DataForce

Salesforce #2
Agile Network Tree View

By: Ben Tarman, Jake Erickson, Nick Corl

6-21-2017

Introduction

Client Description

Salesforce is an American cloud computing company. Their primary service is providing a customizable interface for resource management and task management for businesses. Salesforce also provides a variety of other services including: Force.com, Work.com, Data.com, Desk.com, and the AppExchange. These services find a variety of uses for companies and Salesforce developers across the world.

Vision

Salesforce has tasked our team with creating an agile network view tool to be used as an extension to the Agile Accelerator application. They hope to use the software to allow for easier comprehension of their agile process rather than having to understand the process strictly based on text. The tool will ultimately focus on showing a visual view of the current work items that the developers can filter through. The overall goal is to provide developers using the app a more appealing visual design. In addition, this will allow enhanced searching of parent/child relationships between work items.

At a high level, we are creating a graph to display relationships between work, team members, sprints, epics, and other agile representations. In agile, sprints refer to assigned coding days for which objectives are to be met by a specified deadline. Epics on the other hand refers to a conglomerate of work items in a particular category. Work items should be shown in a traditional Salesforce style table view. Ideally, the user will be able to filter through the work items by applying further details. This could be a name for a Sprint, Epic, or a Team. Upon completion, the web page should override Salesforce's current work item list tabs.

This project will be written using Lightning Components included in Salesforce API. This entails using tools such as html, CSS, javascript, Apex, SLDS along with several CSS stylesheets and javascript libraries. Ultimately, the project will be used by developers who use the Agile Accelerator App in their agile processes.

Upon completion the objective is to have a powerful graphical display of the agile process in order to make comprehension and details more accessible for Salesforce.

Requirements

Functional Requirements

- Use d3.js to create graphical view of node tree graphs.
- Svg, used for vector-based graphics on web pages, will be used to render d3 graphs
- Work items are represented as nodes on the tree graph. Each node must have details associated with it such as a sprint, status, priority, type, epic name, etc.
- Implement a details box whenever a work item is clicked or hovered over to show additional details associated with the work item. Doing so will make less information being visible at first glance making the graph easier to read.
- Strong filter system based on keywords and work variables. Preferably make the user able to search through the filter

Non-Functional Requirements

- Webpage must be implemented using Salesforce Lightning Components
- Webpage should be integrated with the Salesforce Agile Accelerator Application
- Final application must be ran on a Visualforce page. The work item view can then be overwritten to display the page
- Salesforce Lightning Design System (SLDS) must be used for styling in order to maintain consistency with Salesforce's overall visual look
- SOQL, a specifically optimized version of SQL for working with Salesforce.com, will query data in an Apex server
- Node tree graphs should be generated dynamically on the client side only when work items are filtered
- All code must be done with the web programming languages used in Lightning Components: javascript for client side logic, html and CSS for web pages, and Apex for server side logic

Risks

A. Technology Risks

- Our graphs may behave different when using our own test information rather than actual Salesforce data
- The webpage cannot be directly implemented onto the Salesforce site until our project is complete

- Generating our graphs in a neat fashion may not be possible if there is too much data.
- Different browsers may limit the functionality that d3.js nodes can perform
- Different browsers experience different performance

B. Skill Risks

- None of our team members had any experience using lightning, SLDS, d3.js, or SOQL.
- A member of our team had no previous experience with javascript
- Two of our members were familiar with html and CSS. One had some design experience prior to the start of the project.

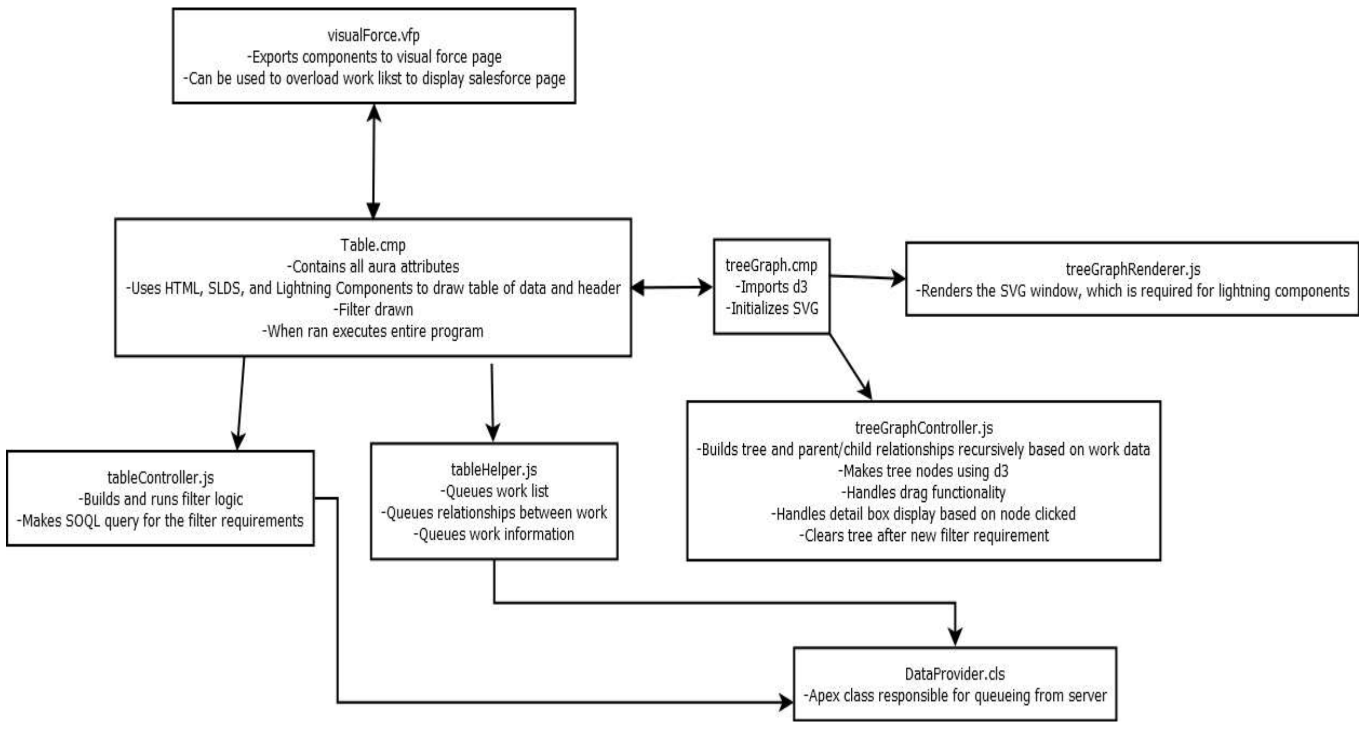
Definition of Done

Upon completion, the project will be reviewed by our client from Salesforce. If all requirements are met, the project will be implemented in conjunction with Salesforce's agile software. The program will be exported through a Visualforce page. The Salesforce web page should be accessible via the "work" tab as it should replace the current work tab via an override. Filtering the work list will generate multiple node tree graphs using the visualizations provided by d3.js. The SVG element used to store the d3 nodes should be scaled to size based on the amount of work filtered. Lastly, the user should be capable of clicking a particular work item. This should display additional details relating to the work item. For example, this may provide the work item's current sprint, description, team working on it, and other information.

System Architecture

Figure 1 displays a high level description of the project's organization. The arrows indicate what each component calls as well as communication between components.

Figure 1


visualForce.vfp
-Exports components to visual force page
-Can be used to overload work likst to display salesforce page
Table.cmp
-Contains all aura attributes
-Uses HTML, SLDS, and Lightning Components to draw table of data and header
-Filter drawn
-When ran executes entire program
treeGraph.cmp
-Imports d3
-Initializes SVG
treeGraphRenderer.js
-Renders the SVG window, which is required for lightning components
treeGraphController.js
-Builds tree and parent/child relationships recursively based on work data
-Makes tree nodes using d3
-Handles drag functionality
-Handles detail box display based on node clicked
-Clears tree after new filter requirement
tableController.js
-Builds and runs filter logic
-Makes SOQL query for the filter requirements
tableHelper.js
-Queues work list
-Queues relationships between work
-Queues work information
DataProvider.cls
-Apex class responsible for queueing from server

In order for the project to function within the Salesforce Framework, Lightning Components must be used. Lightning Components involve a combination of html, Apex, SOQL, javascript, and CSS.

Lighting components are self-contained and reusable units of web page logic. The components contain all of the html and slds formatting to display our data with a Salesforce flair. A component can also contain aura attributes, which are the equivalent of variables in other programming languages. These attributes can be passed to both javascript functions within the controller or helper as well be passed to other lightning components.

The javascript functions primarily edit or set the attributes whereas html can display the attributes. Furthermore, the attributes can hold primitives, custom objects, and more complex data structures such as maps and arrays. Lightning Components may also contain aura handlers, which are used to call functions within the corresponding controller and pass attributes from the html file to the javascript program.

The project uses handlers primarily to call javascript functions upon a component's initialization. Handlers can also call certain javascript functions when a variable is changed. For example, our project uses a handler to detect a change in the work items when a user filters data. A change in the work items being filtered will clear the current SVG elements, then re-render them with the new work items that were filtered. From there, the javascript controller updates the object array containing the work items. Upon detecting a change in the object array, the node tree is redrawn with the updated work.

The controller and helper function in a similar manner as scripts in html. From within those controllers and helpers, data can be retrieved from the Salesforce server and parameters can be set for use in the Apex class. Apex makes queries to the Salesforce server through SOQL, which is Salesforce's SQL equivalent with minor modifications.

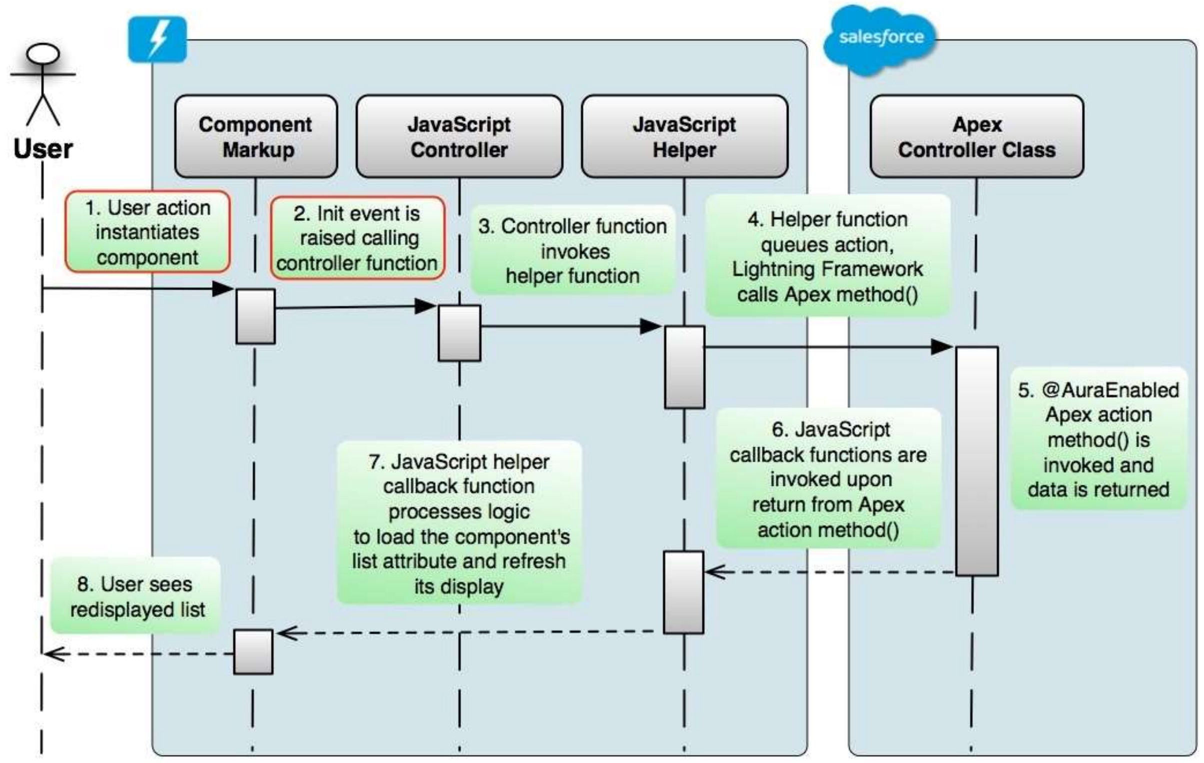
Overall, the project primarily uses js controllers and helpers to perform necessary logic for filtering work items and displaying node tree graphs. This is because logic should be done client-side in javascript rather than server-side in apex as it's necessary to dynamically change the data being used to create graphs.

Lightning events are a special type of Lightning component that detect changes upon a user's interaction with the web page. Events were used in the form of detecting change when the user enters text for the filter.

The functionality was split into different lightning components. For example, one lightning component, called "treeGraph", contained all the code for generating a tree based on SOQL queries. The results of the queries were stored into an object array, which was then used to create the table in html for styling the table. Another lightning component, named "table," serves as a top-level component which calls all the other lightning components and events needed to generate our tree and receive user actions. This better organizes our project and allows the "table" component to be converted into a VisualForce page.

The code was constructed with proper lightning architecture in mind. Given in **Figure 2** is a graph of proper lightning architecture:

Figure 2



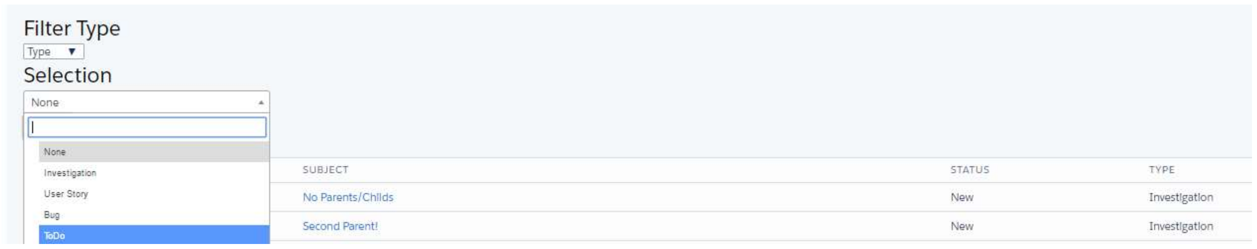
salesforce
User
Component Markup
JavaScript Controller
JavaScript Helper
Apex Controller Class
1. User action instantiates component
2. Init event is raised calling controller function
3. Controller function invokes helper function
4. Helper function queues action, Lightning Framework calls Apex method()
5. @AuraEnabled Apex action method() is invoked and data is returned
6. JavaScript callback functions are invoked upon return from Apex action method()
7. JavaScript helper callback function processes logic to load the component's list attribute and refresh its display
8. User sees redisplayed list

In addition, our client required that our filtered work changed dynamically based upon the user's work attribute and name selection. The filter enables the user to make a selection from combo boxes to determine what kind of filter they desire. The filter button redisplays the work list and generates the tree based on the current work list. Using this process enables the filter to run efficiently by not displaying unnecessary tree nodes.

Technical Design

The web page is primarily focused on improving the visual look of the work item tab. However, the filtering is also improved using a specialized search filter, as shown in **Figure 3**.

Figure 3

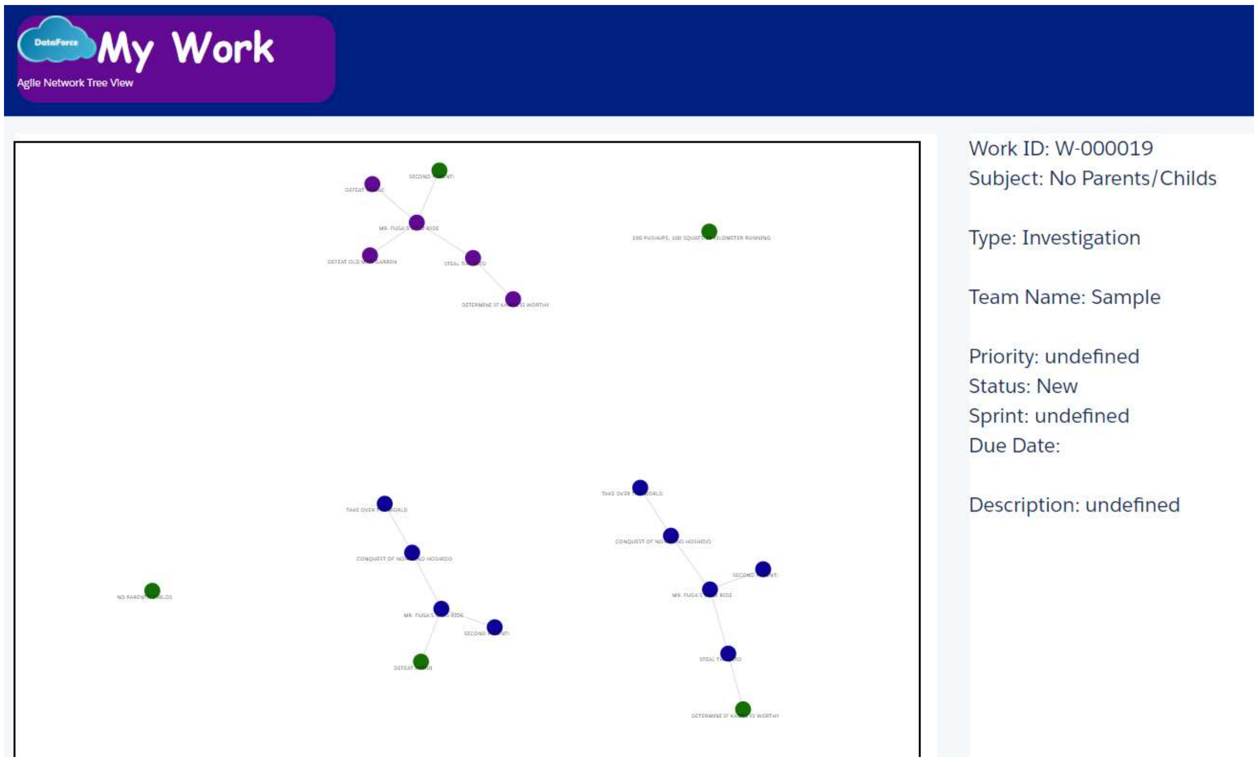


SUBJECT	STATUS	TYPE
No Parents/Childs	New	Investigation
Second Parent!	New	Investigation

The top filter allows the user to apply the agile key-word the user wants to filter their work items by. The bottom filter displays all the work items pertaining to the filter-type the user chooses within the first select bar. Furthermore, the bottom filter includes a drop down menu paired with a search bar. The drop down allows for fast navigation while the search bar helps in finding a specific item within a large dataset. Combined together, this allows for faster navigating to the desired work item list.

Figure 4 displays the tree node graph.

Figure 4



Work ID: W-000019
Subject: No Parents/Childs

Type: Investigation

Team Name: Sample

Priority: undefined
Status: New
Sprint: undefined
Due Date:

Description: undefined

Due to filters having the potential to produce hundreds of work items, the window for the graph supports scrolling to zoom in/out to view all nodes. The tree node window is actually one big SVG element wrapped in a div container. Zooming and scrolling simply rescales the SVG window, thereby making the tree node graph appear bigger or smaller. Each tree graph is its own simulation and therefore cannot interact with one another.

Decisions

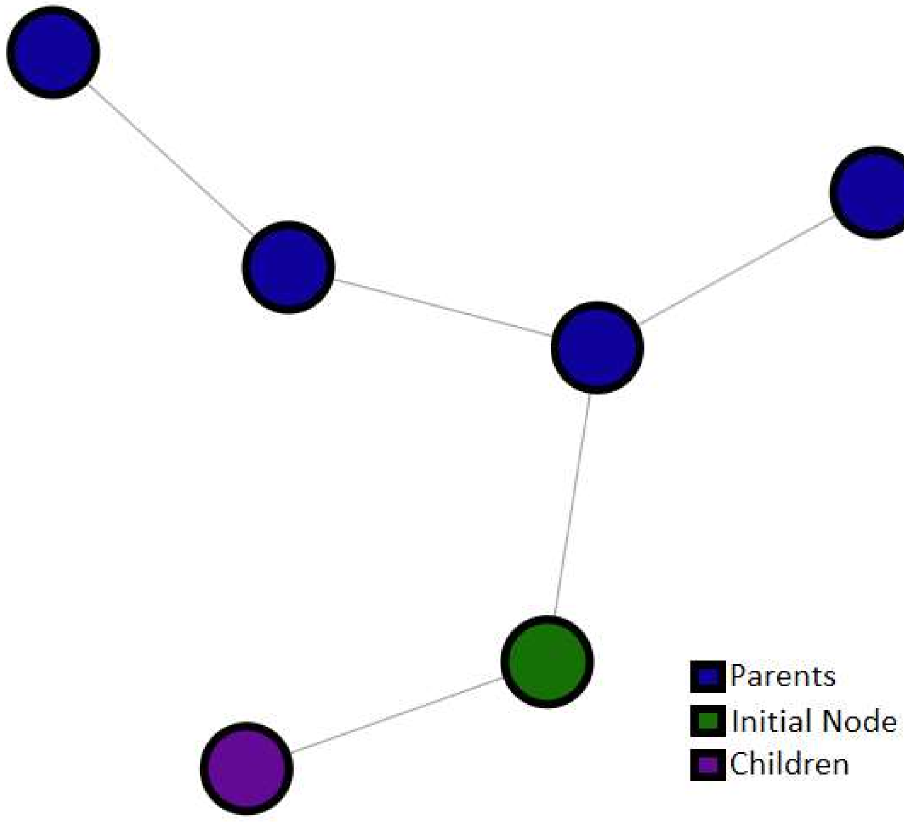
The primary decision the team made was to use a javascript library to construct a graphical UI which would create tree nodes. With many javascript libraries to consider, our options were narrowed down to Three.js and d3.js.

Three.js offers visually appealing modeling along with dynamic canvases. Furthermore, implementing Three.js would allow for more customizability and features. However, Three.js does not support any graphical trees or nodes without creating nodes from scratch. Even if custom nodes were created, Three.js would not perform very well given hundreds of work items. Performance was high on the priority list in terms of deciding between Three.js and d3.js. This library was certainly not created in hope of display big data graphs

On the other hand, d3.js was designed for representing data visualization. Our project made use of d3 Force Directed Layouts, which is convenient for displaying a tree graph structure. In Force Directed Layouts, forces keep the graph items centrally located and enable interactability between graph items. d3.js also functions quite well given a big data load. Hundreds, if not a few thousand, could still be displayed with little performance decrease, which is exactly what the program needed. However, there were some drawbacks to d3.js. Given its nature as a means to represent data it was more arduous to make a visually appealing graph. Despite this, d3.js also has the added benefit of providing a extensive list of tutorials and guides along with a very active community to help.

None of the inbuilt tree graph structures in d3 were fully equipped with all the features our tree design required. As a result, designing our own tree using a force directed layout was the best solution. Below (**Figure 5**) is a representation of the tree produced from our program.

Figure 5



Notice that the tree could contain multiple parents, children, or a combination of both. This was the primary incentive to creating our own tree node graph. The method d3.tree() could not be used as it is constrained to only using one parent and multiple children. Another incentive was that multiple graphs would need to be drawn for each work item in the filtered work array. All current tree graphs in d3.js did not support the functionality required for our program.

A new means was necessary for the implementation of a custom node tree graph. This was done by dynamically creating nodes and edges from the list of work items filtered. An associative array of parent/child relationships is passed to the client side controller, which is then put into two recursive functions. One was used to build parents and another to build children. During these recursive calls, the nodes and edges were dynamically added for the current work list. Lastly, after the data is loaded it was only a matter of using standard d3 techniques to display the data in a visual format.

Results

The program runs on edge, chrome, explorer, and firefox browsers. Tests on edge experienced minimal tear while zooming in and out on nodes, however it still functioned properly. Given no means of accessing an apple computer, the Safari browser was not tested.

All of the drag events still work on all the browsers as well. Earlier into the project drag events only worked on windows explorer. As for the data display, data was tested for accuracy by queueing it within the app; then comparing the queued data with the actual data. Our expected results were correct even when using blank fields within our database.

Our team also tested clicking actions on the nodes displayed. It was expected that clicking would provide the correct work details pertaining to the desired node. All the work displayed the correct details.

Testing the performance of our tree display with large amounts of data were never tested. For security reasons large repositories of work were unavailable for testing. However, the tree was generated only after a filter was made. Doing so would greatly reduce the load time of drawing the trees. It is with great confidence our program will still perform sufficiently given a large workload.

Unimplemented Features

- Graphical displays for both sprints and epics to encase tree nodes were not implemented. Instead sprint and epic details were added to the description box.
- Trees are not interactive with each other. Trees stay confined to their own central location.
- Due to constraints of the Salesforce system the application cannot function as a standalone application. Rather the application is within a visualforce page.
- Trees are not collapsible upon clicking. Readability would have been sacrificed.

Lessons Learned

- SVG can only be used by overwriting the afterRender function and setting the id within a lightning component
- Use static resources on the Salesforce object manager to import images and libraries
- Many Salesforce tutorials are outdated, following more recent posts is more effective
- CSS functions quite differently within Salesforce. Formatting html by wrapping statements in div styles is effectual.
- Pass all shared aura attributes once within one handler. Otherwise the entire component is called multiple times.

Extensions

Within the table, work items could also be extended such that when clicked the tree graph is properly zoomed to focus on the correct graph. Furthermore, functionality to allow work items to be edited or removed would be convenient. The same function could also be implemented to the sprints and epics.

If given a very large amount of data which one scalable SVG element doesn’t perform adequately, the tree graphs can be loaded by partitions. By this the one SVG element which the user is viewing on the web page would be rendered rather than the entire dataset. When scrolled away from that SVG element, the program will clear it then load a new one.

In addition teams, builds, or other data variables within Salesforce could be implemented within the trees. Color coding could also be extended to our program to differentiate work types.