Combining Observable JS and Shiny in a Single Quarto Document

Ronald G. Thomas

2025-12-01

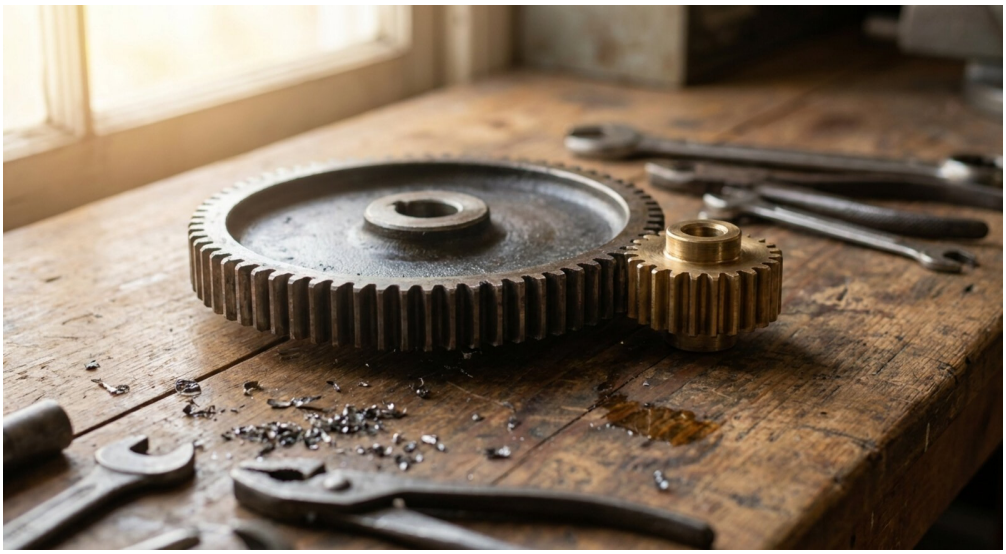

Figure 1: Two different gears, one large and one small, meshed together and turning: two different systems working together

Combining two reactive frameworks in one document is harder than it looks.

Introduction

I did not really know how difficult it would be to combine Observable JS and Shiny in a single Quarto document until I tried to create a side-by-side comparison. Like many statisticians, I assumed that since Quarto supports both frameworks, mixing them would be straightforward. Thirty minutes of setup turned into several hours of debugging.

The core difficulty is that Observable JS and Shiny use fundamentally different execution models. Observable runs entirely in the browser; Shiny requires a server process. When `server: shiny` is added to a Quarto document, the rendering context changes in ways that break standard Observable data-loading patterns.

We document here the journey from naive optimism to a working solution. Every data-loading approach tried failed: `FileAttachment`, `ojs_define`, and local `d3.csv`. Fetching from a public URL was ultimately found to bypass the conflict entirely.

Motivations

- I wanted to compare client-side and server-side reactive frameworks in a single document for a workshop demonstration.
- Colleagues frequently ask which framework to use for interactive visualizations, and a side-by-side comparison seemed more persuasive than a verbal explanation.
- The Quarto documentation implies that OJS and Shiny can coexist, but provides little guidance on the practical difficulties of combining them.
- I was curious whether the multi-language support in Quarto truly extends to mixing reactive systems, or whether it is limited to sequential code blocks.
- I needed to document the gotchas for my own future reference and for others attempting the same combination.

Objectives

1. Create identical interactive visualizations using both Observable JS and Shiny, displayed side-by-side in a single Quarto document.
2. Document the data-loading approaches that fail (`FileAttachment`, `ojs_define`, `d3.csv`) and explain why they fail in the Shiny context.
3. Provide a working solution using `fetch()` from a public URL that bypasses the server-client conflict.
4. Summarize the key syntax differences between OJS and Shiny for statisticians evaluating both frameworks.

This learning process is documented here. Errors spotted or better approaches are always welcome.

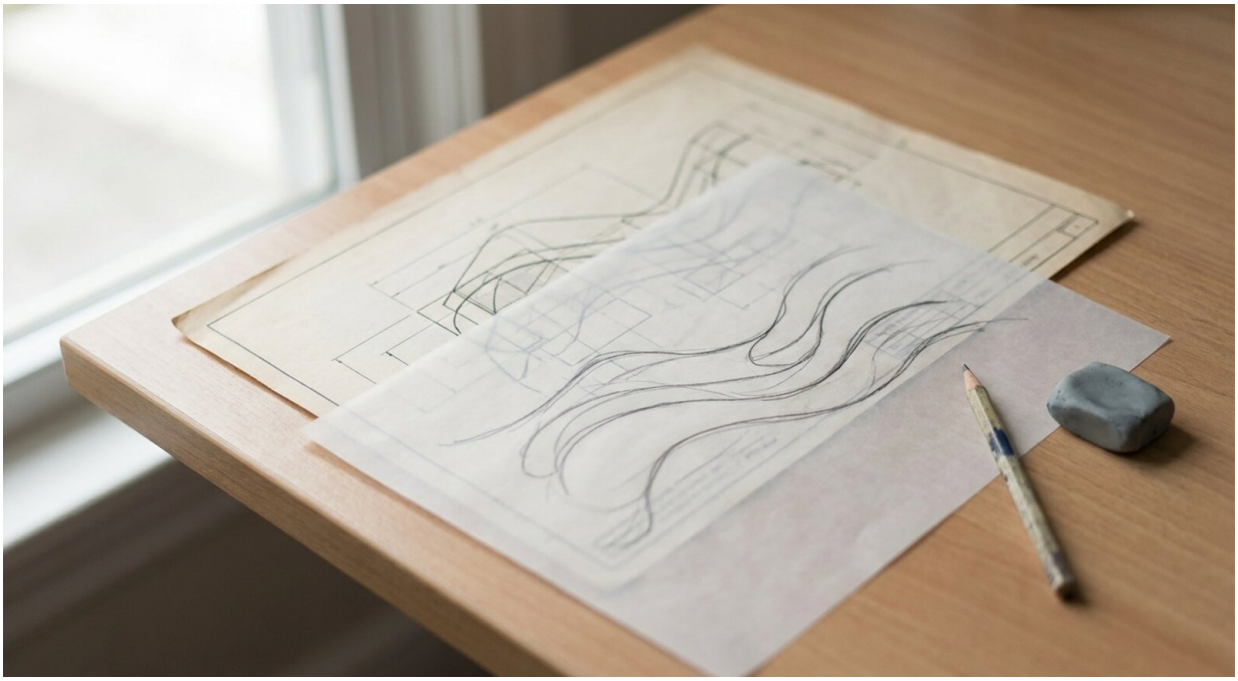

Figure 2: A sheet of tracing paper laid over a printed diagram, revealing a second, complementary drawing sketched on top: one document layer combining two different tools.

Understanding the architectural differences between client-side and server-side reactivity.

Prerequisites and Setup

To follow along with this post, one will need:

- Quarto (1.3 or later)
- R with ggplot2 and dplyr packages
- A web browser (Chrome or Firefox recommended)

The Observable JS code runs in the browser and requires no additional installation. The Shiny code requires a running R server, which Quarto manages automatically when `server: shiny` is set.

The Palmer Penguins dataset is used for both frameworks. For the OJS side, data is fetched from a public URL. For the Shiny side, data is read from a local CSV file.

What is Client-Side versus Server-Side Reactivity?

Client-side reactivity means that all computation happens in the browser. Observable JS exemplifies this approach: data is loaded into the browser, and user interactions trigger JavaScript functions that filter, transform, and visualize data without contacting a server. The advantage is speed; the limitation is that the browser must hold all data in memory.

Server-side reactivity means that user interactions trigger requests to a server process, which performs computation and returns results to the browser. Shiny exemplifies this approach: R runs on the server, and the browser displays rendered output. The advantage is access to the full R ecosystem and arbitrarily large datasets; the limitation is network latency and server resource requirements.

When both are combined in a single Quarto document, one is running a Shiny server that also serves Observable JS. This dual context is where the data-loading conflicts arise.

Getting Started

The Side-by-Side Comparison

Below are two identical visualizations: one built with Observable JS (client-side) and one with Shiny (server-side). Both filter the Palmer Penguins dataset by bill length and display a histogram of body mass.

Observable JS

```
//| echo: true

viewof bill_min_ojs = Inputs.range(
  [32, 50],
  {value: 35, step: 1,
   label: "Min bill length (mm):"}
)

penguins_raw = {
  const response = await fetch(
    "https://raw.githubusercontent.com/" +
    "allisonhorst/palmerpenguins/" +
    "main/inst/extdata/penguins.csv"
  );
  const text = await response.text();
  return d3.csvParse(text, d3.autoType);
}

data_ojs = penguins_raw.filter(
  d => d.bill_length_mm != null &&
       d.body_mass_g != null
)

filtered_ojs = data_ojs.filter(
  d => d.bill_length_mm > bill_min_ojs
)
```

```
Plot.plot({
  height: 250,
  marks: [
    Plot.rectY(
      filtered_ojs,
      Plot.binX(
        {y: "count"},
        {x: "body_mass_g", fill: "species"}
      )
    ),
    Plot.ruleY([0])
  ]
})
```

Count: ${filtered_ojs.length} penguins

Shiny

```
sliderInput(
  inputId = "bill_min_shiny",
  label = "Min bill length (mm):",
  min = 32, max = 50, value = 35, step = 1
)

plotOutput("hist_shiny", height = "250px")
textOutput("count_shiny")
```

```
library(ggplot2)
library(dplyr)

data_shiny <- read.csv(
  "data/raw_data/palmer-penguins.csv"
)

filtered_shiny <- reactive({
  data_shiny |>
    filter(
      bill_length_mm > input$bill_min_shiny
    ) |>
    filter(
      !is.na(body_mass_g),
      !is.na(species)
    )
})
```

```r
output$hist_shiny <- renderPlot({
  ggplot(
    filtered_shiny(),
    aes(x = body_mass_g, fill = species)
  ) +
    geom_histogram(bins = 20) +
    theme_minimal() +
    labs(x = "Body Mass (g)", y = "Count")
})

output$count_shiny <- renderText({
  paste(
    "Count:",
    nrow(filtered_shiny()),
    "penguins"
  )
})
```

Key Syntax Differences

Aspect	Observable JS	Shiny
Input	`viewof x = Inputs.range()`	`sliderInput("x", ...)`
Access value	Use `x` directly	`input$x`
Data loading	`fetch()`	`read.csv()`
Filtering	`data.filter()`	`reactive({})`
Plot	`Plot.plot({...})`	`renderPlot({...})`
Reactivity	Implicit	Explicit wrappers

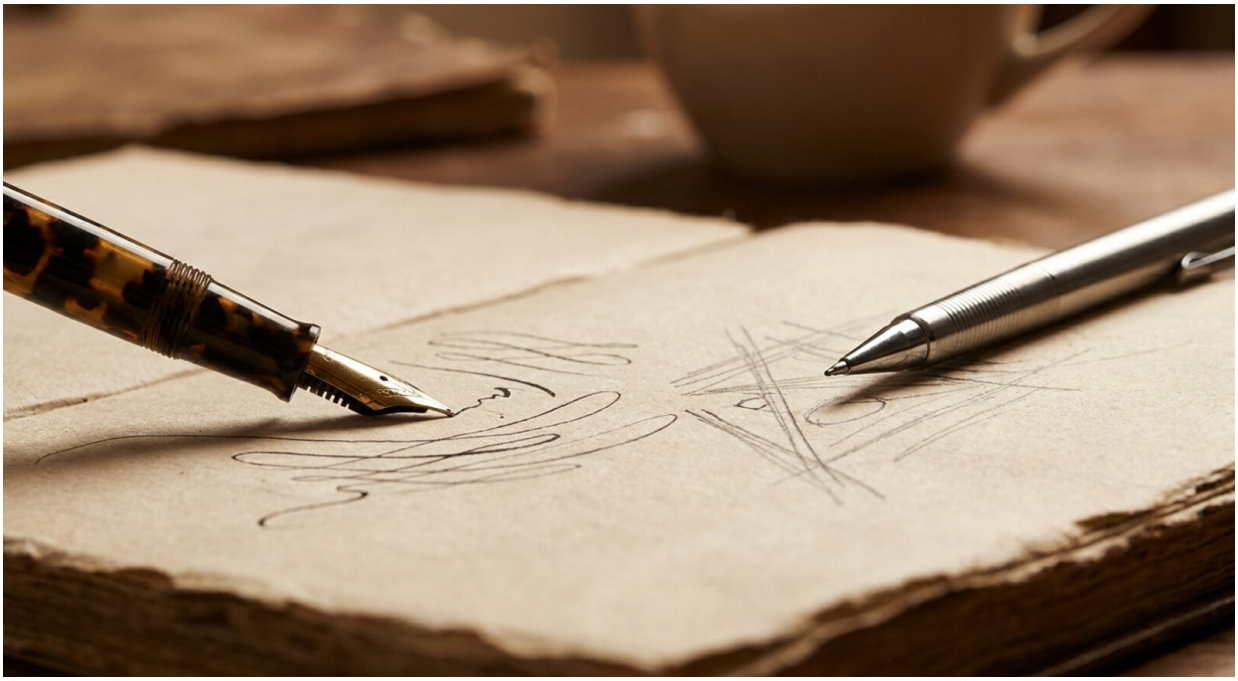

Figure 3: Two pens, one fountain pen and one mechanical pencil, writing on the same open page from opposite corners: two languages contributing to one document

Debugging reactive framework conflicts requires patience and systematic elimination.

The Challenges

Getting the side-by-side comparison working was surprisingly difficult. Each approach I tried revealed a different aspect of the server-client conflict.

Challenge 1: FileAttachment Does Not Work

My first attempt used Observable's standard data-loading mechanism:

```
// THIS DOES NOT WORK IN SHINY DOCUMENTS
data = FileAttachment(
  "palmer-penguins.csv"
).csv({typed: true})
```

Result: `OJS Error: Unable to load file`

Why: When `server: shiny` is added, the document runs as a Shiny application. The Shiny server does not know about Observable's file attachment system. The file resolution mechanism that works in static Quarto documents is unavailable in the Shiny context.

Challenge 2: ojs_define Fails Too

The Quarto documentation mentions `ojs_define()` for passing data from R to OJS:

```
#| context: server
data <- read.csv("penguins.csv")
ojs_define(my_data = data)
```

Result: `Error: Unexpected data.frame object... Did you forget to use a render function?`

Why: In the Shiny context, `ojs_define()` expects reactive expressions, not static data frames. The behavior differs from what the documentation shows for regular (non-Shiny) Quarto documents.

Challenge 3: Local d3.csv Returns 404

I tried D3's CSV loader as an alternative:

```
// THIS ALSO DOES NOT WORK
data = d3.csv(
  "palmer-penguins.csv", d3.autoType
)
```

Result: `OJS Error: 404 Not Found`

Why: The Shiny server does not serve local static files in the way that Observable expects. The file is present on disk but not accessible through the URL that `d3.csv` constructs.

The Solution: Fetch from a Public URL

The only reliable approach I found is fetching data from a public URL:

```
penguins_raw = {
  const response = await fetch(
    "https://raw.githubusercontent.com/" +
    "allisonhorst/palmerpenguins/" +
    "main/inst/extdata/penguins.csv"
  );
  const text = await response.text();
  return d3.csvParse(text, d3.autoType);
}
```

This bypasses the Shiny server entirely by making an HTTP request to an external source. The browser handles the fetch independently of the server process.

Dos and Don'ts

Do:

1. Use `fetch()` with public URLs for OJS data in Shiny documents.
2. Keep OJS and Shiny data loading completely separate.
3. Test incrementally: get Shiny working first, then add OJS.
4. Hard-refresh the browser (Cmd+Shift+R) when debugging.
5. Restart the preview server after major changes.

Do not:

1. Use `FileAttachment()` in Shiny documents.
2. Use `ojs_define()` with static data in the Shiny context.
3. Assume `d3.csv("local.csv")` will work.
4. Trust displayed code during debugging; browser caching causes confusion.
5. Mix OJS and Shiny logic in the same code block.

Additional Gotchas

Browser caching: The browser often shows old code during debugging. The displayed code might show `FileAttachment(...)` even though the file now uses `fetch()`.

Solution: Kill the server, delete generated files, restart, and hard-refresh:

```
rm -rf yourfile_files yourfile.html
quarto preview yourfile.qmd
```

Port changes: Every restart may assign a different port (5155, 6532, 7665). Check the terminal output for the correct URL.

Misleading error messages: The error "Did you forget to use a render function?" does not indicate that `renderSomething()` is needed. It means `ojs_define()` does not work with static data in the Shiny context.

Things to Watch Out For

1. **FileAttachment silently fails.** There is no helpful error message explaining that the Shiny context disables Observable's file system. Only a generic load error is shown.
2. **Browser caching masks fixes.** After changing from `FileAttachment` to `fetch`, the browser may still display the old code and the old error. Hard-refresh every time.
3. **Port numbers change on restart.** Do not bookmark a specific port; check the terminal output after each `quarto preview`.
4. **OJS and Shiny cannot share data.** There is no reliable way to pass data between the two contexts. Keep them independent.
5. **The Quarto documentation does not cover this edge case well.** Expect to rely on experimentation rather than official guidance.

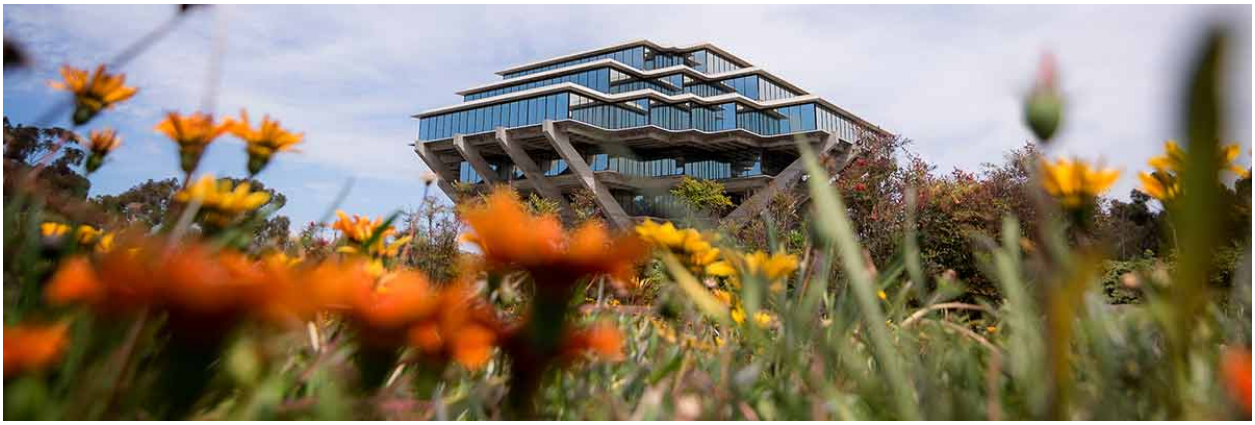

Figure 4: A whiteboard with architectural diagrams comparing client-server models

Understanding the architectural boundary between client-side and server-side is the key insight.

Lessons Learned

Conceptual Understanding

- Observable JS and Shiny use fundamentally different execution models: client-side versus server-side. Combining them means running both simultaneously.
- The Shiny server context overrides Observable's file resolution system. File loading that works in static Quarto documents breaks in Shiny documents.
- Data isolation between the two frameworks is not a limitation to work around but a constraint to accept. Attempting to share data between contexts leads to fragile, unreliable solutions.
- The `fetch()` API provides a clean escape hatch because it operates at the HTTP level, independent of both frameworks' internal resolution mechanisms.

Technical Skills

- Using `fetch()` with `d3.csvParse()` for OJS data loading in Shiny documents became the reliable pattern.
- Keeping OJS and Shiny code in strictly separate blocks prevents context contamination.
- Hard-refreshing the browser (Cmd+Shift+R) and deleting generated files became essential debugging habits.
- Understanding the `#| context: server` chunk option clarified how Shiny server code differs from UI code in Quarto.

Gotchas and Pitfalls

- `FileAttachment()` does not work in Shiny documents, and the error message does not explain why.
- `ojs_define()` behaves differently depending on whether the document uses `server: shiny` or not.

- Browser caching during debugging causes the most time-consuming confusion.
- Error messages from OJS in the Shiny context are often misleading; they reference the wrong cause.

Limitations

- This approach requires hosting data at a public URL for the OJS side, which may not be acceptable for sensitive or proprietary datasets.
- There is no reliable data sharing between OJS and Shiny contexts; each framework loads and processes data independently.
- Debugging is significantly harder than in single-framework documents because two reactive systems interact through the browser.
- The Quarto documentation does not cover this combination in detail, making experimentation the primary learning method.
- Performance may suffer because the browser runs OJS reactivity while simultaneously communicating with the Shiny server.
- The `fetch()` workaround depends on network availability; offline use is not supported for the OJS side.

Opportunities for Improvement

1. Test `ojs_define()` with `reactive()` wrappers in the Shiny server context to determine whether reactive expressions can bridge the gap.
2. Create a helper function or Quarto extension that transparently handles data loading for both frameworks.
3. Explore Shiny's resource-serving configuration to determine whether local files can be made available to OJS through custom routes.
4. Document a clean workflow for teams that need both frameworks in production, including testing strategies and debugging protocols.
5. Investigate whether Quarto's OJS runtime could be modified to respect Shiny's static file serving mechanism.
6. Build a minimal reproducible example repository that demonstrates the working pattern for others to fork.

Wrapping Up

Combining Observable JS and Shiny in a single Quarto document is possible but not straightforward. The fundamental issue is that the Shiny server context disables Observable's standard file-loading mechanisms, and the two frameworks cannot reliably share data.

What this exploration revealed is that the boundary between client-side and server-side execution is not merely architectural: it is a hard constraint that determines which approaches work and which fail silently. Once it is accepted that the two frameworks must remain independent, the `fetch()` solution becomes obvious.

For educational purposes (comparing both technologies side-by-side) this approach is valuable once the setup hurdles are cleared. For production applications, choosing one framework rather than combining both is the more prudent path.

In conclusion, four points merit emphasis. First, `FileAttachment()`, `ojs_define()`, and `d3.csv()` all fail in the Shiny context because the Shiny server disables Observable's standard file-loading mechanisms. Second, fetching from a public URL via `fetch()` is the reliable workaround because it operates at the HTTP level, independent of both frameworks' internal resolution mechanisms. Third, OJS and Shiny data loading must be kept completely independent; there is no reliable bridge between the two contexts. Fourth, a hard browser refresh after every change is essential during debugging, as caching can mask fixes and waste substantial time.

See Also

Related posts:

- Prototyping a Shiny App with ChatGPT: Building a Shiny app iteratively with AI assistance

Key resources:

- Quarto OJS Documentation: Official Observable JS integration guide
- Quarto Shiny Documentation: Official Shiny integration guide
- Observable Plot: The plotting library used in the OJS examples
- Palmer Penguins Dataset: The dataset used in both frameworks
- D3.js: Data-Driven Documents: The JavaScript library underlying Observable Plot

Reproducibility

This post contains live Observable JS code that runs in the browser and Shiny server code that is displayed but not evaluated (set to `eval: false` for static rendering). To run the full interactive version:

```
cd ~/prj/qblog/posts/34-shinyvsobservable/
cd shinyvsobservable
quarto preview analysis/report/index.qmd
```

The OJS code fetches data from a public GitHub URL and requires network access. The Shiny code reads from a local CSV file at `data/raw_data/palmer-penguins.csv`.

Project files:

```
shinyvsobservable/
  analysis/report/index.qmd  (this post)
  data/raw_data/              (penguins CSV)
  analysis/media/images/      (hero, ambiance)
```

Let's Connect

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** rgtlab.org/contact

I would enjoy hearing from readers who:

- Spot an error or a better approach to any of the code in this post.
- Have suggestions for topics to cover.
- Want to discuss R programming, data science, or reproducible research.
- Have questions about anything in this tutorial.
- Simply want to say hello and connect.

Related posts in this cluster

This post is part of the *Shiny and Interactive Visualization* series. Recommended reading order:

1. **Post 90: Combining Observable JS and Shiny in a Quarto Document** (this post)
2. Post 91: Prototyping a Shiny App with ChatGPT