Cytoscape Technical Documentation

Contents

Introduction

This technical document will discuss the process taken to get data displaying in Cytoscape. The process includes loading, parsing and displaying. Moreover, this document draws conclusions regarding how to optimise Cytoscape.js.

Processes

The load process focuses on how we load in the data output from our parser into Cytoscape. The parse process focuses on how we utilised that data to enable Cytoscape to understand the data. Lastly, the display process focuses on how we manipulated the parsed data to display the data in various layouts and grouping.

Load

Cytoscape has access to the data through using NetworkRequest.js. This is imported into Cytoscape.js labelled as data. NetworkRequest.js uses the assets directory to get the data and NetworkRequest.js stores the element data and style data as variables. The element data is retrieved through a promise to ensure all the element data is ready before storing it. The style data is retrieved by fetching data.cycss then storing it as text. Lastly, NetworkRequest.js will store the names of all special nodes for later utilisation for changing layouts by special nodes.

The design choice to keep all the data loading inside of one class is for good code practice. This decouples Cytoscape.js from loading the data since the job of Cytoscape.js is to display data, not load it. Moreover, this approach helps with testing because this is the only place in code where loading takes place. Therefore, for future developers it will help with building on top of the existing work.

Parse

Parsing the survey data into Cytoscape format is done at build time. The end user is simply served the cytoscape data in their browser, as opposed to being exposing the CSV data to process whenever the webpage is loaded.

Using Git pipelines, there is a python parsing script that collects all of the CSV data and turns it into Cystoscape data whenever you upload new CSV data onto your repository. This process requires the survey data to be formatted correctly as discussed in the CSV-Parser Documentation. Once done, it will then deploy the new data to your Connected Worlds website.

After loading the data, `StyleCytoscape.js` is called, which applies CSS styles to the nodes based on the `cystyles.json` and `colors.json` files. It parses the JSON in those files into CSS which is then applied to the nodes. It will apply styles based on what the type of the node is and what overrides have been passed in from when it is called from `Cytoscape.js`.

The design choice behind seperating the styles for cytoscape, and the styling for the rest of the page out into two different files was to be able to change the different styles without having to figure out what styles belonged to what part of the page. We have the styles in a JSON file so that we can have many different styles, for example the styling for many different types of nodes, in one file without it becoming cluttered and difficult to read and use.

Display

We have utilised the several layout designs to be applied to any kind of element type. This allows each special node, such as projects, to be viewed in each layout on offer. The semi-circle layout can be focused around schools instead of only projects. This same applies to the many-circles layout which can now also focus on projects. This means an end user can configure how they wish to display their data.

There is a selector on the user interface for picking which layout to display and which type of special node to focus the layout on. Anytime the layout and focus type changes, Cytoscape will be notified through usage of mobx-react decoration to observe the class which stores this information. There are two steps after Cytoscape is notified, which will be discussed in the next two paragraphs.

`LayoutFactory.js` will recompute the new layout every time the details of the new layout get changed in `CytoscapeStore.js` through `View.js` or `DropDownMenu.js`. It does this by calling `computeLayout(layout,focusType)` from `LayoutFactory.js`.

- *Layout* is the type of layout that has been selected by the end user. For example, `showSegment` represents the semi-circle layout.

- *Focus Type* is the type of special node that the layout is displaying. For example, `school` means it will display around school nodes for the current layout.

Once `computeLayout(layout,focusType)` has been executed, then as mentioned earlier, mobx-react notifies `Cytoscape.js` that the layout has changed. Thus, `Cytoscape.js` will call `autorun()` to update the new layout on screen. It does this by calling `layout.run()` on the new layout.

The design choice behind this is to help with generalising Connected Worlds. Alex developed the original prototype with the assumption that the domain of the data will always be the same. We considered the possibility of you expanding the domain of the data. For example, if you decided to add data for scholarships, any of the layouts can focus the people nodes around scholarships. This examples highlights how Connected Worlds can now displays all different sorts of data onto these layouts.

Conclusions for Optimising Cytoscape

Cytoscape.js could be optimised in the future by changing a lot of code that repeats often for no reason. For example the `checkYears()` function, which is used for checking if a node is in the selected range of years, is run whenever the selected node is changed, when the layout is changed, and when the selected years are changed. It would be much more efficient to only have it run when the selected years are changed.

To make improving and optimising Cytoscape.js easier in the future, the file could be split up into smaller scoped classes, which will assist in narrowing down any potential bugs caused by any future improvements or optimisations. It will also help the process of optimising directly as doing this will mean that finding certain parts of the code will be much easier to find in lots of smaller, correctly named files, rather than one huge file.