GLKN Water Quality Visualizer ReadMe

Hallie Arno

## Introduction

The GLKN water quality visualizer is made in Quarto, a posit product similar to R markdown, using R, HTML, YAML, and Observble JS to wrangle data, create interactive plots, and display outputs. A Quarto file looks similar to an r markdwon file, where each cell is a seperate script. Quarto can support cells in multiple languages, including Observable JS. Observable JS (OJS) is similar to javascript and allows the document to be interactive. The way it works is by re-running any {ojs} cells any time an input is changed. Quarto renders each of these languages into an html document that can be shared on the web. This HTML document does not run R or OJS, these components are rendered into HTML and regular javascript to work like most web pages (this means that data is only updated each time the document is re-rendered, not when it is reloaded in a browser). For more information about integrating Quarto and Observable JS, start here: [Quarto - Observable JS](https://quarto.org/docs/interactive/ojs/). [Infoworld](https://www.infoworld.com/article/3674789/a-beginners-guide-to-using-observable-javascript-r-and-python-with-quarto.html) also has a helpful tutorial for getting started.

## How to run the visualizer

To run the visualizer, download the .qmd file and open in R studio (make sure you have Quarto downloaded, see below). Then press “render” in the upper left area in R studio.



Alternatively, the rendered visualizer can be accessed using the html file.

## **Quarto**

Quarto can be downloaded from [here](https://quarto.org/docs/get-started/), or by installing the quarto package. When creating a quarto document, there are many options for the output. The GLKN WQV is an html output. All quarto documents begin with YAML heading to specify format. Some output formatting can be done in the YAML (such as specifying sidebar width or number of columns on the page), though I found it easier to use a bootstrap grid and a separate scss file, and set the YAML page-layout to “custom”. In the YAML, I also set “echo” to “false” to not display the code. See [Quarto’s guide](https://quarto.org/docs/guide/) for more information.

For the GLKN WQV, I used a sidebar and tabset panel layout. These are easy to create using bootsrap commands around the blocks that it should be applied to, or after “//|” at the top of the cell. For an example, see [Observable JS] for an input panel.

## **Getting the data (R)**

All of the data used in the visualizer is publicly available water quality monitoring data collected by GLKN from 2007-2021. This data can be accessed using the [EPA’s Water Quality Portal](https://www.epa.gov/waterdata/water-quality-data). Because this data is already publicly available and non-sensitive, it can be shared publicly.

Data is retrieved using R and the [dataRetrieval](https://owi.usgs.gov/R/training-curriculum/usgs-packages/dataRetrieval-readWQP/index.html) [@dataRetrieval] package. Site names are provided as a csv downloaded from the EPA Water Quality Portal. All NPS site names begin with “11NPSWRD\_WQX-”.  Other packages used to wrangle data are lubridate [@lubridate], data.table [@data.table], and tidyverse [@tidyverse]. Once the data is loaded and wrangled in a dataframe, the next step is to transpose the data to be readable by Observable JS, using ojs\_define:

library(tidyverse)

my\_r\_data\_frame <- read.csv("station.csv")  
ojs\_define(my\_ojs\_data = transpose(my\_r\_data\_frame))

Note that the tidyverse package must be loaded for this to work.

## **Observable JS**

Once the data are transposed, R cannot be used anymore. This is for two reasons: R cannot interperet the new data structure, and R cells are not re-run when inputs change. This means that R cells load on startup, but cannot be interactive based on OJS inputs. In contrast, ojs cells are re-run simultaneously when inputs change, similar to a spreadsheet. This means that the order of cells in a document does not matter. Because of this, I have structured the GLKN WQV in the order of:

1. HTML header (NPS banner, image, title)
2. Getting data from R
3. Functions to create outputs
4. Sidebar input selections
5. Outputs (calling functions from section 3)

This is similar to how a shiny app would be structured, with a server/ UI in different sections.

Note that OJS is very similar to javascript, but has a few differences, such as how variables are defined, outlined [here](https://observablehq.com/@observablehq/observable-javascript).

Inputs are created in Observable JS, using [viewof syntax](https://observablehq.com/@observablehq/a-brief-introduction-to-viewof), such as:

//|panel: input  
viewof selected\_park = Inputs.select(  
 ["SACN", "ISRO", "VOYA", "SLBE", "INDU", "APIS", "PIRO"],   
 {  
 value: ["ISRO"],   
 multiple: true,   
 label: "Park:"  
 }  
)

This creates a new variable, park, which is an array of selected options. This array can then be used to filter data and display only selected parks on a graph or in a table. Filtering in javascript is fairly easy:

//note that waterFilter is a dummy variable since this is a function  
//where my\_ojs\_data is being supplied as the input  
filtered\_dataset = my\_ojs\_data.filter(function(waterFilter) {   
 return selected\_park.includes(waterFilter.Park);  
})

In OJS, a variable (or plot) can be displayed by simply typing it into the cell:

filtered\_dataset

\*Note that defining variables is different in javascript and OJS and can change depending on whether the variable is wrapped in curly brackets or not.

Plots must be created in OJS in order to be responsive to OJS inputs. Any JS plotting library can be used. I chose to work in [Plotly](https://plotly.com/javascript/). Part of this is because plots look nice out of the box, allow for high levels of customization, and have fairly readable syntax compared to libraries like d3 and Vega-Lite. OJS has their own plotting library (Observable Plot) that is simple to use, but is more difficult to customize.

## **Plotly**

Plotly is an open source graphing library based on d3. It has built in tooltips, zooming, and highlighting. However, one limitation is that it is not able to facet plots like ggplot is (I was unable to find a javascript graphing library that could do this, though they may be out there). This makes it difficult to have, for example, multiple sites on the same plot marked with different colors. Because of this, each “trace” – a set of filtered data, such as a particular site– must be created seperately, then added to a plot. This also means that there is no built in way create a grid and resize each plot based on the number of plots in the grid.

For this draft of the visualizer, I used a loop to filter data for each selected site and variable to create individual traces. These traces were then added to a different plot for each variable. This was all done inside a function. Instead of trying to dynamically create a grid, I simply called the function using a each variable as the function input in a new cell, so that Quarto would handle putting them into a grid using bootsrap. This means that there are functions calling data that does not exist (i.e. calling the make\_plot function on the fourth selected variable when only three variables are selected). This returns “undefined.” To avoid displaying “undefined,” I used the scss file to make the font size zero for the output of these cells. I avoided using subplots because they cannot have their own titles (as of July 2023).

There is almost certainly a better way to do this, and as I learn more I hope to streamline this process. Despite the innefficiencies, the plots render extremely quickly. However, on some screens they may extend past the edge of the screen.

## **Leaflet**

The map on the first tabset is created using [leaflet](https://leafletjs.com/). I used clipart to create [custom markers](https://leafletjs.com/examples/custom-icons/) that can be toggled based on sidebar inputs or when clicked. This is accomplished by keeping a litst of clicked results and checking if the clicked or selected input is in that list, and switching markers accordingly. The results list is pushed to (synced with) the site input so that sites selected on the map can be used to filter data on the plots tab. Resources on [GitHub](https://github.com/kbvernon/hndsr-watersheds) and [StackOverflow](https://stackoverflow.com/questions/25683871/assign-id-to-marker-in-leaflet) helped to accomplish that feature.

## **scss**

An scss file is used to style the rendered html page, most importantly the black NPS banner at the top (which is currently unofficial and uses the wrong font), the header image, and background colors. For each item to style, a class is added in html, then that class is referenced in the scss file. More information can be found on [Quarto’s website](https://quarto.org/docs/output-formats/html-themes.html). Note that functionality is not dependent on this, and most styles can be specified directly in the Quarto document.

## Contact

For questions, please contact arno.hallie@gmail.com.