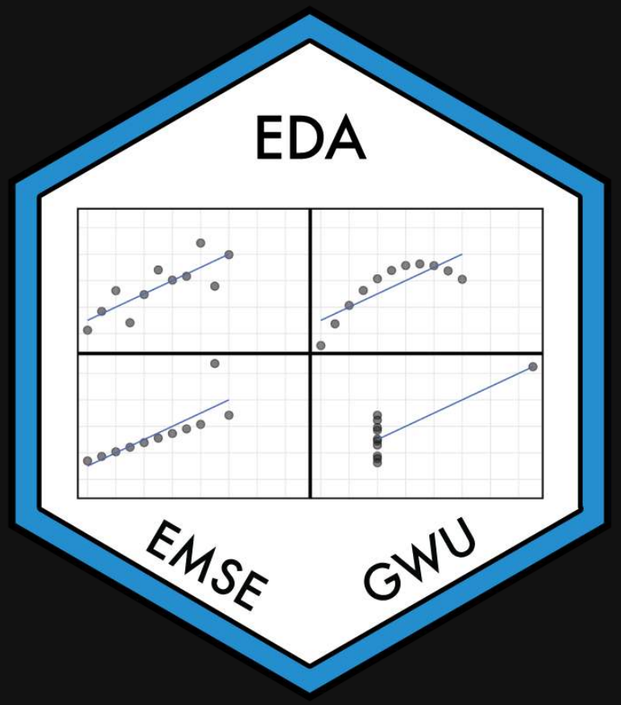


Interactivity

EMSE 4572/6572: Exploratory Data Analysis

John Paul Helveston

November 18, 2026

Week 13: *Interactivity*

1. JavaScript graphics: packages
2. JavaScript graphics: D3 + Claude
3. Shiny apps
4. Shiny extras

Week 13: *Interactivity*

1. JavaScript graphics: packages
2. JavaScript graphics: D3 + Claude
3. Shiny apps
4. Shiny extras

Why build an interactive chart?

The reader can…

- **Hover** for exact values
- **Filter** to the part they care about
- **Zoom** into a crowded region
- **Play** through time
- **Scroll** through a story, one step at a time

The cost

- Doesn’t work on paper or in a PDF
- Not easily sharable (e.g., social media)
- More to build, and more to *check*
- Hides information behind a mouse movement

Two ways to make JavaScript-based graphics

Packages that wrap JavaScript

`plotly`, `reactable`

- Hide the JavaScript behind R code
- Short, readable code you could write yourself
- Limited to what the package supports

Custom graphics with D3

`D3` + agents

- Build **anything** you can describe
- Don’t write it by hand - prompt and iterate
- Verify the result against a static chart and the data

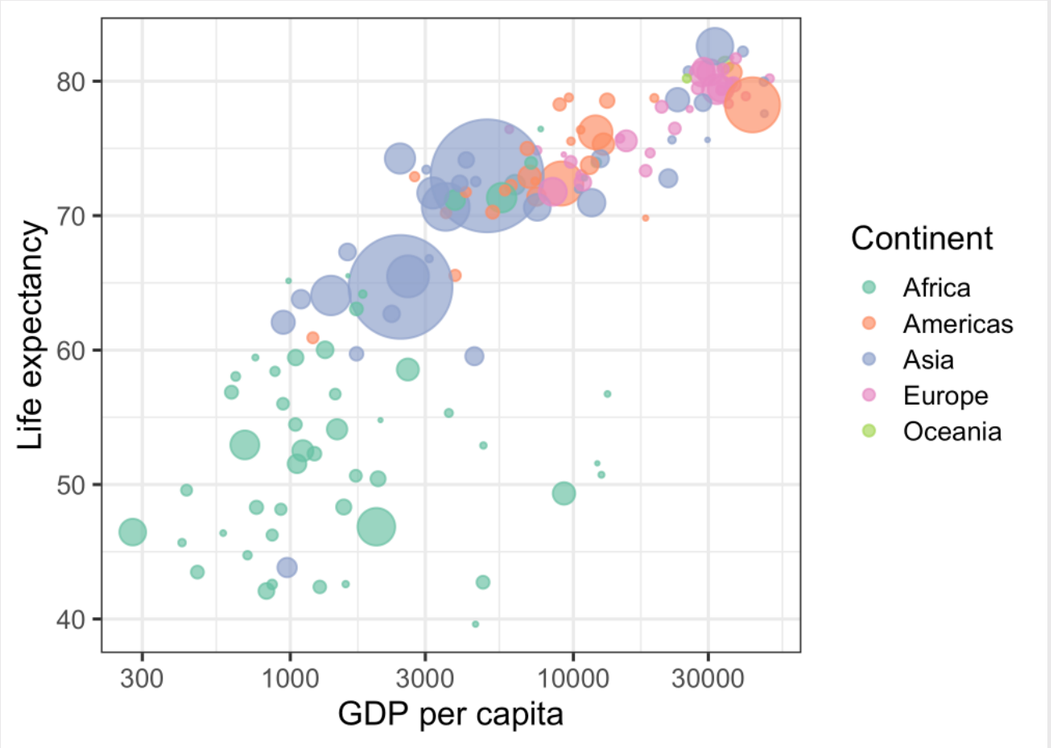
Plotly uses JavaScript to create interactive charts

But you don’t have to know JavaScript to use it! 🎉

Turn any ggplot into an interactive chart with `ggplotly()`

```
plot <- gapminder %>%
  filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp,
             size = pop, color = continent,
             label = country)) +
  geom_point(alpha = 0.7) +
  scale_color_brewer(palette = 'Set2') +
  scale_size_area(
    guide = FALSE, max_size = 25) +
  scale_x_log10() +
  theme_bw(base_size = 16) +
  labs(x = 'GDP per capita',
       y = 'Life expectancy',
       color = 'Continent')

plot
```



Turn any ggplot into an interactive chart with `ggplotly()`

```
1 ggplotly(plot)
```

Control what the tooltip shows with the `tooltip` argument

```
ggplotly(
  plot,
  tooltip = c("country", "pop")
)
```

`ggplotly()` is a *translation*

Translations lose things

- **Tooltips** often show raw aesthetic names (`x`, `y`, `colour`) and unrounded numbers
- **Subtitles, captions, and annotations** can move or vanish
- **Fonts and themes** don’t always survive
- **Legends** can be duplicated or oddly placed

Alternative: just use plotly directly

Save as html page

```
htmlwidgets::saveWidget(
  ggplotly(plot),
  file = file.path('figs', 'gapminder.html')
)
```

Insert using iframe (same thing we'll do with D3!)

```
htmltools::tags$iframe(
  src         = file.path('figs', 'gapminder.html'),
  width       = "100%",
  height      = "400",
  scrolling   = "no",
  seamless    = "seamless",
  frameBorder = "0"
)
```

Interactive *tables*: `reactable::reactable()`

```
1 library(reactable)
2
3 gapminder %>%
4   reactable()
```

`reactable()` has some nice options!

```
 1  library(reactable)
 2
 3  gapminder %>%
 4    reactable(
 5      searchable = TRUE,
 6      highlight = TRUE,
 7      filterable = TRUE,
 8      defaultPageSize = 5,
 9      showPageSizeOptions = TRUE,
10      pageSizeOptions = c(5, 10, 15)
11    )
```

Your Turn: Interactive Charts & Tables

10:00

1. Open `practice.qmd`
2. Take a chart from a past class or homework and make it interactive with `ggplotly()`, or build it directly using the `plotly` package.
3. Make a `reactable()` table of the gapminder data
4. Check the tooltips, and compare against the static version
5. Post your favorite in Slack `#class` channel

Week 13: *Interactivity*

1. JavaScript graphics: packages
2. JavaScript graphics: D3 + Claude
3. Shiny apps
4. Shiny extras

What packages can’t do easily

- A **custom scale** or axis
- A **custom animation** (not just “play through the frames”)
- A bespoke interaction: linked views, brushing, step-by-step stories
- A layout nobody wrote a function for

Enter D3.js

- The JavaScript library underneath most custom interactive graphics (The Pudding, NYT, The Guardian)
- It can build **anything**, so it's much more complex than `ggplotly()`
- Browse the D3 Graph Gallery to see what's possible

D3 code for roughly what `geom_point()` does

Part of a basic scatterplot (full file)

```
// Scales: map data values to pixel positions
const x = d3.scaleLog()
  .domain([200, 60000])
  .range([margin.left, width - margin.right]);

const y = d3.scaleLinear()
  .domain([40, 85])
  .range([height - margin.bottom, margin.top]);

// Axes
svg.append("g")
  .attr("transform", `translate(0,${height - margin.bottom})`)
  .call(d3.axisBottom(x));

// Marks: one circle per row of data
svg.selectAll("circle")
  .data(data)
  .join("circle")
    .attr("cx", d => x(d.gdpPercap))
    .attr("cy", d => y(d.lifeExp))
    .attr("r", d => r(d.pop))
    .attr("fill", d => color(d.continent));
```

You don’t write D3. You prompt, and iterate.

You

- Decide what the chart should show
- Describe it **precisely**
- Check that it’s right

Claude

- Writes the JavaScript
- Revises until it looks right

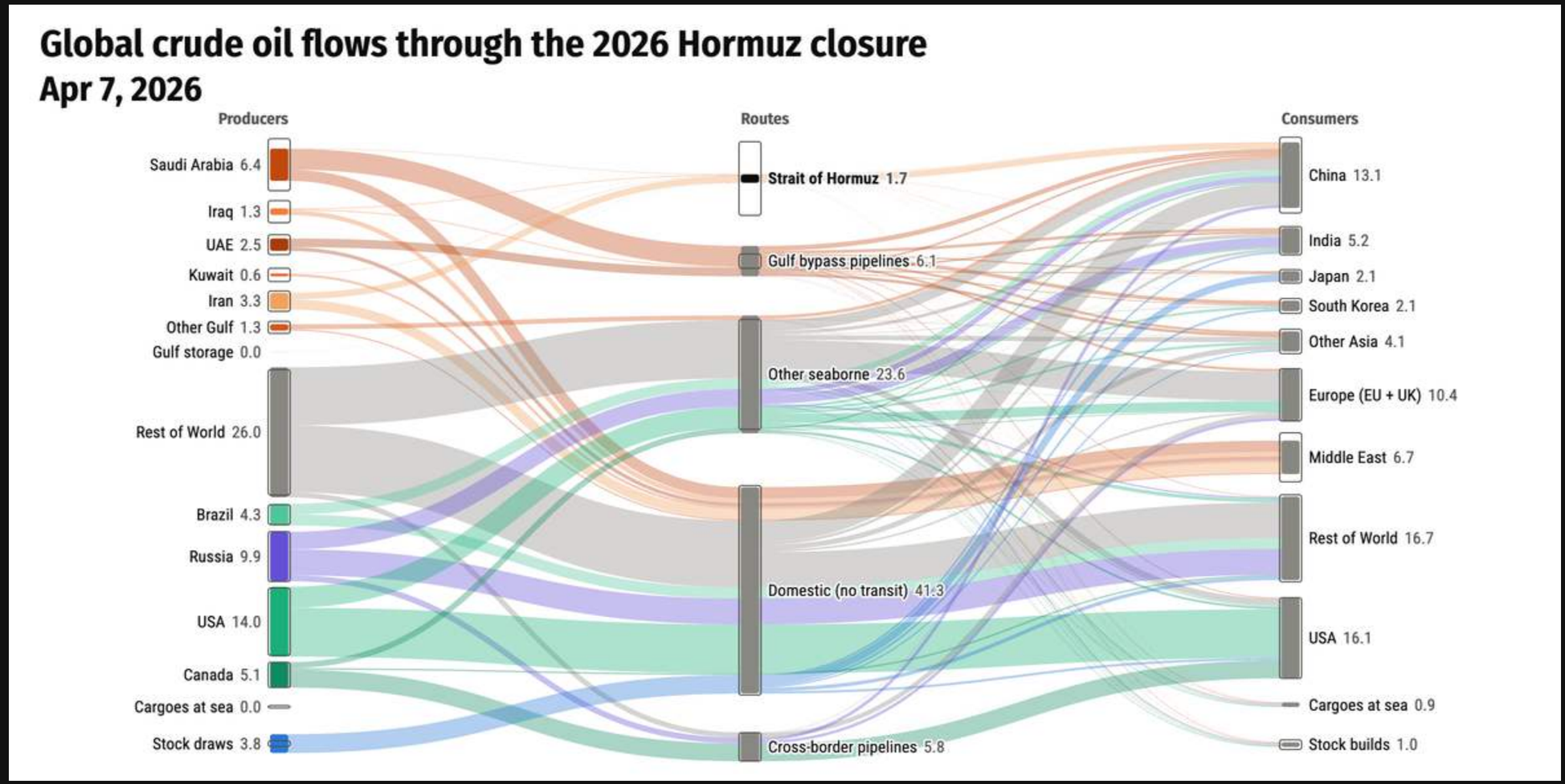
How do you verify it without reading the code?

Verify against the data, not the code

1. Build a **static chart** first and check it carefully (your ggplot, or a plotly version)
2. Prompt Claude to bring that same chart to life as an interactive, animated version
3. **Compare** the interactive chart to the static one, and to the data

The static chart is your reference

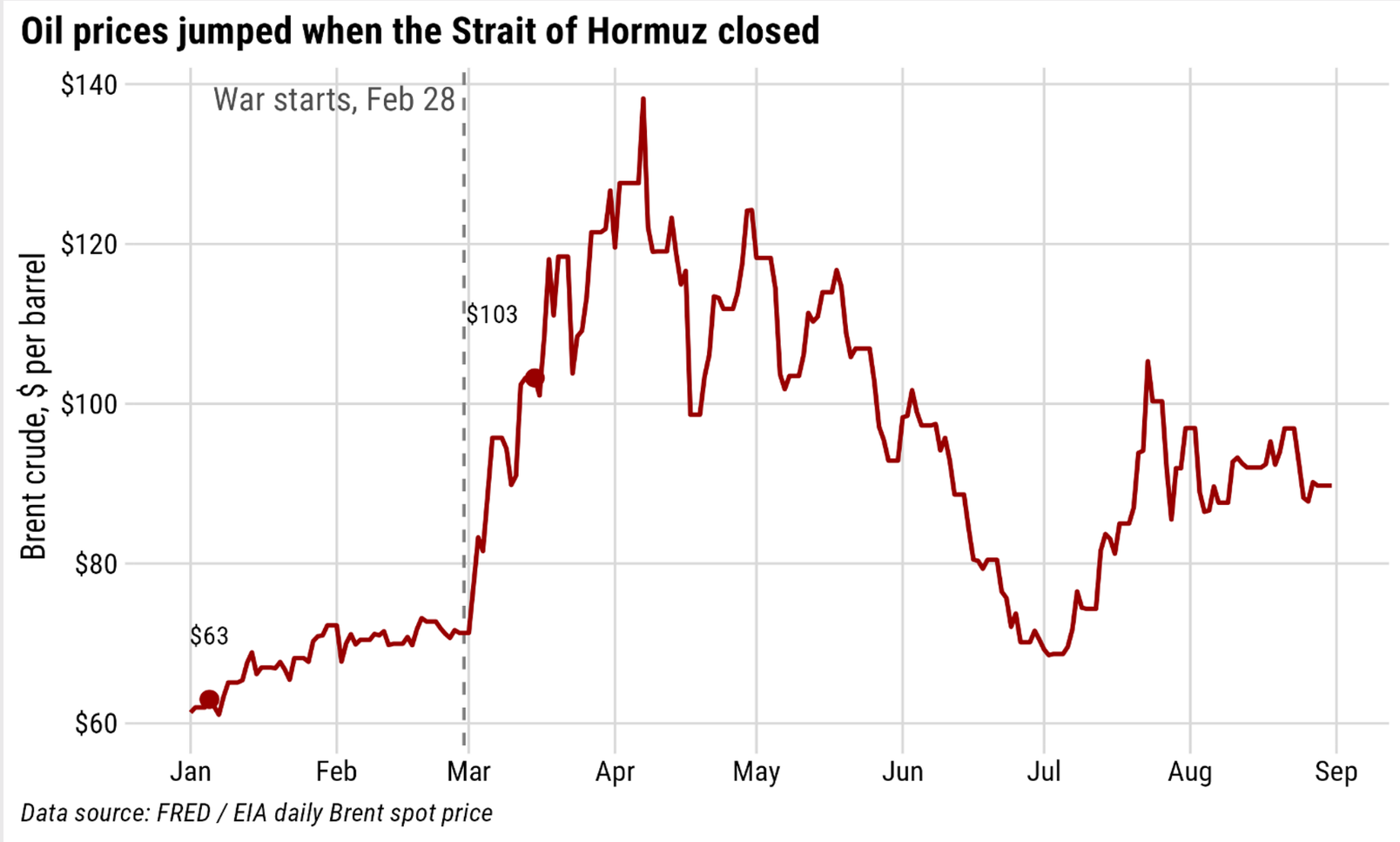
A real example: global oil flows through the 2026 Hormuz closure



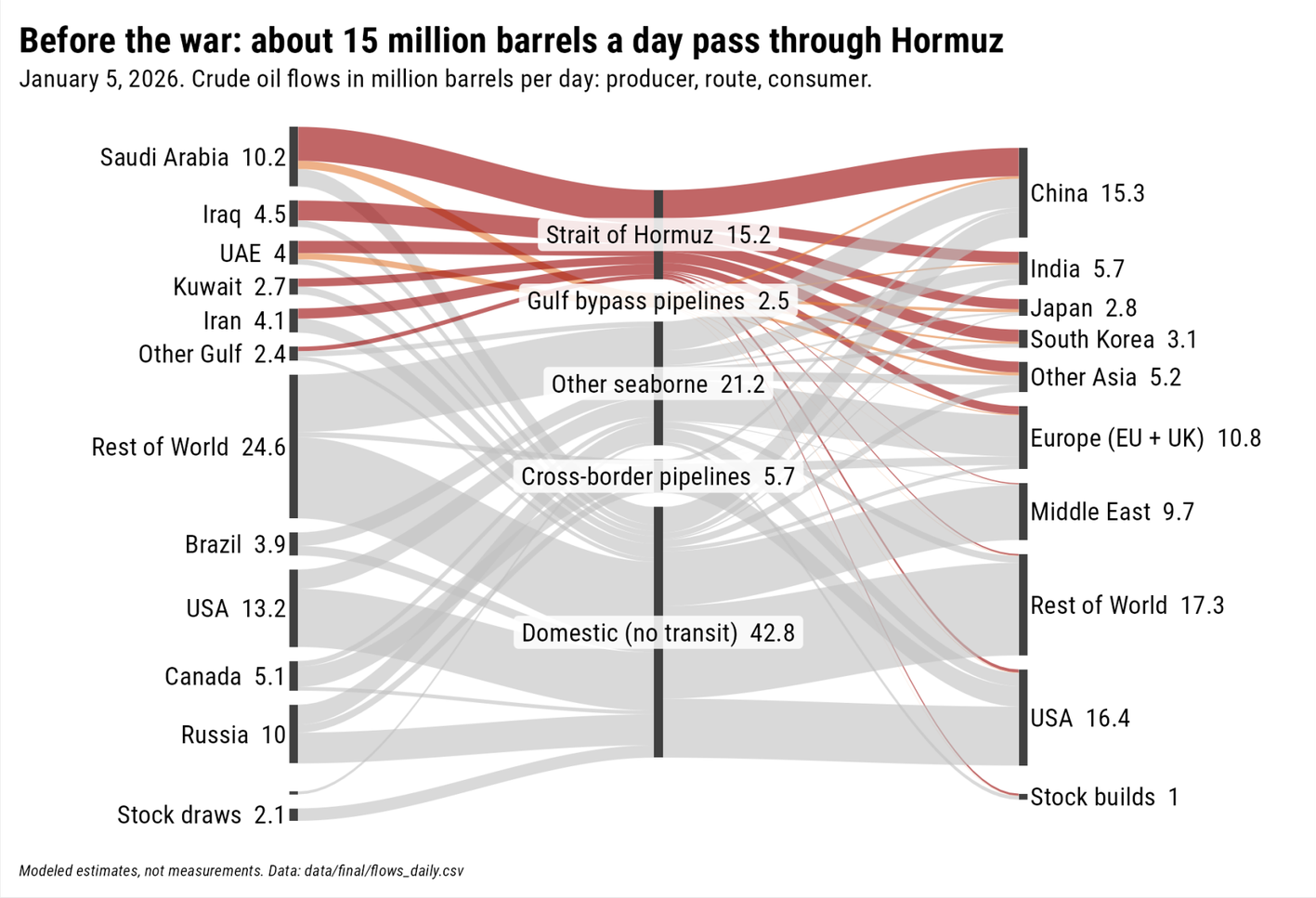
How I got there

1. Find the data, working with Claude (prices, tanker traffic, trade flows)
2. Make simple static figures to prove the data is right and I know what the final chart should look like
3. Build up to the full figure, entirely in D3, by prompting Claude and iterating

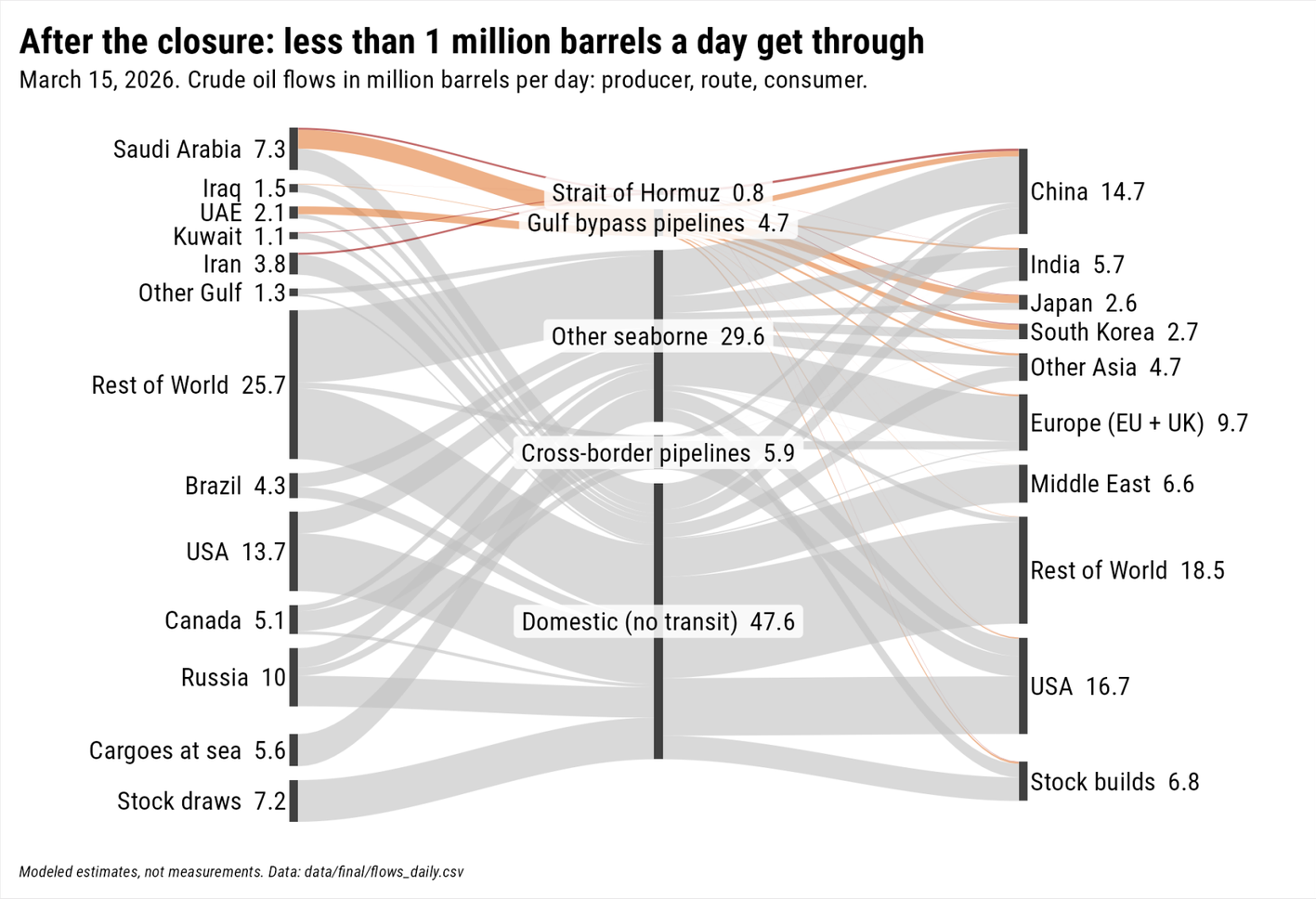
Static figure 1: do the prices look right?



Static figure 2: do the flows look right? (early January)



Static figure 3: and after the closure? (mid-March)



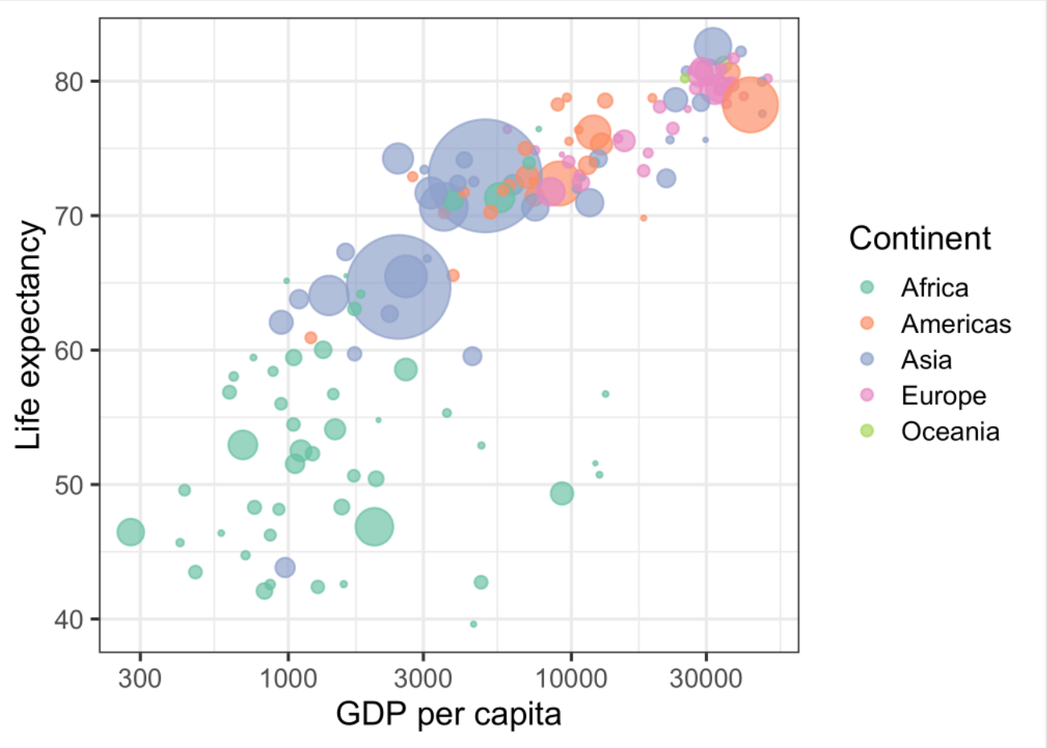
Now I know what the final figure must show

- Brent jumps from about **$63 to $103** by mid-March
- The Hormuz band shrinks from **15.2 to 0.8** million barrels/day
- Every other number in the static figures has to appear in the final one

The workflow

1. Make a static chart and verify it
2. Sketch what the interactive version should do
3. Prompt Claude with your code, your data, and your sketch
4. Iterate until it looks right
5. Verify against the static chart and the data

Example, step 1: a static chart we've already verified



What we know from the data:

- 142 countries in 2007
- China and India are the biggest bubbles
- Life expectancy ranges from about 40 to 83 years

Example, steps 2-3: the prompt

Here is my static chart (code below) and the data (`gapminder.csv`). Build an **animated, interactive version in D3**:

- One bubble per country, same axes and colors as the static chart
- A **year slider** (1952-2007) and a **play button** that animates between years
- **Continent filter** buttons, and a **tooltip** on hover
- A title that states the main takeaway

Put it in a single self-contained file called `d3-chart.html`, loading D3 from a CDN. **Embed the data directly in the file.**

Example, step 4: the result

Example, step 5: verify against the static chart and the data

1. Set the slider to **2007**: does it match the static chart?
2. **Hover** over a bubble: do the values match the data?
3. **Count** the bubbles: 142?
4. Check the **axes**: ranges, units, labels
5. Click **everything**: first and last year, each filter

```
1  gapminder %>%
2    filter(country == "China", year == 2007)
```

```
#> # A tibble: 1 × 6
#>   country continent  year lifeExp        pop gdpPercap
#>   <chr>   <chr>     <dbl>   <dbl>      <dbl>     <dbl>
#> 1 China   Asia       2007    73.0 1318683096     4959.
```

```
1  gapminder %>%
2    filter(year == 2007) %>%
3    nrow()
```

```
#> [1] 142
```

Checks for any chart you didn’t write by hand

1. **The numbers.** Hover over several points and compare to your data
2. **The axes.** Units? Start points? Resize the window: does it still work?
3. **The claim.** Does the data support exactly what the title says?
4. **The interaction.** Does every control do what it says?

If something looks off, tell Claude exactly what’s wrong and give it the right value from your data

Gotchas

1. **Embed the data in the file.** A local `.html` can’t load a local CSV (browsers block it), and the chart will be blank
2. **Load D3 from a CDN.** A public server hosts the library and the file links to it (like `library(d3)`), so viewing the chart needs an internet connection
3. **Open it in the browser** and check the console (right-click → Inspect → Console) for red errors
4. To embed in a `.qmd`, the `.html` must be **in the same folder** and **committed**:

```
<iframe src="d3-chart.html" width="100%" height="500" style="border:none;"></iframe>
```

Your Turn: A Custom Chart in D3

20:00

1. Make a **static chart** (any data set) and check it
2. **Sketch** the interactive version: what moves, what hovers, what filters?
3. **Prompt** Claude to build it in `d3-chart.html`
4. **Verify** it against your static chart and the data
5. **Embed** it in `practice.qmd` with an iframe

Fast finishers: add a second interaction, or a whole second chart

One more thing to consider...

Make a “scrollytelling” page in Quarto:
https://github.com/nrennie/scrollytelling

Examples:

- US is addicted to trucks SUVs
- The clubs that connect the world cup

Intermission

05:00

Week 13: *Interactivity*

1. JavaScript graphics: packages
2. JavaScript graphics: D3 + Claude
3. Shiny apps
4. Shiny extras

License

These slides were modified from Florencia D'Andrea's RLadies Shiny Meetup Slides

This work is licensed under a
Creative Commons Attribution-ShareAlike 4.0 International License

New libraries to install

```
1 install.packages('shiny')
2 install.packages('shinyWidgets')
3 install.packages('rsconnect')
```



Interactive Webapps in R

Check out the Shiny Gallery

Anatomy of a Shiny App

A Shiny app is a web page (UI) connected to a computer running a live R session (Server)



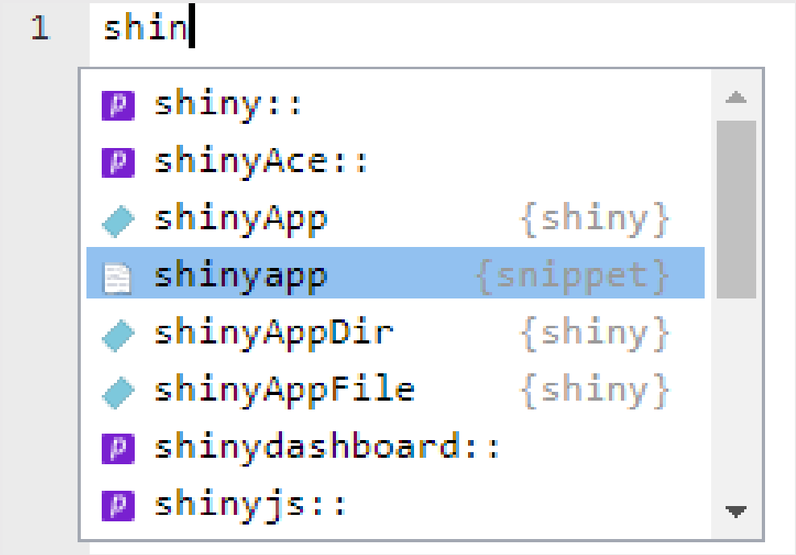




app.R

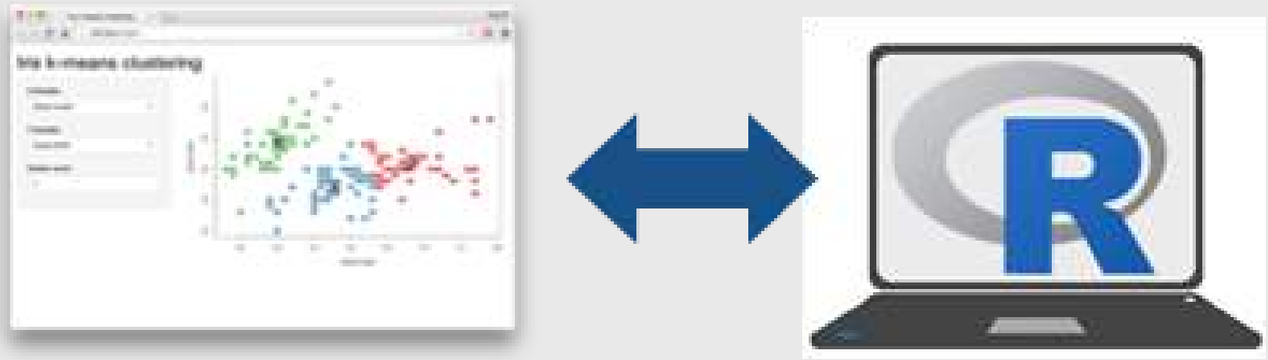
You can insert all the code at once with the shinyapp snippet!

Just start typing `shiny`...




```
 1  library(shiny)
 2
 3  ui <- fluidPage(
 4
 5  )
 6
 7  server <- function(input, output, session) {
 8
 9  }
10
11  shinyApp(ui, server)
```

Building a shiny app



`ui`

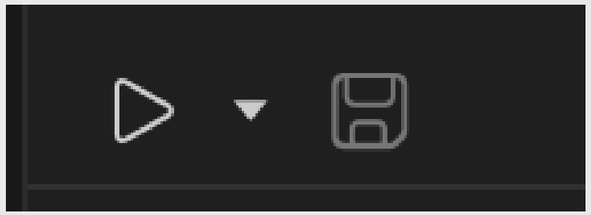
1. Pick a layout function
2. Add inputs widgets
3. Add `*Output()` functions

`server`

1. Use `render*()` functions to make outputs
2. Link outputs with `output$<id>`
3. Link inputs with `input$<id>`

Run the app 🎬

- **Option 1**: Click the “Run App” button in the toolbar:



- **Option 2**: `shiny::runApp()` with the path to the **app.R** file.

Your Turn

hello_shiny.app

Create a new file called `app.R` in Positron, then copy in this code

```r
library(shiny)

# Define UI for application that draws a histogram
ui <- fluidPage(

    # Application title
    titlePanel("Old Faithful Geyser Data"),

    # Sidebar with a slider input for number of bins
    sidebarLayout(
        sidebarPanel(
            sliderInput("bins",
                        "Number of bins:",
                        min = 1,
                        max = 50,
                        value = 30)
        ),

        # Show a plot of the generated distribution
        mainPanel(
           plotOutput("distPlot")
        )
    )
)

# Define server logic required to draw a histogram
server <- function(input, output) {

    output$distPlot <- renderPlot({
        # generate bins based on input$bins from ui.R
        x    <- faithful[, 2]
        bins <- seq(min(x), max(x), length.out = input$bins + 1)

        # draw the histogram with the specified number of bins
        hist(x, breaks = bins, col = 'darkgray', border = 'white')
    })
}

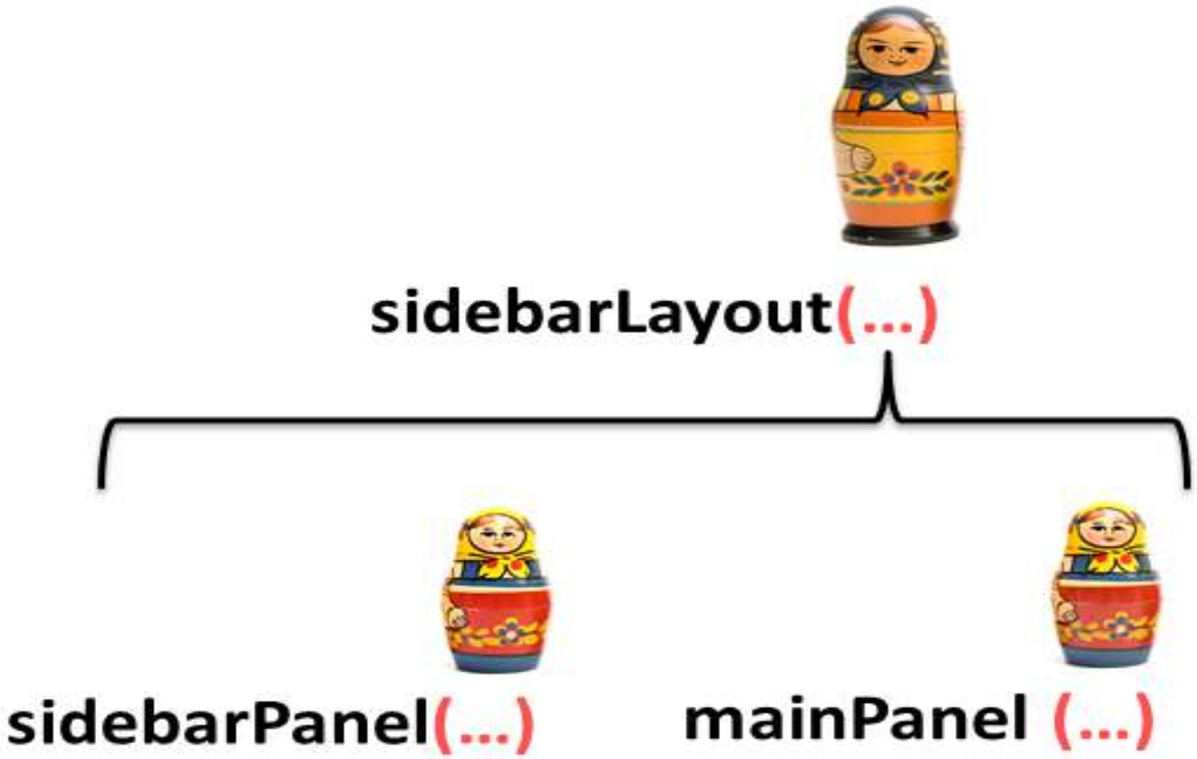
# Run the application
shinyApp(ui = ui, server = server)
```

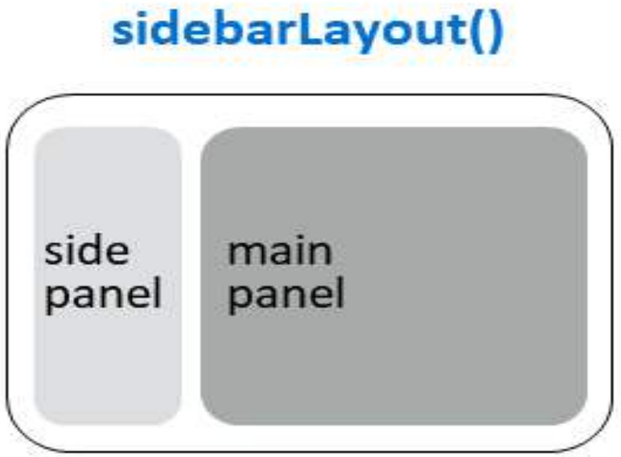
User Interface (UI)



Organize panels and elements into a layout with a **layout function**

Top level is usually `fluidPage()`


sidebarLayout(...)
sidebarPanel(...)
mainPanel (...)


sidebarLayout()
side
panel
main
panel

sidebarLayout()

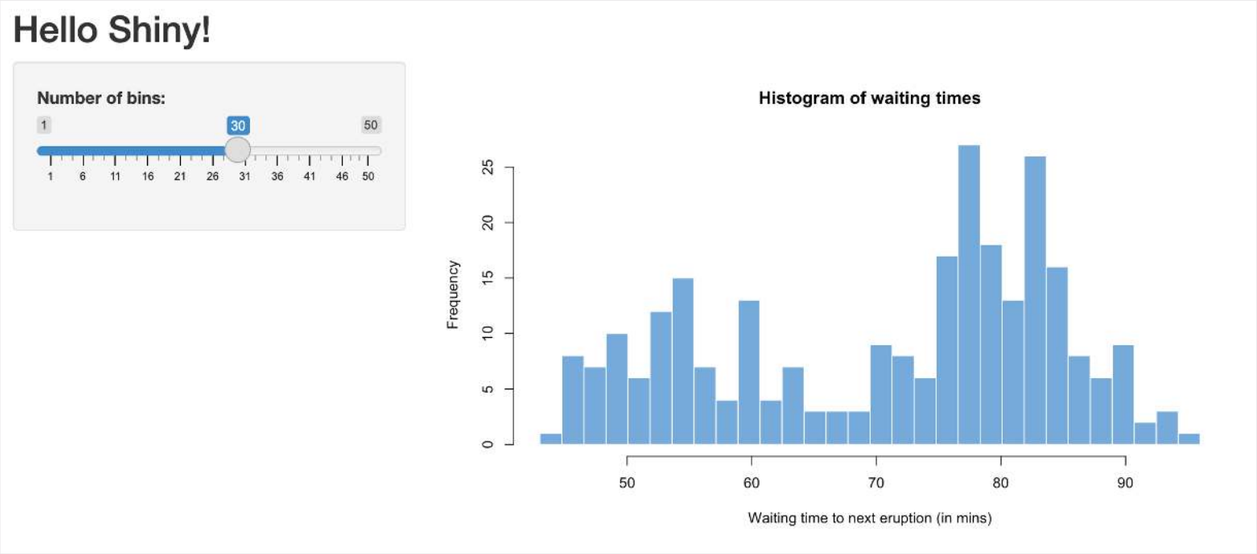
```
ui <- fluidPage(

  titlePanel("Hello Shiny!"),

  sidebarLayout(

    sidebarPanel(
      sliderInput(
        "bins", label = "Number of
        min = 1, value = 30, max =
      )
    ),

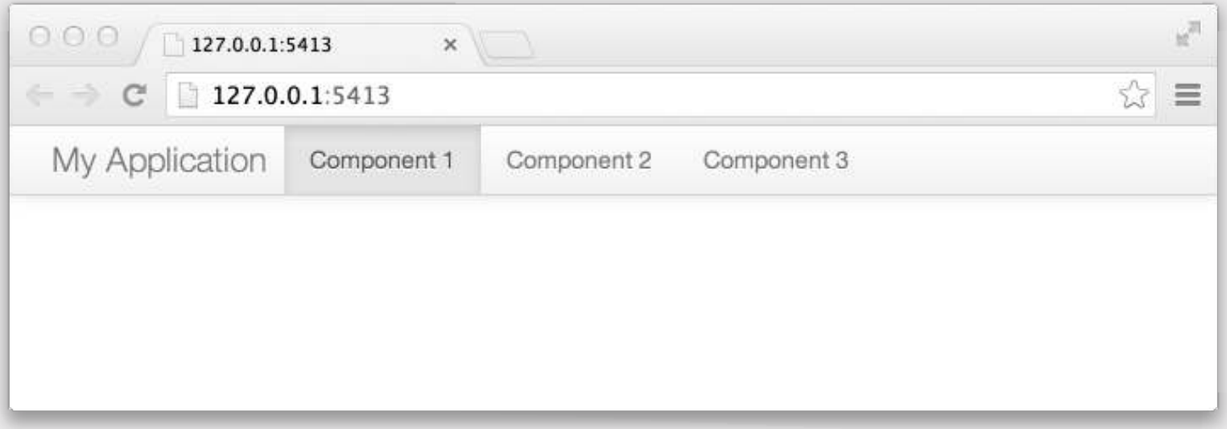
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```



`navbarPage()`: An alternative to `fluidPage()`

Think of each `tabPanel()` as it's own `fluidPage()`

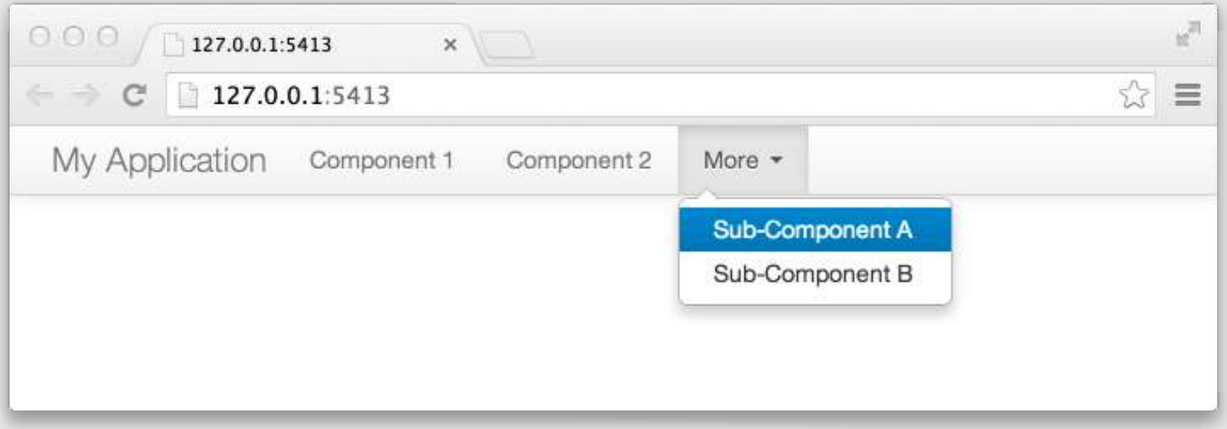
```
1 ui <- navbarPage("My Application",
2   tabPanel("Component 1"),
3   tabPanel("Component 2"),
4   tabPanel("Component 3")
5 )
```



`navbarPage()`: An alternative to `fluidPage()`

Use `navbarMenu()` to create a nested menu item

```
1 ui <- navbarPage("My Application",
2   tabPanel("Component 1"),
3   tabPanel("Component 2"),
4   navbarMenu("More",
5     tabPanel("Sub-Component A"),
6     tabPanel("Sub-Component B"))
7 )
```



The UI defines the “what” and “where” for:

1. **Inputs**: collect values from the user
2. **Output**: display something to the user

Inputs: collect values from the user

Buttons

Action

Submit

actionButton()
submitButton()

Single checkbox

Choice A

checkboxInput()

Checkbox group

Choice 1
Choice 2
Choice 3

checkboxGroupInput()

Date input

2014-01-01

dateInput()

Colour input

#52CC4E

shinyjs::colourInput()

Date range

2014-01-24 to 2014-01-24

dateRangeInput()

File input

Choose File No file chosen

fileInput()

Numeric input

1

numericInput()

Password Input

••••••••••

passwordInput()

Radio buttons

Choice 1
Choice 2
Choice 3

radioButtons()

Select box

Choice 1

selectInput()

Sliders



sliderInput()

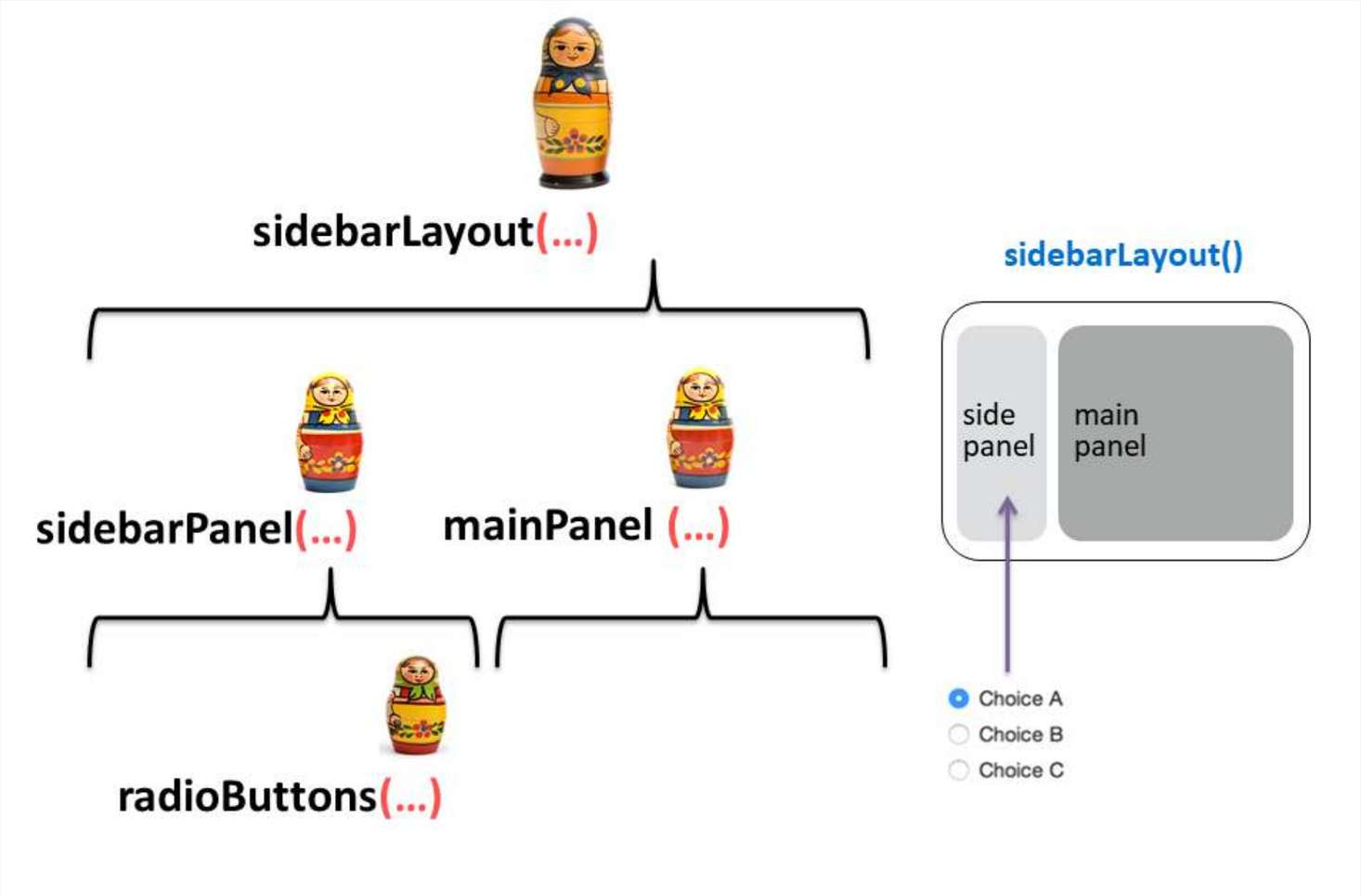
Text input

Enter text...

textInput()

Example Input: Radio buttons in the sidebar

```
1 ui <- fluidPage(
2   sidebarLayout(
3     sidebarPanel(
4       radioButtons(…)
5     ),
6     mainPanel(…)
7   )
8 )
```





((((((((( )))))))))

```
        ) #mainPanel
    ) #sidebarLayout
) #fluidPage
```

Quick practice

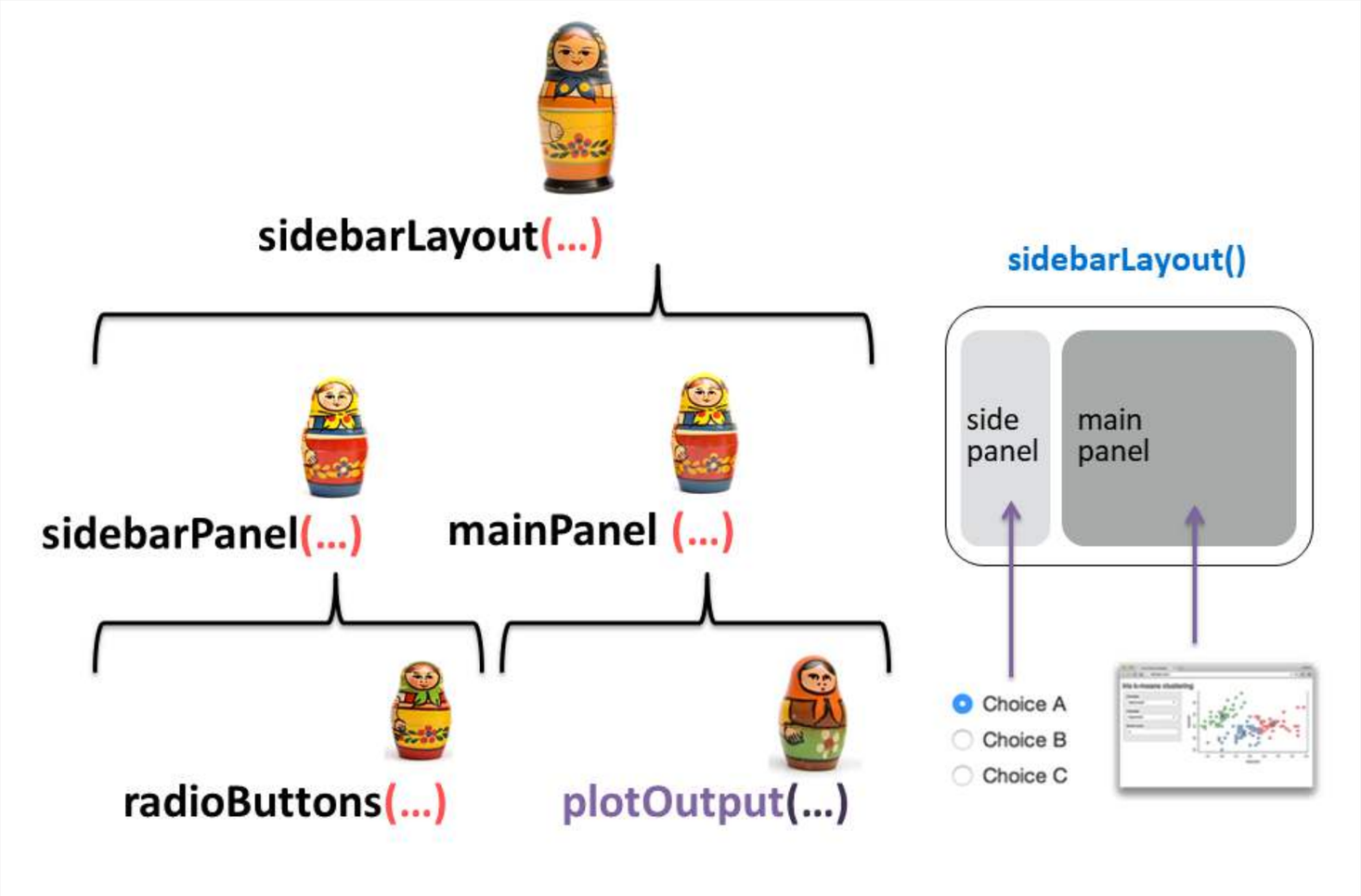
1. Open the `widgets.R` file.
2. Run the app (Click “run app” button).
3. Go to the Shiny Widgets Gallery to see other input widgets.

The UI defines the “what” and “where” for:

1. **Inputs**: collect values from the user
2. **Output**: display something to the user

Example Output: Put a plot in the main panel

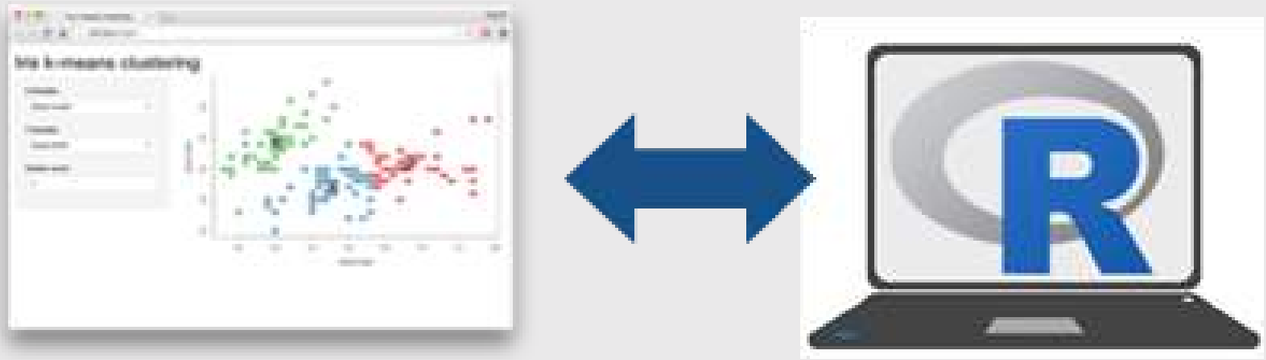
```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      radioButtons(…)
    ),
    mainPanel(
      plotOutput(…)
    )
  )
)
```



Output: display something to the user

Output function	Description
`plotOutput()`	Display a reactive *plot*
`dataTableOutput()`	Display a `DT::datatable()`
`textOutput()`	Display reactive *text*
`imageOutput()`	Display an image

Building a shiny app



`ui`

1. Pick a layout function, e.g. `sidebarLayout()`
2. Add inputs widgets
3. Add `*Output()` functions

`server`

1. Use `render*()` functions to make outputs
2. Link outputs with `output$<id>`
3. Link inputs with `input$<id>`

Outputs - render*() and *Output() functions work together to add R output to the UI



DT::**renderDataTable**(expr, options, callback, escape, env, quoted)

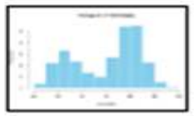
works with

dataTableOutput(outputId, icon, . . .)



renderImage(expr, env, quoted, deleteFile)

imageOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)



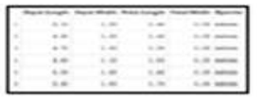
renderPlot(expr, width, height, res, . . ., env, quoted, func)

plotOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)



renderPrint(expr, env, quoted, func, width)

verbatimTextOutput(outputId)



renderTable(expr,. . ., env, quoted, func)

tableOutput(outputId)

foo

renderText(expr, env, quoted, func)

textOutput(outputId, container, inline)



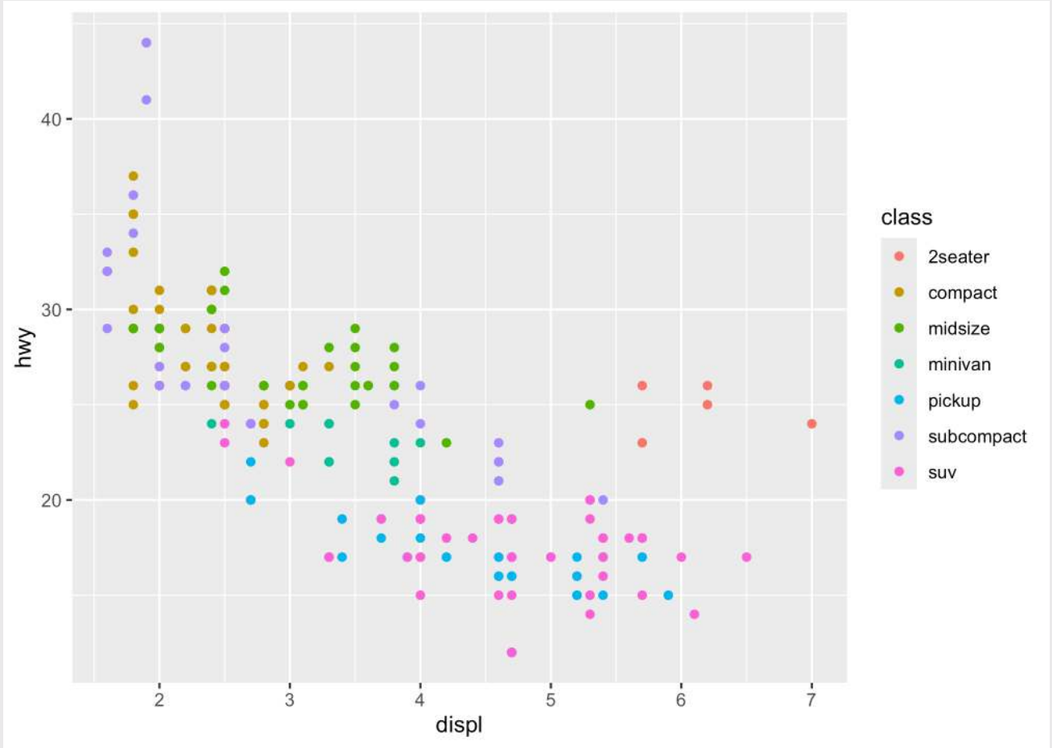
renderUI(expr, env, quoted, func)

uiOutput(outputId, inline, container, . . .)

& **htmlOutput**(outputId, inline, container, . . .)

Using `renderPlot()`: make a plot

```
1 library(ggplot2)
2
3 ggplot(mpg) +
4   geom_point(
5     aes(x = displ, y = hwy, color = class))
```



Link plot to output with `output$<id>`

ui

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      radioButtons(…)
    ),
    mainPanel(
      plotOutput(
        outputId = "mpg_plot"
      )
    )
  )
)
```

server

```
server <- function(input, output, session) {
  output$mpg_plot <- renderPlot({
    ggplot(mpg) +
      geom_point(
        aes(x = displ, y = hwy, color = class))
  })
}
```

Link user inputs to plot with `input$<id>`

`ui`

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      radioButtons(
        inputId = "xvar",
        label = "Select the x-axis variable:",
        selected = "displ",
        choices = c(
          "Highway miles per gallon" = "hwy",
          "City miles per gallon" = "cty",
          "Engine displacement, in litres" = "displ")
      ),
    ),
    mainPanel(
      plotOutput(
        outputId = "mpg_plot"
      )
    )
  )
)
```

`server`

```
server <- function(input, output, session) {
  output$mpg_plot <- renderPlot({
    ggplot(mpg) +
      geom_point(
        aes_string(
          x = input$xvar,
          y = "hwy",
          color = "class"))
  })
}
```

Note: I switched the ggplot code from `aes()` to `aes_string()`

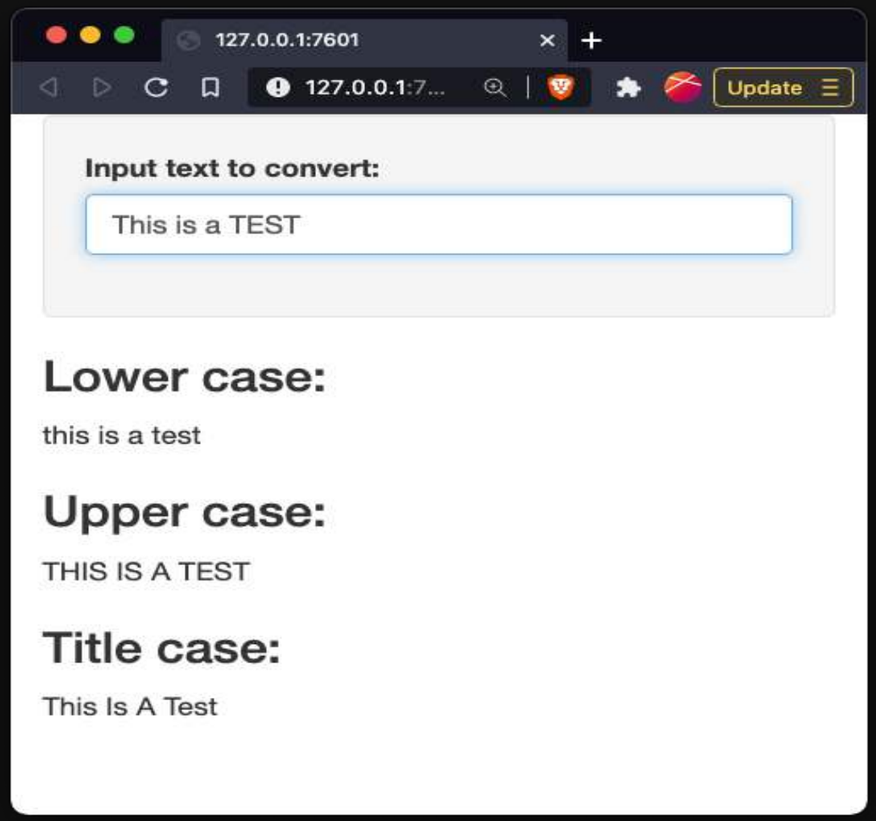
Quick practice

Open the `mpg.R` app and click the “Run App” button

Your Turn

10:00

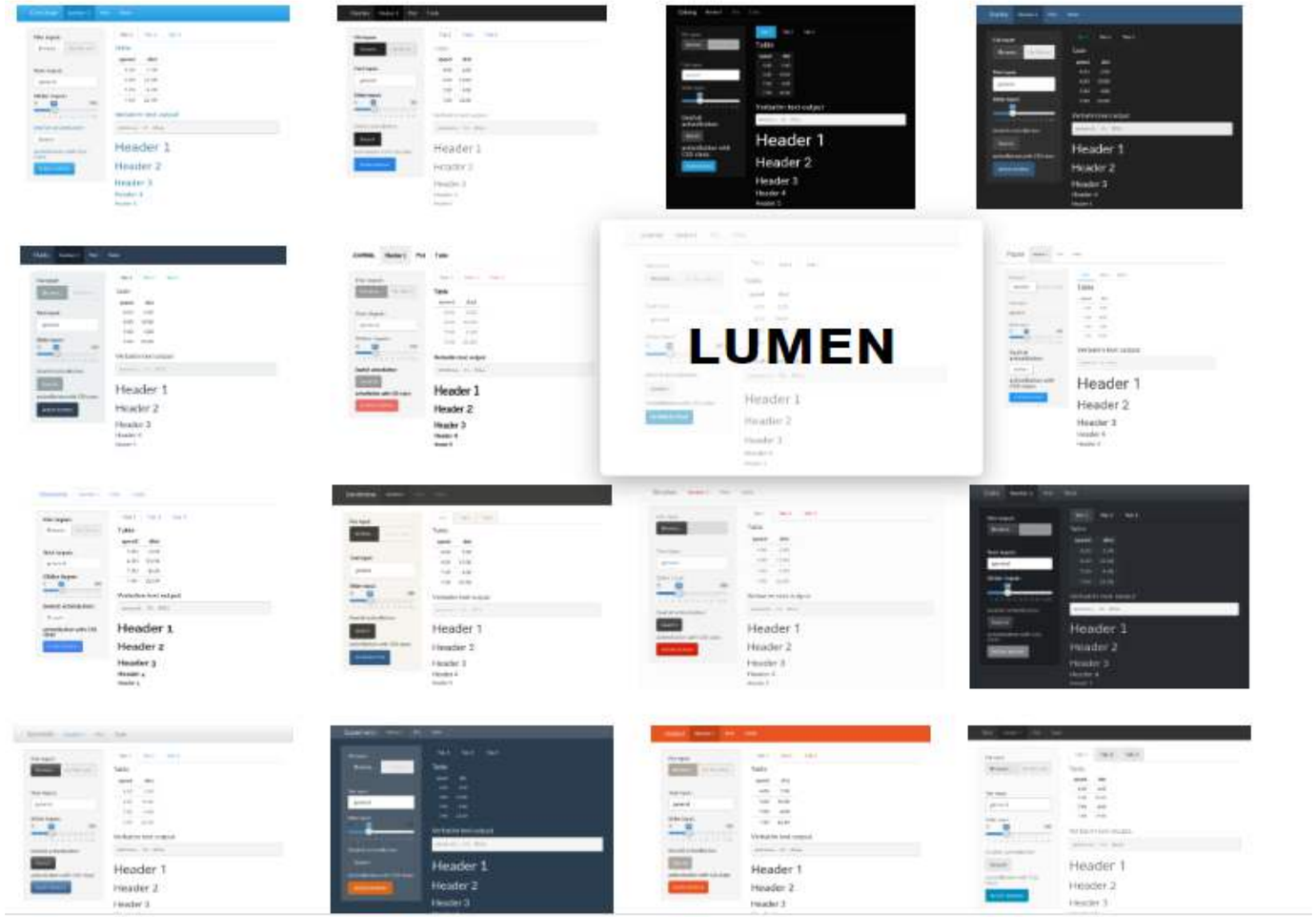
1. Open the `caseConverter.R` file.
2. In the `server`: Write code in the provided `renderText()` to convert the input text to lower case.
3. Run the app and test that it's working.
4. In the `ui` main panel: Add two more `textOutput()` functions for also displaying the input text in "upper" case and "title" case.
5. In the `server`: Define two more outputs to convert the input text to "upper" case and "title" case.



Week 13: *Interactivity*

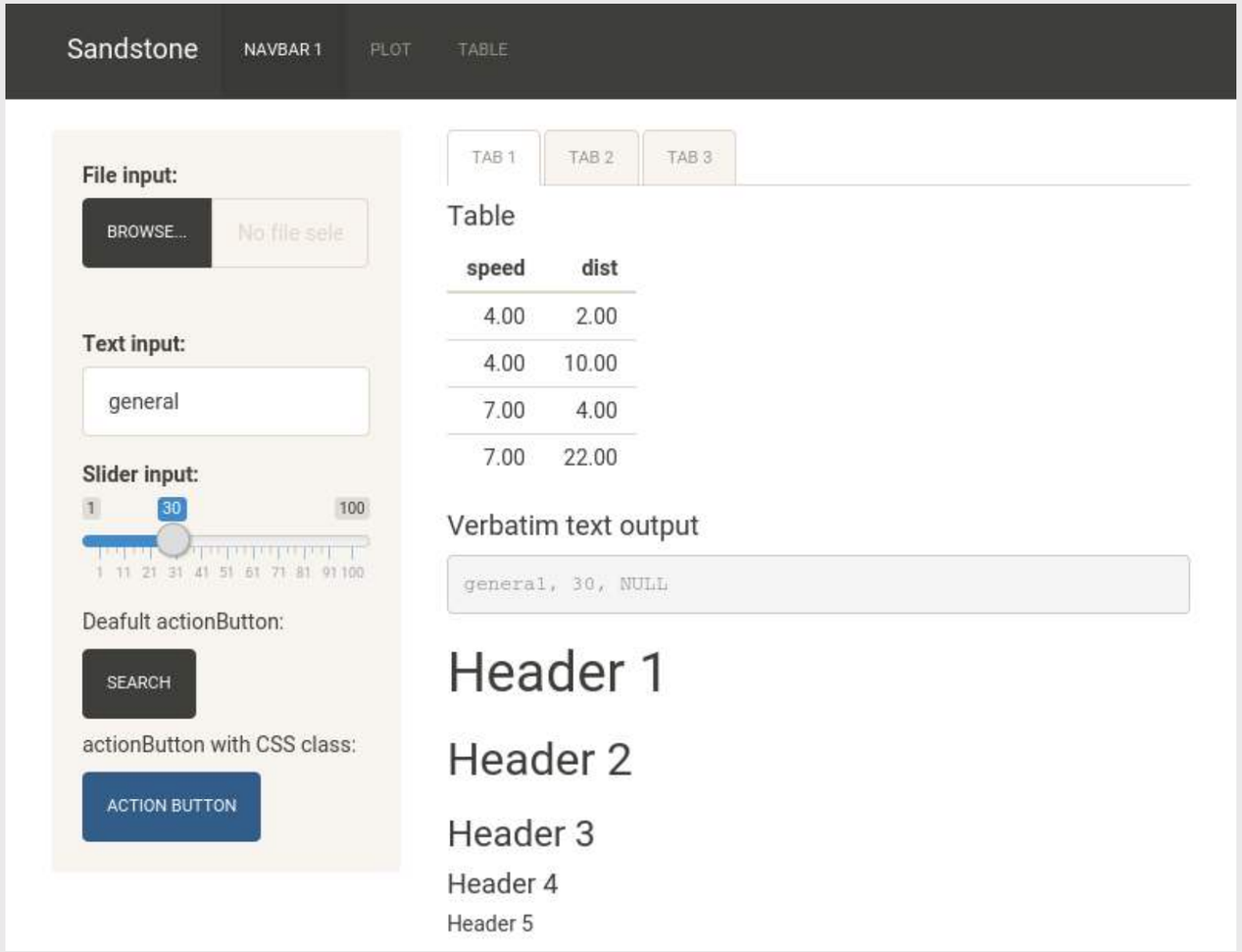
1. JavaScript graphics: packages
2. JavaScript graphics: D3 + Claude
3. Shiny apps
4. Shiny extras

Add a theme with “shinythemes” package



Insert theme at top of main `ui` layout function

```
1  library(shinythemes)
2
3  ui <- fluidPage(
4    theme = shinytheme("sandstone"),
5    sidebarLayout(
6      sidebarPanel(
7        <insert widgets>
8      ),
9      mainPanel(
10       <insert outputs>
11     )
12   )
13 )
```



Fancier widgets with “shinyWidgets” package

Open the `shinyWidgets.R` app and click the “Run App” button

If you really want to get good at this:

1. Print out this Cheatsheet

2. Watch this 2.5 Hour Comprehensive RStudio Tutorial

3. Use this reference manual: Mastering Shiny

You can deploy an app for free on Posit Connect Cloud

(shinyapps.io is being retired in favor of Connect Cloud)

Use **Posit Publisher**, which comes built into Positron:

1. Click the **Posit Publisher** icon in the left sidebar
2. Under **Credentials**, click **+** → **Posit Connect Cloud** and sign in
3. Click **+** to add a deployment, pick `app.R`, check the project files, and click **Deploy Your Project**

Your Turn

15:00

1. Open the `internetUsers.R` file.
2. Modify the server code so that the inputs control the plot.
3. (Optional) Deploy your app to Posit Connect Cloud

Next (and final!) class: Story-telling