover over their mouse pointer on the *<path>* tags which is the hexagon of the parliament. When the viewers hover over the parliaments, the tooltip will show the parliament code and parliament name. The code for this event listener are shown in Figure 4.17:

|  |
| --- |
| (a)    (b)  **Figure 4.17** Code for mouse hover each parliament event listener |

Based on Figure 4.17, the codes represent the settings needed for D3.js to initialize the event listener for mouse hover for each of the parliaments. On line 72 shows the tooltip element that are initialized first before it is being used.

The event listener are triggered by D3.js from the function *.on()*. This function requires two parameters namely an event name that indicates the event that the D3.js will be listening and the callback function that will do the action when the event is triggered. The event name is a string that describes the name of the event. For example, “mouseover” event lets D3.js listen for event when a mouse pointer is hovering over the specified element, “mousemove” lets D3.js listen for event when the mouse pointer is moving within the specified element and “mouseout” lets D3.js listener for event when the mouse pointer is no longer hovering over the specified element.

When the mouse pointer hovers over the parliament, the tooltip will appear beside the mouse pointer with the transitioning effect from zero percent opacity to 90 percent opacity. When the mouse pointer is moving inside the parliaments, the tooltip will follow the mouse pointer. When the mouse pointer is no longer hovers over the parliaments, the tooltip will fade by reducing its opacity from 90 percent until zero percent.

Based on Figure 4.13, on line 442 until 453 is the event that will be triggered when the mouse pointer hovers over the hexagon tile which in this case is it will pop up a tooltip beside the mouse pointer with the parliament code and its name while transitioning itself from zero percent until 90 percent opacity. On line 454 until 457 is the event that will be triggered when the mouse pointer moves while hovering over the hexagon which is the tooltip will follow the mouse pointer. On line 458 until 461 is the event that will be triggered when the mouse pointer no longer hovers over the hexagon tile which will result in the tooltip slowly transitioning its opacity from 90 percent to zero percent. The result of the above code is shown in Figure 4.18:

|  |
| --- |
| **Figure 4.18** Output of the tooltip mouse hover over event listener |

**4.2.5 Mouse on Click Popup Setup**

The system also implements the “on click”event listener provided by D3.js to listen for user click on each states and parliament hexagons. Developer can provide their own functions for the “on click” event listener. However, for this project there are two different events that will be triggered when viewer clicks the map depending on the current level of the information for the map. There are 3 different levels of information for the map and each level are explained in detail in the User Interface sub-topic. Those event listeners will only be triggered when the map is in the first and second level of information only.

The event listener for the first level of information are initialized for each of the states in the map and when the viewer clicked on one of the states, an information will pop up inside the popup board that explains about the demography information of that states. The code for this “on click” event listener is shown in Figure 4.19:

|  |
| --- |
| (a)    (b)    (c)  **Figure 4.19** Code for first level of information on click event listener |

Based on Figure 4.19, on line 270 shows the code for initialization of the “on click” event listener. Using D3.js, the “on click” event listener can be accessed through *.on()* function by providing the “click” as its first parameter to represent the “on click” event listener that the application has to listen and a callback function as its second parameter to represent the action that should be taken when the “on click” event are being triggered. Line 271 represents the conditions that checks the information level through the global *level* variable.

On line 276, it shows the popup board content initialization. The content of the popup board are initialized using the *.load()* function provided by the jQuery library. This function requires two parameters which is file name and the callback function and this function will import all the content in the specified file (provided in the first parameter) into the popup board and this will in turn overwrites any content inside the popup board. The callback function of *.load()* for this initializes the style for the popup and the popup board content by retrieving data in “election.json”.

On line 307 and 308 is the process of zooming the map so that it focuses on the clicked state only. Line 307 shows the initialization of the *zoomContainer* and these container are invisible in the map because they are used as a guide for the zoom mechanic define on line 308 as *zoomIn()* to determine where the camera should focus on. The *zoomIn( )*is a local function that defines the zoom mechanic of the system and it will zoom on a certain location depending on what element the *zoomContainer* is focused.

The rest of the codes in Figure 4.19 (b) shows the initialization process of user interface aspect of the map. Those aspects are decreasing the opacity for all states except the clicked state to give a focused view on that current state and divide all the hexagon in the focused states to represents each parliament by filling the background colour of each parliaments according to the colour of the winning party and stroking a white border around each of the hexagons.

On line 495 until 504, the code shows the initialization process of bar chart inside the popup board that will represent the gender demographic value of that state. Firstly, the data are initialized in line 495 and the data are taken from the “demography.json” file. Then, the bar chart is initialized on line 500 by using a local function called *loadVerticalBarChart()*. The output of the code in Figure 4.19 are shown in Figure 4.20:

|  |
| --- |
| **Figure 4.20** Output for on click event listener in the first level of information |

The event listener for the second level of information are initialized for each of the hexagons (which represents parliaments) in the state that the viewer had clicked and when the viewer clicked on one of the hexagons, the information in the popup board changes to the election information instead of the demography information.

|  |
| --- |
| (a)    (b)    (c)  **Figure 4.21** Code for second level of information on click event listener |

Based on Figure 4.21, line 385, shows the code for initializing the on click event listener for the second level of information. Line 388 represent the condition logic that will check the levels of information and whether the hexagon that the viewer has clicked is currently focussed or not. All remaining codes except on line 442 until 469, are functioning in the same way as the code for overwriting the popup element to represent the demography information for each state.

On line 442 until 456, the data for the pie chart of election result and demographic ethnic value are initialized the same way as in first level of information. On line 458 until 685 is the initialization process of the pie chart and bar chart by using the previously initialized election result and demographic data. The process uses the local function *loadPieChart()* and *loadHorizontalBarChart()* that takes seven parameters namely the value for *id* attribute of the pie chart and bar chart, the *<div>* tag that the pie chart and bar chart should overwrite, the data, height and width of the pie chart and bar chart, outer radius value of the pie chart and the inner radius value of the pie chart. The output for the code in Figure 4.21 are shown in Figure 4.22:

|  |
| --- |
| **Figure 4.22** Output for second level of information on click event listener |

**4.3 Data Entry by Administrator**

This system comes with an administration system, which can be used for data entry. The administration system uses mainly the Hypertext Preprocessor (PHP) scripting language as its core language. The data that are entered in the administration is the parliament data, demographic data and the actual election data. When the administration submits the data entry, the system will store all of the data into each of the JSON files (“parliament.json”, “demography.json” and “election.json”).

**4.3.1 Code for Administration System**

Most of the implementation of data entry system for administration has the same pattern of codes for all data entries systems. Those implementations are retrieving data from JSON data file, decode those data, populate those data into the form for submission, retrieve data in the form and save those data into the same JSON data file by overwriting the old data.

|  |
| --- |
| (a)    (b)    (c)    (d)    (e)  **Figure 4.29** Code for administration demographical data entry |

Figure 4.29 (a) shows the process of data retrieval from JSON data file and the data retrieval process is a bit different than D3.js in PHP. PHP uses *file\_get\_contents()* function to retrieve data from external files through Hypertext Transfer Protocol (HTTP) GET request. The function requires the developer to include the path of the files as its parameters and it will return the content of the file in string format or a *null*. The retrieved data are then decoded from JSON into PHP associative array.

Figure 4.29 (b) and (c) show the process of initializing the HTML documents together with its CSS and Javascript file for Bootstrap UI template. All codes inside the <body> tag shown in Figure 4.29 (d) is the initialization of the HTML document and the process of populating the data retrieved from external JSON data file were done here. Figure 4.29 (e) shows the process of retrieving user input data in the form and storing those data into the same JSON data file. Figure 4.30 until Figure 4.32 show the output of the administration data entry system.

|  |
| --- |
| **Figure 4.30** Output of data entry system for parliament |

|  |
| --- |
| **Figure 4.31** Output of data entry system for demography |

|  |
| --- |
| **Figure 4.32** Output of data entry system for election |

**4.7 Summary**

In summary, this chapter explains about how the project has been developed and the technologies that have been used. The development uses and integrates the features of the technologies, which are namely HTML, Javascript, JSON file, CSS and especially D3.js library. Data preparation in this project used JSON format to organize all the election data. There are four JSON files in this project, which are ”setting.json”, “election.json”, “demography.json”, and “parliament.json”. For the main part of this system, it focuses on using the D3.js as the main library for all the data visualization components by embedding these components in Javascript. These components include hexbin plugin, pie chart function and SVG elements. Arrangement of layout is done using CSS in HTML. This chapter ended with the introduction to the development of the administration data entry system. To ease the data entry by the administrator, three forms are created for parliament data, demography data, and election data.