Visualization Linter

Static and runtime check tool for D3.js

Youssef Sherif
Wei Zheng

Background

- Why do we need a visualization linter?
  - To help the “average Joe” data visualization user adhere to visualization best practices.
- Why two tools?
  - Each of the static analysis tool and run-time library have its own uses, pros, and cons.

Static Analysis Tools

- Not meant to check data-related issues
  - Why? Because data is dynamic
    - Example: Http request from a backend server
- Meant to check for logical problems

Runtime Tools

- Has access to data during runtime
- Cons
  - Warnings and Errors are displayed on runtime (not immediately)

Programming Language and Framework:

- JavaScript
  - Web applications are on the rise
    - JavaScript is the default web language
- D3.js
  - Most popular
  - Open source

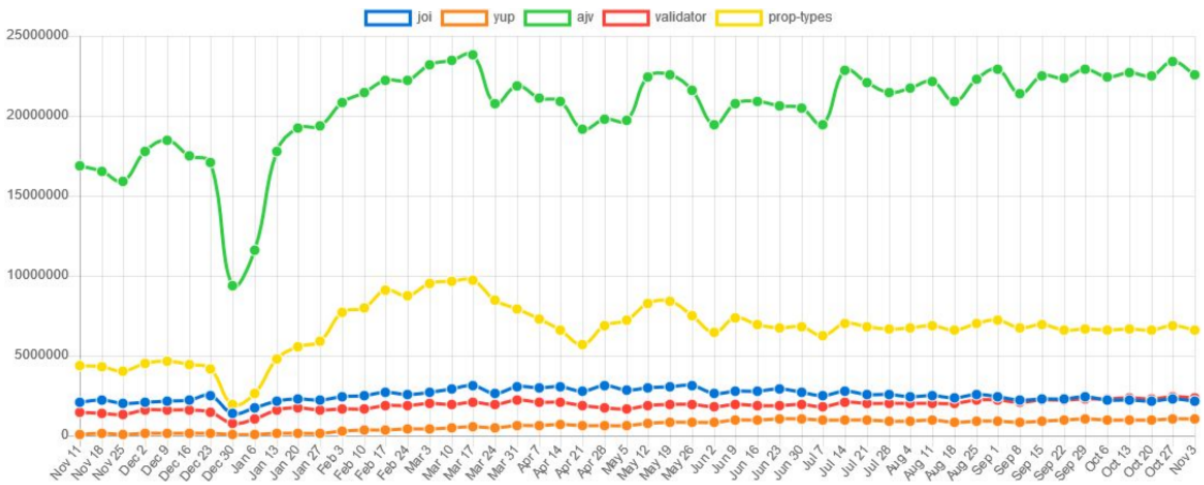
Previous Works

Andrew Mcnutt’s Vislinter

- Run-time library checker for Matplotlib
- He proposed a long list of data visualization rules
- Implemented few of them
- Wrote “Linting for Visualization: Towards a Practical Automated Visualization Guidance System” paper

Existing JavaScript Runtime libraries checkers

- Check most popular run time checker libraries on npm
- We are planning to check the public API for at least one of these libraries to conform to best practices for runtime library checkers



Matplotlib vs D3.js

Matplotlib	**D3.js**
High level library	Low-level library
Less control	More control
Easier to build simple visualizations	Harder to build simple visualizations
Can easily infer statically the visualization the user wants to build	Hard to infer statically the visualization the user wants to build

Implementation

- Static Analysis Tool
  - ESlint plugin
    - ESlint is the most widely used JavaScript pluggable static linter

- Run-time library checker
  - A regular npm package

What are our personal expertise?

Youssef

- Worked as a full-stack web developer
    - Used JavaScript and other JavaScript libraries
- Partially built static analysis tools

William

- Experienced in Data Analysis with Python, and visualization tools, including Matplotlib and Seaborn
- Built a web app using vanilla JavaScript
- Applied machine learning algorithms with Java

What we are supposed to do?

Youssef

- Build the static analysis tool
- Structure the runtime checker library and set the public API
- Set webpack and npm scripts to be used for the library

William

- Select the rules for runtime checking
- Implement the runtime check part

Resources for Rules for Best Practices

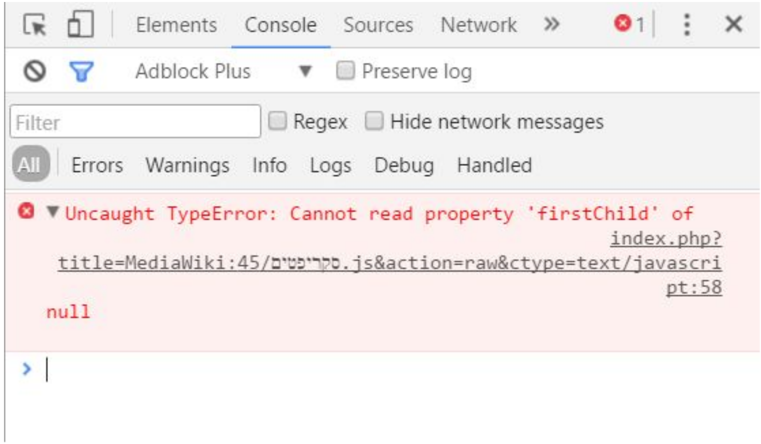
- Tamara’s book “Visualization Analysis and Design”
  - Example: order of effectiveness
- The Visualization Guidelines Repository
- Yan Holtz’s online guideline

Static analysis tool

- Run a command line prompt
- Have the IDE detect it if we use ESlint

Run-time library checker

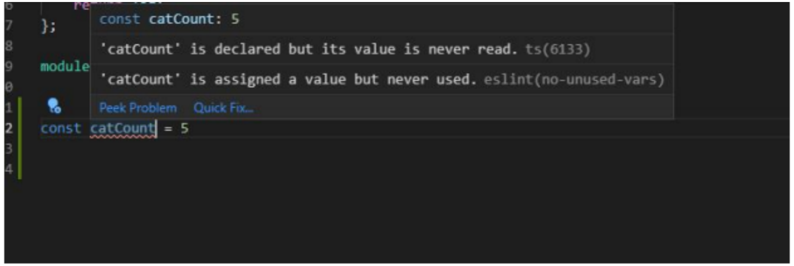
- Check console warnings



Future Advancements

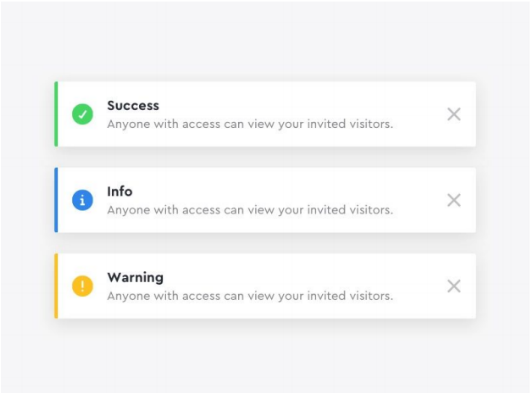
Static analysis tool

- IDE extension



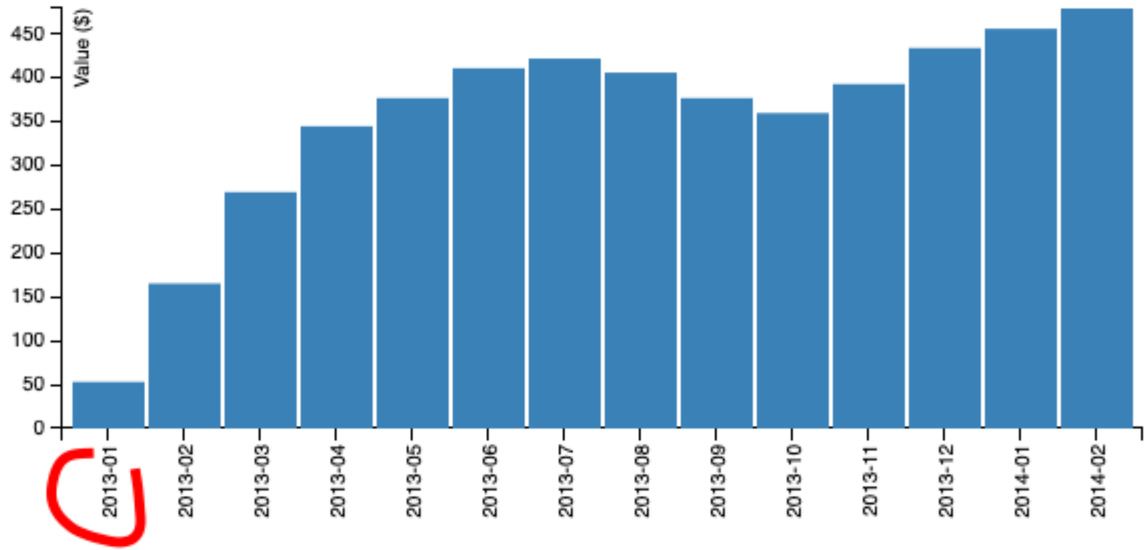
Run-time library checker

- Unobtrusive toasts



Attempt to implement a rule

“No horizontal labels”



Tried to implement the rule statically

Current Solution: use node.js to parse the entire js file written by users as string and detect key words such as `.selectAll("text")` , `.attr("transform", "rotate(-90)")` to detect the part which users try to deal with the labels of x-axis.

Problem: This would not work if the text is the same as it is. For example using a variable and the string ‘text’ would make our tool fail. This is where runtime checks shine

```
▶Use horizontal labels. Avoid steep diagonal or vertical type,  VM639:1
 as it can be difficult to read.
```

Tried to implement the rule runtime with svg

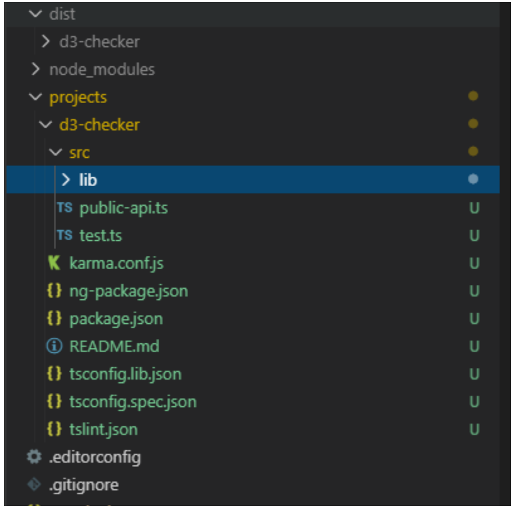
Current Solution: use node.js to parse the entire js file written by users as string and detect key words such as `.selectAll("text")`, `.attr("transform", "rotate(-90)")` to detect the part which users try to deal with the labels of x-axis.

Problem: This would not work if the text is the same as it is. For example using a variable and the string ‘text’ would make our tool fail. This is where runtime checks shine

```
▸Use horizontal labels. Avoid steep diagonal or vertical type,  VM639:1
 as it can be difficult to read.
```

Runtime Library

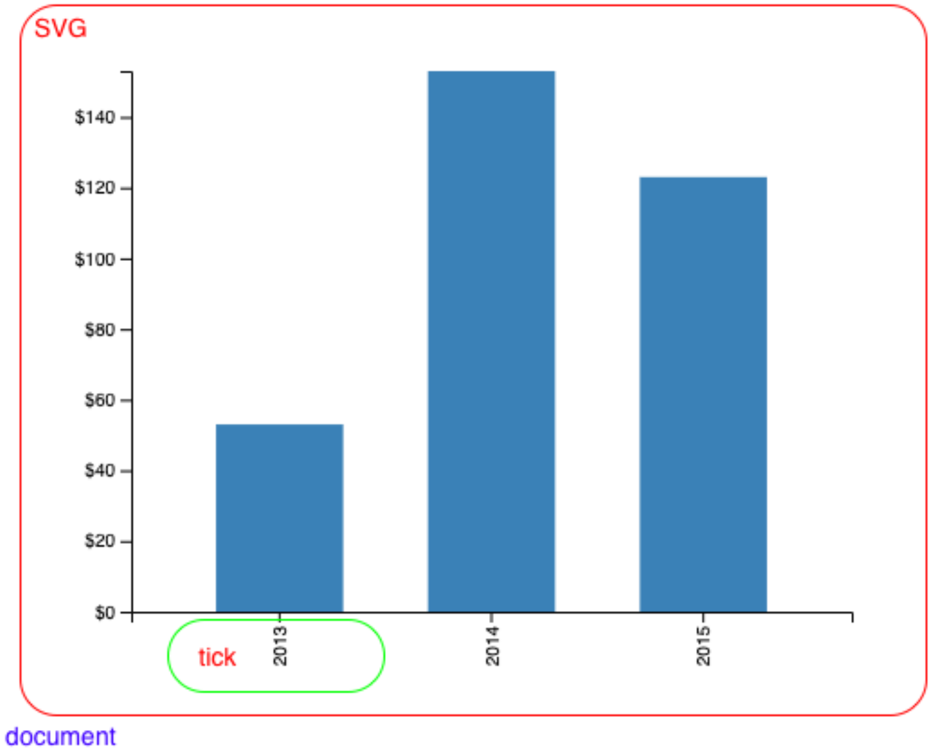
- Typescript instead of JavaScript
  - We think types would be of much help since we would need to deal with the data structures of the D3
- Karma and Jasmine for testing
  - We might replace that with Jest if the extra speed is worth the change



Dealing with SVG

- Till now, it is almost impossible for us to detect from the SVG data structure the graph's features

Dealing with SVG



Check the rotation for the tick in green circle in the following way:

```
var svg = document.getElementsByTagName("svg");
var label = svg.querySelector(".tick > text");
var valueStr = label.getAttribute("transform");
```

In this way, we can check d3 at run time.

```
▶Uncaught TypeError: svg.querySelector is not a function    test3.js:50
    at checkTick (test3.js:50)
    at test3.js:57
```

Dealing with SVG

Weakness for this method:

Finding out related properties is the key to implement rules, however, it turns out difficult.

Fragile when users intentionally change the class of DOM nodes generated by d3.

Deal with other elements instead of SVG?

- Since dealing with SVG is hard, we thought about asking the user to manually provide in code features of the graph like
    - The type of graph
    - X-axis of the graph if applicable
    - Y-axis of the graph if applicable
- Unlike SVG's data structure, data structures of other graph's elements such as the axis can are easier to parse and comprehend

```
D3Checker.lint({
    type: 'barChart',
    xAxis: xAxis,
    yAxis: yAxis
})
```

```
xAxis ▾ n()
    arguments: null
    caller: null
  ▸ innerTickSize: function innerTickSize()
    length: 1
    name: "n"
  ▸ orient: function orient()
  ▸ outerTickSize: function outerTickSize()
  ▸ prototype: Object { … }
  ▸ scale: function scale()
  ▸ tickFormat: function tickFormat()
  ▸ tickPadding: function tickPadding()
  ▸ tickSize: function tickSize()
  ▸ tickSubdivide: function tickSubdivide()
  ▸ tickValues: function tickValues()
  ▸ ticks: function ticks()
  ▸ <prototype>: function ()
```

Problems with this approach

- A D3 user might want to have a customized graph that does not fit in one of the types we support like a “bar chart” or “pie chart”
- A hassle for the user to provide all of these data
- Fragile since the user via D3 can transform the svg itself later and these data structures might not be representatives of the real graph

A third approach

Fork D3.js and manipulate the code inside to add checkers

- This approach needs a lot of time of studying D3 internal code
- We abandoned this approach

Final Approach: Use Chart.js instead of D3

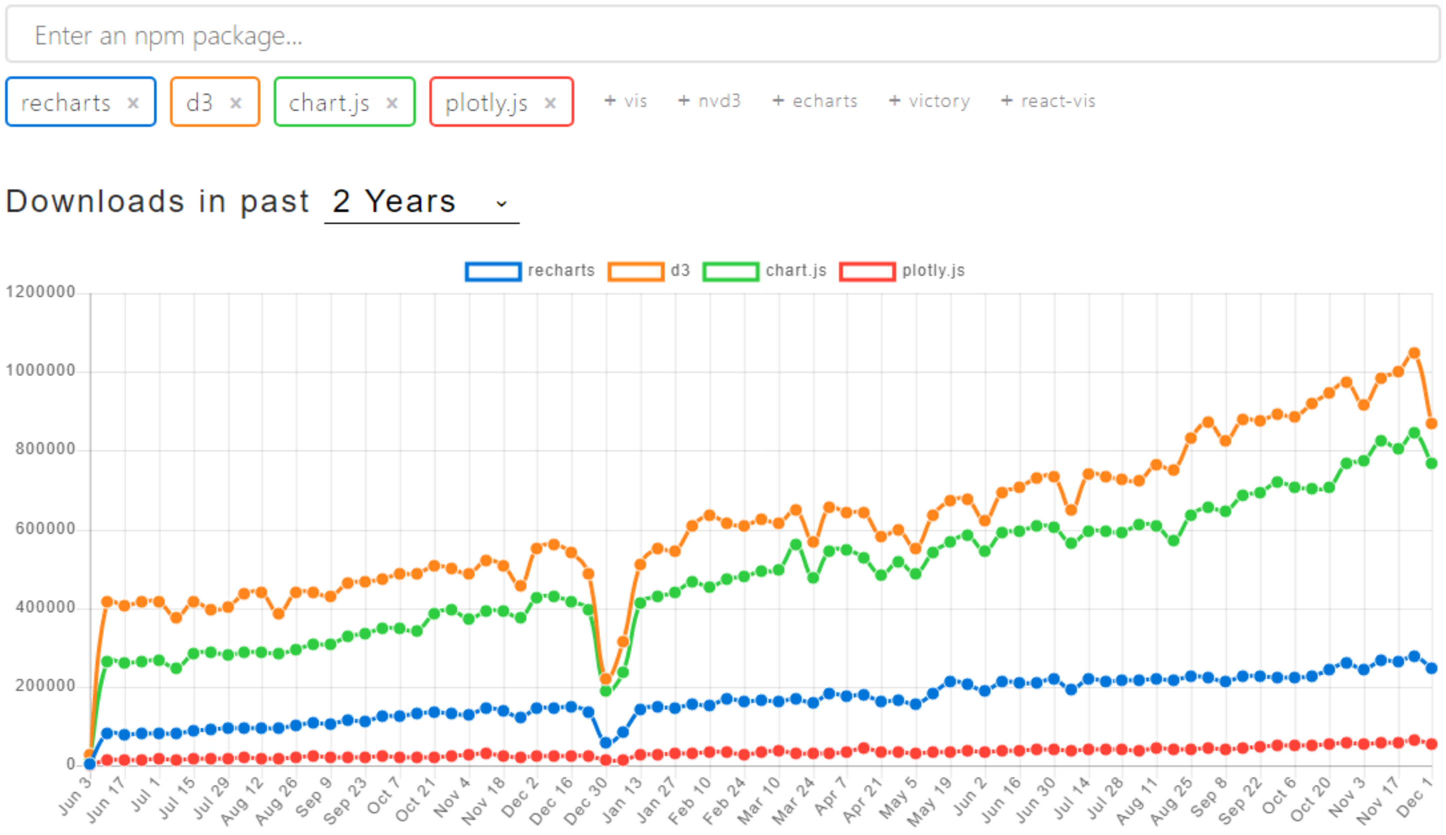
- It appears that adding a runtime and static analysis checker might be
  - Fragile/inaccurate
    - Lots of false positive
- Reasons
  - It is hard to know the intent of what the user wants to visualize from the code or even the exposed data structure
    - This might make it hard/almost impossible to implement most of the rules
  - The variety and possibilities of visualizations that might be drawn via D3 is almost infinite
    - Lots of rules would be inconvenient for lots of cases
      - Example:

Different Approaches Comparison

Features/ Approach	Parse SVG in D3	Fork D3 and modify internal code	Parse Smaller Elements in D3	Replace D3 with another library
Ease of detecting graph's properties	Possible	Unknown	Hard	Easy
Fragility	Medium	Unkown	Fragile	Not fragile
User's ease of use	Relatively easy	Very easy	Hard	Relatively easy
Time needed to invest in the project	A lot	Extreme	A lot	Reasonable
Probability pursing approach	Average	Low	Low	High

Why Chart.js?

- Easier data structure interface
- Most adopted after D3 (based on best of our knowledge)



Enter an npm package...
recharts ×
d3 ×
chart.js ×
plotly.js ×
+ vis
+ nvd3
+ echarts
+ victory
+ react-vis
Downloads in past 2 Years
recharts
d3
chart.js
plotly.js
1200000
1000000
800000
600000
400000
200000
0
Jun 3
Jun 17
Jul 1
Jul 15
Jul 29
Aug 12
Aug 26
Sep 9
Sep 23
Oct 7
Oct 21
Nov 4
Nov 18
Dec 2
Dec 16
Dec 30
Jan 13
Jan 27
Feb 10
Feb 24
Mar 10
Mar 24
Apr 7
Apr 21
May 5
May 19
Jun 2
Jun 16
Jun 30
Jul 14
Jul 28
Aug 11
Aug 25
Sep 8
Sep 22
Oct 6
Oct 20
Nov 3
Nov 17
Dec 1

ChartJS Simpler Data Structure

For example, I can directly know from the data structure that the chart is a “bar chart” the title is at the top of the chart

```
▼Chart.Controller
   animating: false
 ▶boxes: (4) [ChartElement, ChartElement, ChartElement, ChartElement]
 ▶chart: Chart {config: {…}, ctx: CanvasRenderingContext2D, canvas: canvas#myChart, width: 400, height: 400, …}
 ▶chartArea: {left: 33.34765625, top: 32, right: 400, bottom: 328.5031314380094}
 ▶config: {type: "bar", data: {…}, options: {…}}
 ▶events: {mousemove: ƒ, mouseout: ƒ, click: ƒ, touchstart: ƒ, touchmove: ƒ}
   id: 0
 ▶legend: ChartElement {ctx: CanvasRenderingContext2D, options: {…}, chart: C…t.Controller, legendHitBoxes: Array(1), doughnutMode: false, …}
 ▶options: {responsive: false, responsiveAnimationDuration: 0, maintainAspectRatio: true, events: Array(5), hover: {…}, …}
 ▶scales: {x-axis-0: ChartElement, y-axis-0: ChartElement}
 ▶titleBlock: ChartElement {ctx: CanvasRenderingContext2D, options: {…}, chart: C…t.Controller, legendHitBoxes: Array(0), maxWidth: 400, …}
 ▶tooltip: ChartElement {_chart: Chart, _chartInstance: C…t.Controller, _data: {…}, _options: {…}, _model: {…}, …}
   data: (...)
 ▶get data: ƒ ()
 ▶__proto__: Object
```