AChart Creator: A Command-Line Tool to Create Accessible Charts with D3 and Node

Inti Gabriel Mendoza Estrada, Mirza Kabiljagic, Stefan Rajinovic, and Aleksandar Stojicic

Institute of Interactive Systems and Data Science (ISDS),
Graz University of Technology
A-8010 Graz, Austria

31 Jan 2020

Abstract

AChart Creator is based on SVG-Describler. It is a tool written on TypeScript intended to read data from a CSV file and create an SVG chart with D3. Furthermore, it adds `aria` attributes to the SVG that allows it to be read by a screen reader. AChart Creator creates accessible SVG charts, and it is intended to be used alongside AChart Reader and be compatible to Describler. In this report we will give an in-depth overview of the tool, its technologies, as well as the incremental update we have added to it.

© Copyright 2019 by the author(s), except as otherwise noted.

This work is placed under a Creative Commons Attribution 4.0 International (CC BY 4.0) licence.

Contents

List of Listings

Chapter 1

Introduction

AChart Creator creates accessible SVG from CSV files. It is a tool that uses Accessible Rich Internet Applications (ARIA), was built from SVG-Describer, and is intended to be used with AChart Reader and (currently) Describler. Currently there are some issues needed to be addressed of the tool. In this report we will discuss these issues and how we solved them, as well as any Quality-of-Life improvements.

SVG stands for Scalable Vector Graphics. The currently browser- supported SVG version is 1.1. It is a vector-based format in which different shapes contain at least one markup element. SVG code is remarkably similar to HTML and, like HTML, it can be styled using CSS and manipulated through JavaScript (and by extent TypeScript).

1.1 Accessible Rich Internet Applications (ARIA)

ARIA is "set of attributes that define ways to make web content and web applications (especially those developed with JavaScript) more accessible to people with disabilities. It supplements HTML so that interactions and widgets commonly used in applications can be passed to assistive technologies when there is not otherwise a mechanism. For example, ARIA enables accessible navigation landmarks in HTML4, JavaScript widgets, form hints and error messages, live content updates and more" [MDN contributors 2020].

As mentioned before, SVG files contain markup attributes, thus when creating an SVG file with AChart Creator, we are able to inject ARIA attributes to it. An example can be seen in Listing 1.1. The `aria-charttype` role defines the chart type, `tabindex` sets the index of this element that would be accessed by pressing the TAB button. As the `tabindex` is 0, when pressing TAB for the first time, the chart will be highlighted and the screen reader would understand that a chart is currently being highlighted. Due to the attribute `aria-labeledby`, the screen reader knows that the title and description is being defined by the `<title>` and `<desc>` tags, and it can then promptly and correctly read this information out loud.

```
<g id="PieRoot" role="chart" aria-charttype="pie" tabindex="0"
 aria-labelledby="title desc">
<title id="title"></title>
<desc id="desc"></desc>
<text role="heading" tabindex="0" x="200" y="25" text-anchor="middle">
</text>
```

Listing 1.1: Example ARIA-roles.

Without these ARIA attributes, a screen reader would not be able to read them. AChart Creator is properly able to extract and set ARIA attribute values from a CSV file. Tools like AChart Reader and Describler enhance this even more.

1.2 Describler and AChart Reader

Currently Describler and AChart Reader serve relatively the same purpose. Describler was created by Doug Schepers and is currently an experimental prototype screen reader for SVG files [Schepers 2020]. AChart Reader is currently developed by Christopher A. Kople with contributions by Inti Gabriel Mendoza Estrada.

These tools are able to interpret ARIA attributes (making SVG files screen readble). Furthermore, they are able to generate statistics of the chart, if desired. Information like the highest value, the average, the range and domain of the chart, etc. . .

Describler has a web application at describler.com. AChart Reader is built similarly to AChart Creator.

1.3 AChart Creator

AChart Creator is a Node-based application. It uses `gulp` and is written in TypeScript. The `gulp` task `acreate` reads from the commandline arguments and passes it to the appropriate TypeScript recipe. This TypeScript recipe reads a given CSV file (`gulp` specifies the paths and everything) and, using D3, creates an SVG image with the appropriate ARIA attributes. The `gulp` task then transpiles the TypeScript code into JavaScript code and then runs this JavaScript code to output the proper SVG file. An example on running AChart Creator to create a pie chart from a CSV file can be seen in Listing 1.2.

```
gulp acreate --file pie.ts --dataset fruit-pie.csv
```

Listing 1.2: Example running AChart Creator

When we started improving AChart Creator, AChart Creator would output SVG code in minified form. When read with a text editor, the code would be written in a single line. This is useful to save disk space, however, extremely difficult to be read by a human. If the user wants to make changes to the SVG, finding the right tag and attribute of interest proves difficult. An example snippet of this code can be seen in Listing 1.3. We have since fixed this, as discussed in Chapter 2.

```
<g id="ChartRoot" role="chart" aria-charttype="line" tabindex="0" transform="
    translate(50,25)" aria-labelledby="title desc"><title id="title" role="heading">
    Title</title><desc id="desc">Description</desc><g><rect id="backdrop" x="-75" y=
    "-50" width="500" height="500" fill="#fff"></rect><rect role="chartarea" width="
    400" height="400" fill="none"></rect></g><text x="200" text-anchor="middle">
    Title</text><g id="xScale" role="xaxis" aria-axistype="category" tabindex="0"
    aria-valuemin="100" aria-valuemax="465" transform="translate(0,400)" fill="none"
     font-size="10" font-family="sans-serif" text-anchor="middle"><path class="
    domain" stroke="currentColor" d="M0.5,6V0.5H400.5V6"></path><g class="tick"
    opacity="1" transform="translate(0.5,0)"><line stroke="currentColor" y2="6"></
    line><text fill="currentColor" y="9" dy="0.71em" role="axislabel" id="x-2011"
    style="text-anchor: middle;">2011</text></g>
```

Listing 1.3: Example Initial (Unreadable) SVG.

Chapter 2

Beautifying Tools

As mentioned in Chapter 1, AChart Creator initially created human-unreadable SVG code (example show in Listing 1.3). To fix this, we looked for tools online that might allow us to fix this. We found 4 tools representative of the myriad of tools online: Uniminify (web-based), JS Beautifier (JavaScript tidier), Scour (external Python library), and SVGO (Node-based tidier). We discuss these tools and their usefulness overall and for our project in the remaining of this chapter.

2.1 Unminify

Unimify is a Web-based tidier. It is hosted in https://unminify.com/. This tool was written on JavaScript and JQuery and developed by Media4x. The use case defined by Media4x is to unminify - to unpack, deobfuscate - JavaScript, CSS, and HTML code [Media4x 2020].

This tool does not allow us to decide the way code is unminified. For example, setting our desired indentation size is not possible. Instead, the tool decides the size and the proper indentation is based on tag hierarchy. Furthermore, since our tool is Node-based and is not dependant on cloud resources (it is run locally), integrating Unminify is only possible through server calls to its serve, it is not very useful for our project.

For "quick and dirty" jobs, this tool delivers its promise of unminifying JavaScript, CSS, and HTML (and SVG by extention) to industry standard code (4 spaces indentations).

2.2 JS Beautifier

JS Beautifier is another code cleanup and tidier Nodejs based tool. JavaScript was used to write this tool, as well as Nodejs. Main usage of this tool is to reformat and fix indentations or add new ones if needed, unpack

There are three ways of using it, either installing it via pip, npm or to include the .js files in the project index.

Installing with npm:

```
    npm -g install js-beautify
```

Installing with Python:

```
    pip install jsbeautifier
```

The tool was created by Einar Lielmanis and maintained by Liam Newman[Einar Lielmanis 2016]. There is also a Web-based UI where the tool can be used, by accessing https://beautifier.io/

- -l, –indent-level Specify initial indentation level

- -n, –end-with-newline End output with new line
- -C, –comma-first Put commas at the beginning of new line instead of end
- –indent-empty-lines Keep indentation on empty lines

Also, there are command-line flags for the JavaScript and Python scripts:

- -f, –file Input file(s) (Pass '-' for stdin)
- -r, –replace Write output in-place, replacing input
- -o, –outfile Write output to file (default stdout)
- –config Path to config file
- –type [js|css|html] ["js"] Select beautifier type (NOTE: Does *not* filter files, only defines which beautifier type to run)
- -q, –quiet Suppress logging to stdout
- -h, –help Show this help
- -v, –version Show the version

2.3 Scour

The Scour is the tool written in Python with the main job is to optimize and clean SVG code. Romoving unnecessary data and optimizing structure the size of vector graphic is reduced. Also Scour can be used in order to create a streamlined vector graphics. This vector graphics can be used for further processing as well publishing and sharing in order to use in the web. There are a lot of information, redundant information, that were created by producing SVG code by the most of SVG editors, so in order to render and remove this information the Scour is used.
Scour was developed in 2010 by Jeff Schiller and Louis Simard. In 2013 the whole project is published on GitLab and is maintained by Tobias Oberstein and Eduard Braun. It is open-source and licensed under Apache License 2.0 [Tobias Oberstein 2020]

Installation:

```
    t pip install scour
```

Standard usage:

```
    scour -i input.svg -o output.svg
```

Better (for older versions of Internet Explorer):

```
    scour -i input.svg -o output.svg --enable-viewboxing
```

Maximum scrubbing

```
     scour -i input.svg -o output.svg --enable-viewboxing
 --enable-id-stripping --enable-comment-stripping --shorten-ids
 --indent=none
```

Maximum scrubbing and a compressed SVGZ file:

```
    scour -i input.svg -o output.svgz --enable-viewboxing
--enable-id-stripping --enable-comment-stripping --shorten-ids
--indent=none
```

```
?xml version="1.0" encoding="UTF-8"?>
<svg role="graphics-document" viewBox="0 0 800 600" xmlns="http://www.w3.org/2000/
    svg">
 <style>.bar {fill: steelblue; }</style>
 <g id="ChartRoot" transform="translate(100,50)" aria-charttype="bar"
     aria-labelledby="title desc" role="chart" tabindex="0">
  <title id="title" role="heading" tabindex="0">Most Popular Fruits</title>
  <desc id="desc">Most Popular Fruits in 2019 given in percentage</desc>
  <rect id="backdrop" x="-100" y="-50" width="800" height="600" fill="#fff"/>
  <rect width="600" height="400" fill="none" role="chartarea"/>
  <text x="300" y="-25" text-anchor="middle">Most Loved Fruits in 2019</text>
  <g id="xScale" transform="translate(0,400)" fill="none" font-family="sans-serif"
      font-size="10" text-anchor="middle" aria-axistype="category" role="xaxis"
      tabindex="0">
   <path class="domain" d="M0.5,6V0.5H600.5V6" stroke="currentColor"/>
   <g class="tick" transform="translate(56.757)">
    <line y2="6" stroke="currentColor"/> ...
```

Listing 2.1: Example Scour SVG Output

After execution the example snippet of this code can be seen in Listing 2.1. In comparation to the Listing 1.3

With this tool we get a wanted output, but we can not use python with our TypeScript. As well the Scour SVG Ouput works on Describler.

2.4 SVGO

SVG is optimizing Nodejs tool for SVG files which contains sometimes a lot of redundant and not useful informations. The architecture of SVG optimizer is plugin-based and every plugin offers different optimization. With this SVG tool is easily and safely possible to remove some non-optimal values in order to not affect the SVG rendering [Belevich 2020].

Plugins:

- There are about 50 plugins and some of them are automatic enabled and some not
- They can move attributes, modify contents and SVG elements and any wanted operation

Config file:

- Default config file includes all plugins that have specific positions in the plugin list
- Parameter name from config file is used to enable/disable some plugin action
- Every user can setup their parameters in config file and then these parameter will be accessible in plugin
- Using –config command line and parameter full: true enable using your personal settings

SVGO applies all plugins from config file to AST data. AST data is the converted content of SVG file. Every user can decide to use optimizer for all SVG files or just for exactly one [Belevich 2020].

Therefore, plugins are separated into three groups:

- perItem
- perItemReverse

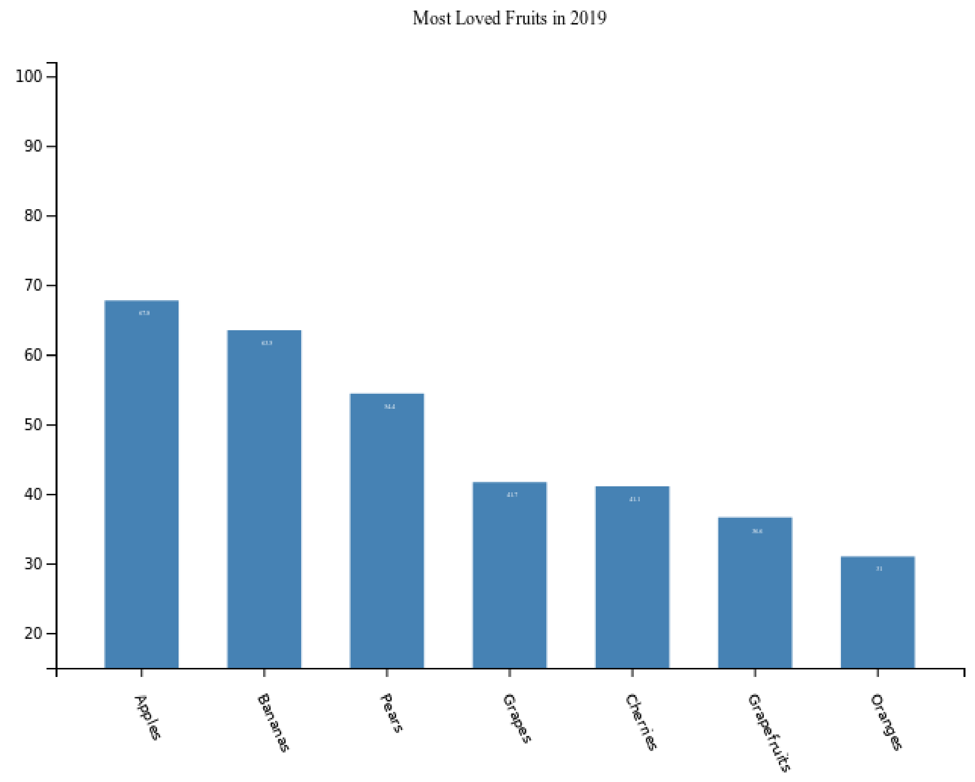

Figure 2.1: Bar recipe optimized with SVGO tool [Image taken by the author.]

- full

There is a long API list for potential contributors.

List of active plugins:

- cleanupAttrs: remove attributes from newlines or repeating spaces
- inlineStyles: modify <style> elements by moving and merging into style attributes
- removeDoctype: with this plugin doctype is removed
- removeXMLProcInst: wit this plugin all XML processing instructions are removed
- removeComments: cleanup of comments
- removeMetadata: deleting <metadata> tag
- removeTitle: deleting <title> tag
- removeDesc: deleting <desc>
- removeXMLNS: deleting xmlns attributes
- convertColors: with this plugin the color is converted from rgb() to #rrggbb

Installation:

```
const svgmin = require('gulp-svgmin')

gulp.task('pretty', function () {
    return gulp.src('build/svg/bar.svg') // file to beautify
        .pipe(svgmin({
            js2svg: {
                pretty: true
            }
        }))
        .pipe(gulp.dest('./out')) // output directory
});
```

Listing 2.5: SVGO gulp Task.

```
        [sudo] npm install -g svgo
```

Listing 2.2: SVGO Command-Line Installation

Usage:

```
            svgo [OPTIONS] [ARGS]
```

Listing 2.3: SVGO Command-Line Usage

Options:

- -h, –help : Help
- -i INPUT, –input=INPUT : Input file, "-" for STDIN
- -o OUTPUT, –output=OUTPUT : Output file or folder (by default the same as the input), "-" for STDOUT
- –config=CONFIG : Config file or JSON string to extend or replace default
- –disable=PLUGIN : Disable plugin by name, "–disable=PLUGIN1,PLUGIN2" for multiple plugins
- –enable=PLUGIN : Enable plugin by name,"–enable=PLUGIN3,PLUGIN4" for multiple plugins

Arguments: INPUT : Alias to –input

Example with files can be seen in Listing 2.4:

```
    svgo test.svg
    svgo *.svg
    svgo test.svg -o test.min.svg
```

Listing 2.4: SVGO Command-Line Examples

2.4.1 SVGO gulp

SVGO has a `gulp` flavour. It can be used as a `gulp` task. To include this in your `gulpfile.js`, refer to Listing 2.5. This tool optmizes the SVG as well as properly beautifying it.

Even though SVGO works with Describler, the `gulp` flavour does not. It unfortunately removes some ARIA roles and attributes it deems useless, which are paramount to the functionality of our tools AChart Creator and AChart Reader.

Chapter 3

Our Parser

We have made improvements on AChart Creator over the past six weeks. We have also fixed and found some bugs, as well as adding some Quality-of-Life changes. We research different kinds of Beautifying Tools. This tools are described in 2. After research we come to some conclusion. The Unminify and Scour are incompatible with local development. JS Beautifier only works with JavaScript files on node it is not possible to add new lines with SVGO. Make more mess and unused SVG output. SVGO gulp removes attributes it deems useless.

Tools are not useful for us, different environments, don't add newlines and make it unusable with describler (and AChart Reader). For all of this reasons we need to capture the string written into the SVG file and tidy it ourselves. This leads us to make our own beautifier. For that we use JavaScript integrated in TypeScript. Before we write out the SCG file we call some functions to beautify the code. The calling routine You can see in 3.1

```
//-------------------------------------------------------
// PRINT OUT:
fs.writeFileSync('build/svg/' + fileName + '.svg', remove_commas(parse_lines(parse(
    d3.select(doc).select("body").html()))));
```

Listing 3.1: Example Scour SVG Output

The main itea is to take a input SVG code and to add new lines to end of each element ̈<>.̈ In order to do this job we implemented the function parse. In the 3.2 you can see how we did it.

After put new line after each closed element, we need to beautify the code further. So we take the return value from function parse and try to add new line if the line length is bigger than 73 chars, as well we need to take care that none of elements or identifier want be broken. For this we implemented the function parse lines that we can see in 3.3

After all parsing because we played with lines and strings in JavaScript we need to delete all unnecessary commas. We implemented small function to do this. We can see in 3.4

After all this parsing we get clean and readable code. Next improvements would be to insert tabs for each element and delete some unused SVG code reproduced by SVG editor.

```
function parse(text_to_parse){

  var output;
  for (var i = 0; i < text_to_parse.length; i++) {
    if(text_to_parse[i] == "<" && text_to_parse[i-1] == ">")
      {
        text_to_parse = splitValue(text_to_parse, i);

      }

  }
  return text_to_parse;
}
```

Listing 3.2: function parse

```
function parse_lines(text){
var lines = text.split('\n');

  for(var i = 0;  i < lines.length; i++){
    if (lines[i].length < 73)
    {
      continue;
    }
    for(var j = 55; j < 73; j++)
    {
      if(lines[i][j] == " ")
      {
        lines[i] = splitValue(lines[i], j);
        if(lines[i].length >  140)
        {
          for(var j = 120; j < 140; j++)
          {
            if(lines[i][j] == " ")
            {
              lines[i] = splitValue(lines[i], j);
              break;
            }
          }

        }
        else
        {
          break;
        }

        //TODO: maybe recursive
      }
      else{
        if(j >= 72){
        for(var l = 73; l < lines[i].length; l++)
          {
            if(lines[i][l] == " ")
            {
              lines[i] = splitValue(lines[i], l);
              break;
            }
          }
        }
      }


    }


}
return (lines.toString());


}
```

Listing 3.3: function parse lines

```
function remove_commas(string_line){

for(var n = 0; n < string_line.length; n++)
{
  if(string_line[n] == "," && string_line[n - 1] == ">" && string_line[n + 1] == "<"
      )
  {
    string_line = replaceAt(string_line, n, "\n");
  }
}

console.log(string_line);
return string_line;
}
```

Listing 3.4: Function remove commas

Chapter 4

Generating an .EXE File from Gulp

Pkg is a command line interface which provides a possibility to pack a NodeJS project into a single executable file which then can be run on any device without having NodeJS previously installed[Klopov 2020].

Most popular use cases for this tool:

- Fast way to make executable for other platforms
- Create temporary version of your application without using additional resources
- Spares you the trouble of using npm to install dependencies
- Pack everything up in one runnable file
- Check and test application against newer versions of NodeJS without the need to install them first

To install it via npm, by running:

```
    npm install -g pkg
```

List of arguments:

- -v, –version Check pkg version
- -c, –config package.json or any json file with top-level config
- -t, –targets followed by target machine(node4-linux, node6-win, node6-mac)
- -h, –help outputs usage information
- -o, –output output file name or template for several files
- -d, –debug show more info during packaging process
- –public speed up and disclose the sources of top-level project
- –out-path specify where to save the executable

We were not successful to generate the executable file with the arguments for our project.

4.1 JavaScript Instead

Another idea would be to use JavaScript as a replacement instead, by extracting the file separately and using the pkg tool with it.

Chapter 5

AChart Creator Changes

We have made improvements on AChart Creator over the past six weeks. We have also fixed and found some bugs, as well as adding some Quality-of-Life changes.

5.1 Folder Structure

Initially, if the folder structure was not correctly set up, the transpiled JavaScript and output SVG files were not able to be saved. It was required to have a folder called `build` which must include two separate folders called `js` and `svg`. If this folder structure was not like this, no file would be written. A code snippet of how this was fixed in the `gulpfile.js` file can be seen in Listing 5.1.

```
if(!fs.existsSync(paths.build)) {
    fs.mkdirSync(paths.build);
    console.log('folder created: ', paths.build)
}
```

Listing 5.1: Creating Proper File Structure.

5.2 SVG Title, Description, and Header

Previously, to add the title, description, and the header (the visible title) of a chart in the SVG code, it had to be done by hardcoding it into the TypeScript code, as seen in Listing 5.2. The `.text` attributes in Listing 5.2 are then swapped by variables whose values are obtained from the command line.

An example of the new command line arguments to include title and description can be seen in Listing 5.3. The `gulpfile.js` file sanitizes these arguments and passes the proper strings to serve as the title and description of the SVG file to the TypeScript recipe. A code snippet of the resulting SVG can be seen in Listing 5.4

5.3 Embedded Table

To further add Quality-of-Life changes, we embedded the data found in the input CSV file as a JSON string. This allows the SVG to keep the content/data that was used to create it. If the user were to need this, it would be very accessible. A snippet of the SVG code which includes this pseudo-table can be seen in Listing 5.5.

```
rootNode.append("title")
    .attr("id", "title")
    .attr("tabindex", "0")
    .attr("role", "heading")
    .text("Most Popular Fruits");

rootNode.append("desc")
    .attr("id", "desc")
    .text("Most Popular Fruits in 2019 given in percentage");
```

Listing 5.2: Title and Description being Hard-Coded.

```
gulp acreate --file pie.ts --dataset fruit-pie.csv --title "New_Title"
    --desc "New_Desc"
```

Listing 5.3: Example running AChart Creator with Title and Description Arguments

```
<svg viewBox="0 0 400 400" version="1.1" xmlns="http://www.w3.org/2000/svg"
 xmlns:xlink="http://www.w3.org/1999/xlink" role="graphics-document">
<g id="PieRoot" role="chart" aria-charttype="pie" tabindex="0"
 aria-labelledby="title desc">
<title id="title">New_Title</title>
<desc id="desc">New_Desc</desc>
<text role="heading" tabindex="0" x="200" y="25" text-anchor="middle">New_Title</
    text>
<g id="pie-chart" transform="translate(200,200)" role="dataset">
```

Listing 5.4: Title and Description SVG Output.

```
<svg viewBox="0 0 400 400" version="1.1" xmlns="http://www.w3.org/2000/svg"
 xmlns:xlink="http://www.w3.org/1999/xlink" role="graphics-document">
<p>[{"fruits":"Apples","value":9},{"fruits":"Bananas","value":20},{"fruits":"
    Grapefruits","value":30},{"fruits":"Lemons","value":8},{"fruits":"Oranges","
    value":12}]</p>
<g id="PieRoot" role="chart" aria-charttype="pie" tabindex="0"
 aria-labelledby="title desc">
<title id="title">New_Title</title>
<desc id="desc">New_Desc</desc>
<text role="heading" tabindex="0" x="200" y="25" text-anchor="middle">New_Title</
    text>
<g id="pie-chart" transform="translate(200,200)" role="dataset">
```

Listing 5.5: SVG Embedded Pseudo-Table Example.

Chapter 6

New Recipes

6.1 BoxPlot

D3.js Boxplot is box and whisker plot with axes. The core of this Boxplot is Mike Bostock's implementation. The main difference is that Mike uses individual svg elements and in this implementation all box plots are rendered with. It makes possibility for every user to easily add axes [Grubert 2019].

The other differences are:

- transitions are not used
- with labels variables the visibility of boxplots can be switched
- CSV files are supported

We didn't adapt this Boxplot for our project. Given ts code can not be adapted for our case. Debugging into ts doesn't offers us some help.

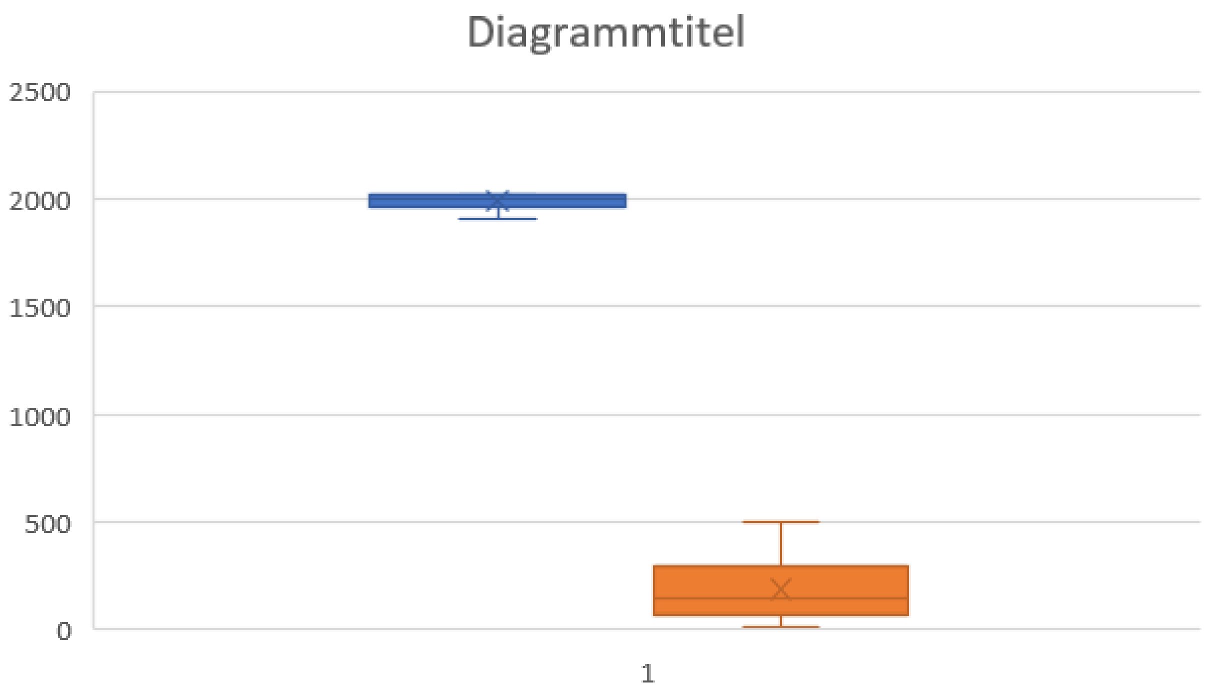

Figure 6.1: D3.js Boxplot with Axes and Labels [Image taken by the author.]

Chapter 7

Conclusion

Using Unminify is not possible for AChart Creator as it does not integrate with it, given that it is web-based and AChart Creator must be run locally. JS Beautifier can be integrated properly. However, as it recognize the `"<"` and `">"` as arithmetic and logical symbols, it adds spaces before and after these characters. Effectively making HTML and SVG code useless. Scour is a Python library - integrating it is impossible, although it does exactly what we want it to do. SVGO optimizes our SVG code and leaves it integrally intanct. However, it does not add newlines to make our SVG human-readable. SVGO `gulp` adds the newlines but removes attributes, messing with the file's integrity.

We then made our own parser, as seen in Chapter 3. It makes it human-readable but indentation based on hierarchy is still missing.

Building our own `.exe` from the Node project proved difficult do to the argument-based nature of our tool. Our best bet to running our recipes locally without having to run Node is to run the output JavaScript files found in the `build/js` folder. Taking a quick look at which arguments to add and which are possible is as simple as looking at the source code or at Listing 5.3 and removing the arguments that start with a `"-"`.

Furthermore, we have included the possibility of adding a title, description, and heading to our SVG files (as seen also in 5.3). This addition was also made compatible with Describler and AChart Reader.

At last, the stepping stone of our BoxPlot recipe will be invaluable to whomever decides to undertake the project of AChart Creator's further improvement.

Bibliography

Belevich, Kir [2020]. *SVGO*. `https://github.com/svg/svgo/` (cited on page 5).

Einar Lielmanis, Liam Newman [2016]. *JSBeautifier*. 2016. `https://github.com/beautify-web/js-beautify` (cited on page 3).

Grubert, Jens [2019]. *D3.js Boxplot*. `http://bl.ocks.org/jensgrubert/7789216` (cited on page 17).

Klopov, Igor [2020]. *PKG*. `https://github.com/zeit/pkg/` (cited on page 13).

MDN contributors [2020]. *ARIA*. 12 Jan 2020. `https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA` (cited on page 1).

Media4x [2020]. *Unminify*. `https://unminify.com` (cited on page 3).

Schepers, Doug [2020]. *Describler*. 31 Jan 2020. `http://describler.com/` (cited on page 2).

Tobias Oberstein, Eduard Braun [2020]. *Scour*. `https://github.com/scour-project/scour` (cited on page 4).