

# REACTIVE PROGRAMMING & UNDERSTANDING UI

# OUTLINE

- ▶ Reactive Programming Part 1
  - ▶ Reactivity 101
  - ▶ Reactive objects
    - ▶ Reactive sources and endpoints
    - ▶ Reactive conductors
    - ▶ Implementation
    - ▶ Observers and side effects
  - ▶ Render functions
- ▶ Understanding UI
  - ▶ Ladder of Progression
  - ▶ High Level View
  - ▶ Shiny built-ins
  - ▶ External packages
  - ▶ HTML Tools
  - ▶ RAW HTML





# REACTIVE PROGRAMMING

## PART 1

# Reactivity

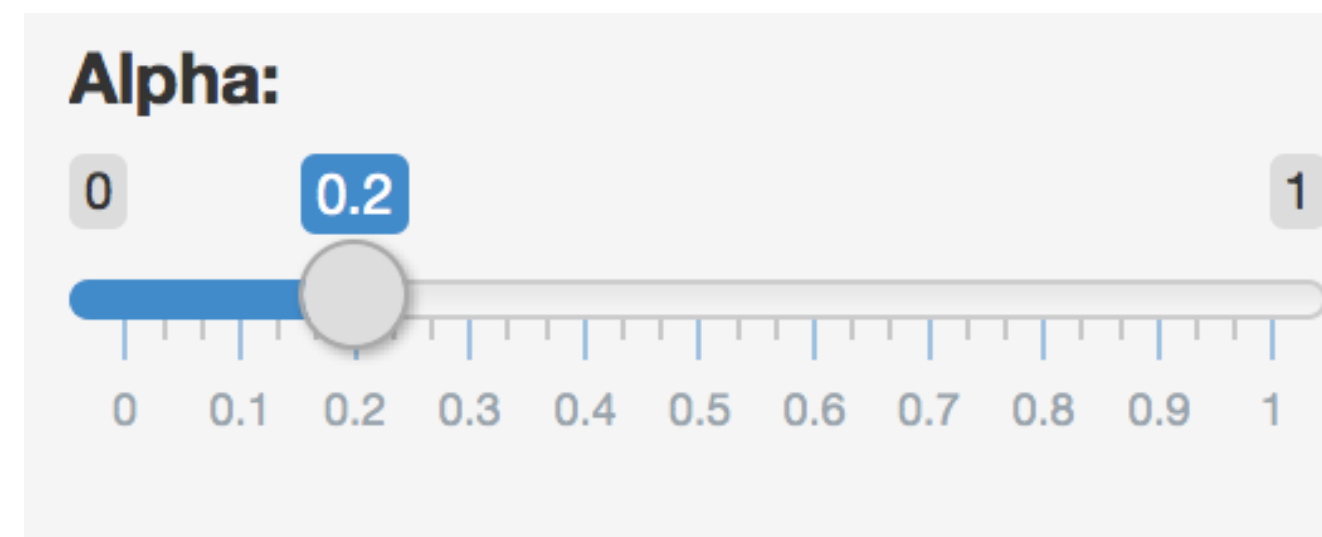
101

# REACTIONS

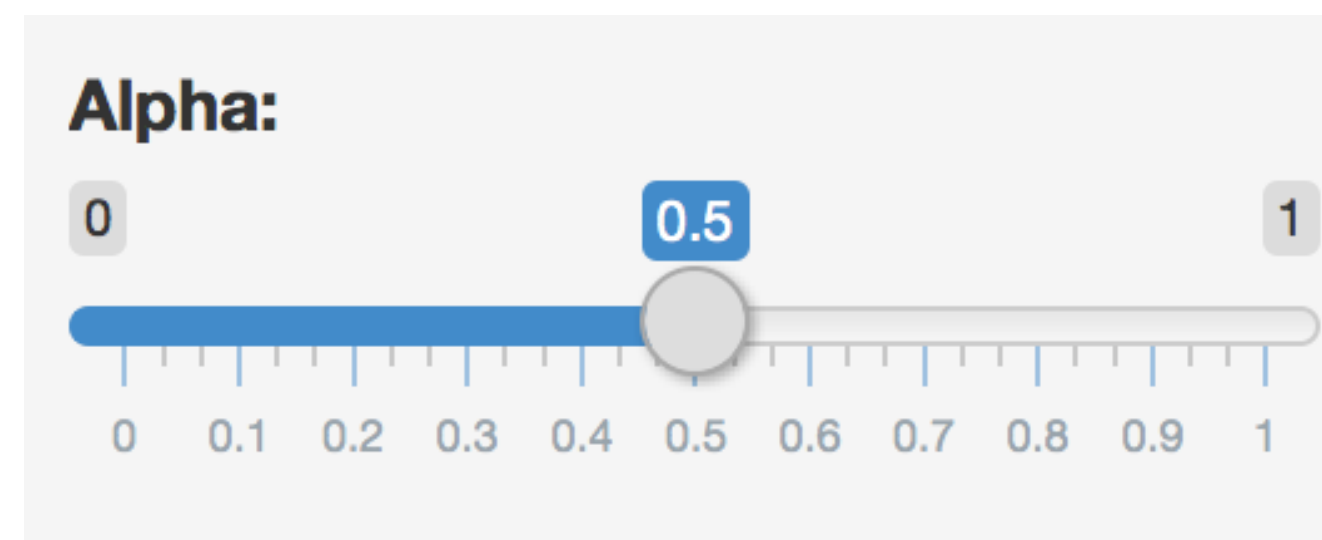
The **input\$** list stores the current value of each input object under its name.

```
# Set alpha level  
sliderInput(inputId = "alpha",  
            label = "Alpha:",  
            min = 0, max = 1,  
            value = 0.5)
```

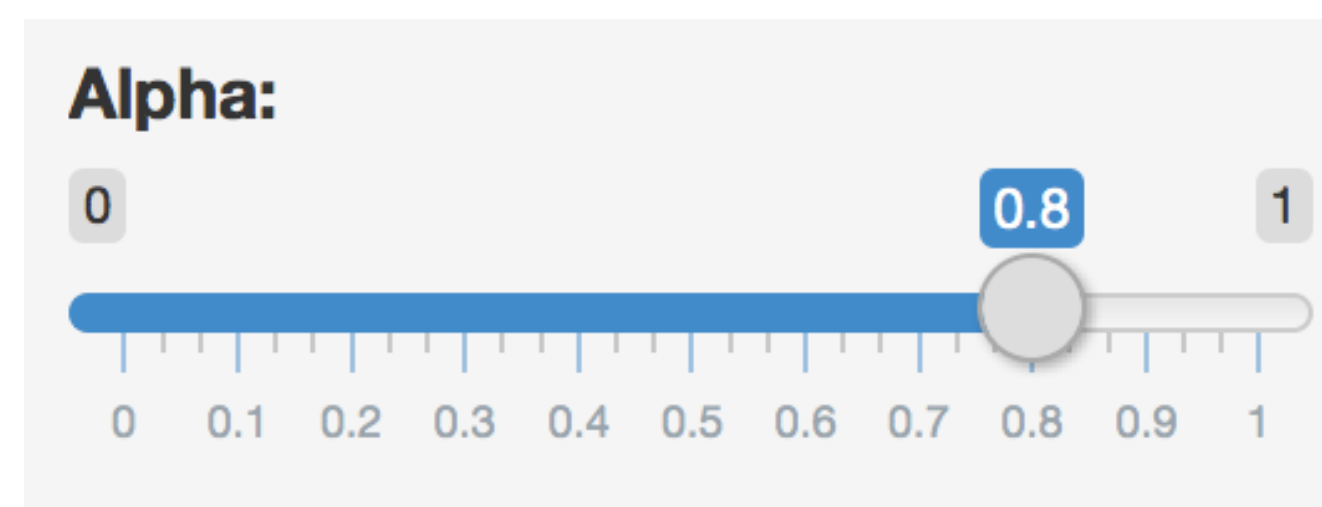
**input\$alpha**



**input\$alpha** = 0.2



**input\$alpha** = 0.5



**input\$alpha** = 0.8



# REACTIVITY 101

Reactivity automatically occurs when an input value is used to render an output object

```
# Define server function required to create the scatterplot
server <- function(input, output) {
  # Create the scatterplot object the plotOutput function is expecting
  output$scatterplot <- renderPlot(
    ggplot(data = movies, aes_string(x = input$x, y = input$y,
                                     color = input$z)) +
    geom_point(alpha = input$alpha)
  )
}
```

# EXERCISE



- ▶ Go back to the app you built last class
- ▶ Add a new sliderInput defining the size of points (ranging from 0 to 5)
- ▶ Use this variable in the geom of the ggplot function as the size argument
- ▶ Run the app to ensure that point sizes react when you move the slider
- ▶ Compare your code / output with the person sitting next to / nearby you

3<sub>m</sub> 00<sub>s</sub>



# SOLUTION

Solution to the previous exercise

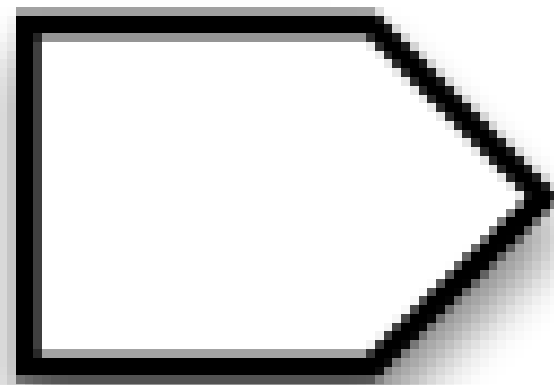
`movies_06.R`



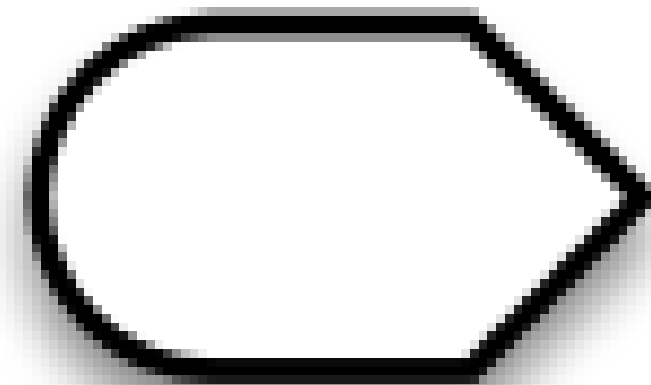
# Reactive objects

# TYPES OF REACTIVE OBJECTS

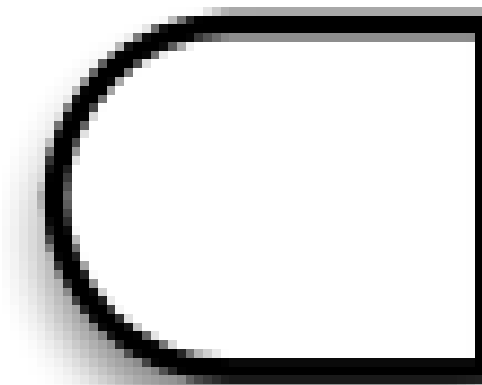
Reactive source



Reactive conductor



Reactive endpoint



# Reactive sources and endpoints



# SOURCES AND ENDPOINTS

- ▶ **Reactive source:** Typically, this is user input that comes through a browser interface
- ▶ **Reactive endpoint:** Something that appears in the user's browser window, such as a plot or a table of values

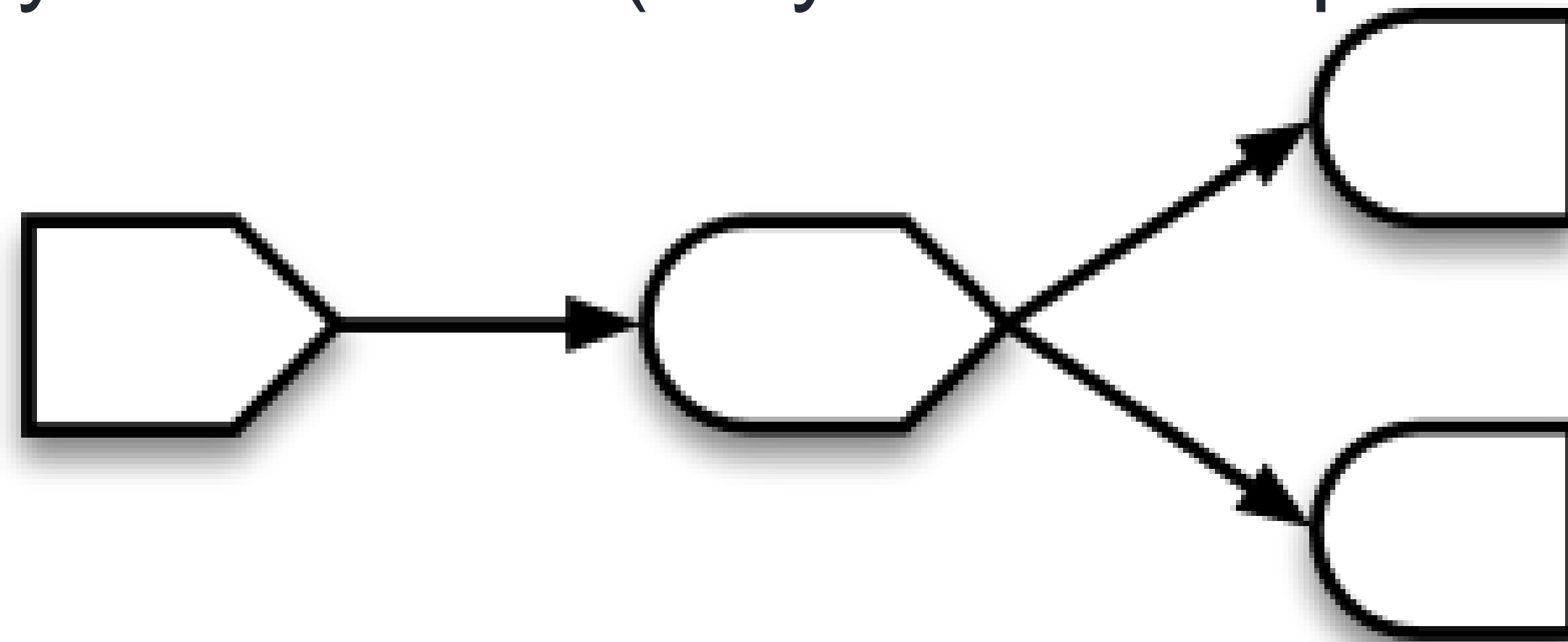


- ▶ This is the built-in reactivity discussed in the previous section
- ▶ A reactive source can be connected to multiple endpoints, and vice versa


# Reactive conductors

# CONDUCTORS

- ▶ **Reactive conductor:** Reactive component between a source and an endpoint
- ▶ A conductor can both be a dependent (child) and have dependents (parent)
  - ▶ Sources can only be parents (they can have dependents)
  - ▶ Endpoints can only be children (they can be dependents)





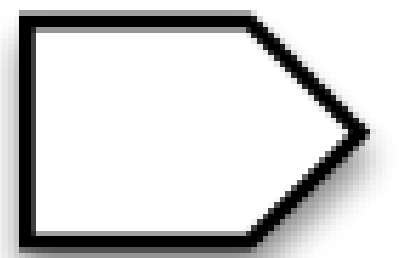


Suppose you want the option to plot only certain types of movies as well as report how many such movies are plotted:

1. Add a UI element for the user to select which type(s) of movies they want to plot
2. Filter for chosen title type and save as a new (reactive) expression
3. Use new data frame (which is reactive) for plotting
4. Use new data frame (which is reactive) also for reporting number of observations

- 
1. Add a UI element for the user to select which type(s) of movies they want to plot

```
# Select which types of movies to plot
checkboxGroupInput(inputId = "selected_type",
               label = "Select movie type(s):",
               choices = c("Documentary", "Feature Film", "TV Movie"),
               selected = "Feature Film")
```





## 2. Filter for chosen title type and save the new data frame as a reactive expression

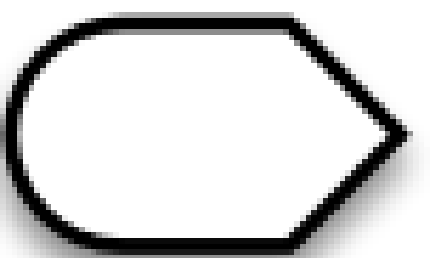
before app:

```
library(dplyr)
```

server:

```
# Create a subset of data filtering for chosen title type
movies_subset <- reactive({
  req(input$selected_type)
  filter(movies, title_type %in% input$selected_ty
})
```

Creates a **cached expression** that knows it is out of date when input changes





### 3. Use new data frame (which is reactive) for plotting

```
# Create the scatterplot object the plotOutput function is expecting
output$scatterplot <- renderPlot({
  ggplot(data = movies_subset(), aes_string(x =
    geom_point(...) +
    ...
  })
```

**Cached** - only re-run when  
inputs change

## 4. Use new data frame (which is reactive) also for printing number of observations

ui:

```
mainPanel(  
  ...  
  # Print number of obs plotted  
  uiOutput(outputId = "n"),  
  ...  
)
```

server:

```
# Print number of movies plotted  
output$n <- renderUI({  
  types <- movies_subset()$title_type %>%  
    factor(levels = input$selected_type)  
  counts <- table(types)  
  
  HTML(paste("There are",  
             counts,  
             input$selected_type,  
             "movies in this dataset."  
             <br>"))  
})
```



# DEMO

## Putting it all together...

`movies_07.R`

(also notice the HTML tags,  
added for visual separation, in the mainPanel)



# WHEN TO USE REACTIVES

- ▶ By using a reactive expression for the subsetting data frame, we were able to get away with subsetting once and then using the result twice
- ▶ In general, reactive conductors let you
  - ▶ not repeat yourself (i.e. avoid copy-and-paste code) which is a maintenance boon)
  - ▶ decompose large, complex (code-wise, not necessarily CPU-wise) calculations into smaller pieces to make them more understandable
- ▶ These benefits are similar to what happens when you decompose a large complex R script into a series of small functions that build on each other

# EXERCISE



- ▶ For consistency, in `movies_07.R`, there should be at least one more spot on the app where the new `movies_subset` dataset should be used, instead of the full `movies` dataset
  - ▶ *Hint:* Does the data table match the plotted data?
- ▶ Find and fix
- ▶ Run the app to confirm your fix is working
- ▶ Compare your code / output with the person sitting next to / nearby you

3<sub>m</sub> 00<sub>s</sub>




# SOLUTION

Solution to the previous exercise

`movies_08.R`

# EXERCISE



Suppose we want to plot only a random sample of movies, of size determined by the user. What is wrong with the following?

**ui:**

```
# Select sample size
numericInput("n_samp", "Sample size:", min = 1, max = nrow(movies), value = nrow(movies))
```

**server:**

```
# Create a new data frame that is a sample of n_samp observations from movies
movies_sample <- sample_n(movies, input$n_samp)

# Plot the sampled movies
output$scatterplot <- ggplot(data = movies_sample,
                             aes_string(x = input$x, y = input$y, color = input$z)) +
  geom_point(...)
```

1<sub>m</sub> 00<sub>s</sub>



# SOLUTION

Solution can also be found in movies\_09.R.

Note that output\$n and output\$datatable are also updated in the script.

ui:

```
# select sample size
numericInput("n_samp", "Sample size:", min = 1, max = nrow(movies), value = 50)
```

server:

```
# Create a new data frame that is n_samp observations from selected type movies
movies_sample <- reactive({
  req(input$n_samp) # ensure availability of value before proceeding
  sample_n(movies_subset(), input$n_samp)
})

# Plot the sampled movies
output$scatterplot <- renderPlot({
  ggplot(data = movies_sample(), aes_string(x = input$x, y = input$y, color = input$z)) +
    geom_point(...)
})
```

# Implementation

# IMPLEMENTATION OF REACTIVE OBJECTS

- ▶ **Reactive values – reactiveValues():** implementation of reactive sources
  - ▶ e.g. input object is a reactive value, which looks like a list, and contains many individual reactive values that are set by input from the web browser
- ▶ **Reactive expressions – reactive():** implementation of reactive conductors
  - ▶ Can access reactive values or other reactive expressions, and they return a value
  - ▶ Useful for caching the results of any procedure that happens in response to user input
  - ▶ e.g. reactive data frame subsets we created earlier
- ▶ **Observers – observe():** implementation of reactive endpoints
  - ▶ Can access reactive sources and reactive expressions, but they don't return a value; they are used for their **side effects**
  - ▶ e.g. output object is a reactive observer, which also looks like a list, and contains many individual reactive observers that are created by using reactive values and expressions in reactive functions

# REACTIVITY ONLY WORKS WITH REACTIVE OBJECTS

- ▶ Only reactive primitives (like the ones on the previous slide) and things built on top of reactive primitives, will elicit reactivity. In particular, do NOT expect changes to "normal" variables to cause reactivity.

```
x <- 10  
y <- reactive({ x })  
  
# Much later...  
x <- 20
```



# REACTIVE VALUES

- ▶ Like an R environment object (or what other languages call a hash table or dictionary), but reactive
- ▶ Like the input object, but not read-only

```
rv <- reactiveValues(x = 10)  
rv$x <- 20  
rv$y <- mtcars
```



# REACTIVE VALUES

- ▶ Reading a value from a `reactiveValues` object is a reactive operation.
  - ▶ The act of reading it means the current reactive conductor or endpoint will be notified the next time the value changes.
- ▶ Maybe surprisingly, setting/updating a value on a `reactiveValues` object is *not* in itself a reactive operation, meaning no relationship is established between the current reactive conductor or endpoint (if any!) and the `reactiveValues` object.

# Observers and side effects



# EXERCISE

Suppose we want the user to provide a title for the plot. What is wrong with the following, and how would you fix it? See `movies_10.R`.

**ui:**


```
textInput(inputId = "plot_title",  
          label = "Plot title",  
          placeholder = "Enter text to be used as plot title"),
```

**server:**

```
output$pretty_plot_title <- toTitleCase(input$plot_title)  
output$scatterplot <- renderPlot({  
  ggplot(data = movies_sample(), aes_string(x = input$x, y = input$y, color = input$z)) +  
    geom_point(alpha = input$alpha, size = input$size) +  
    labs(title = output$pretty_plot_title)  
})
```

3<sub>m</sub> 00<sub>s</sub>

# SOLUTION



Observers do not have dependencies, use reactivities instead.  
Solution can also be found in movies\_11.R.

**ui:**

```
textInput(inputId = "plot_title",  
          label = "Plot title",  
          placeholder = "Enter text to be used as plot title"),
```

**server:**

```
pretty_plot_title <- reactive({ toTitleCase(input$plot_title) })  
output$scatterplot <- renderPlot({  
  ggplot(data = movies_sample(), aes_string(x = input$x, y = input$y, color = input$z)) +  
    geom_point(alpha = input$alpha, size = input$size) +  
    labs(title = pretty_plot_title())  
})
```

# REACTIVE EXPRESSIONS VS. OBSERVERS

- ▶ Similarities: Both store expressions that can be executed
- ▶ Differences:
  - ▶ Reactive expressions return values, but observers don't
  - ▶ Observers (and endpoints in general) *eagerly* respond to reactivities, but reactive expressions (and conductors in general) do not
  - ▶ Reactive expressions must not have *side effects*, while observers are *only* useful for their side effects





We cheated earlier, let's make it right with an observer!

See `movies_12.R`.

```
server <- function(input, output, session) {  
  ...  
  # Update the maximum allowed n_samp for selected type movies  
  observe({  
    updateNumericInput(session, inputId = "n_samp",  
                        value = min(50, nrow(movies_subset())))  
  })  
  ...  
}
```

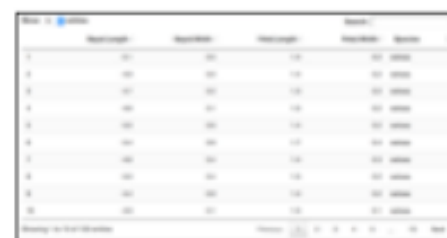
Render  
functions

# RENDER FUNCTIONS

```
render*({ [code_chunk] })
```

- ▶ Provide a code chunk that describes how an output should be populated
- ▶ The output will update in response to changes in any reactive values or reactive expressions that are used in the code chunk

# LIST OF REACTIVE FUNCTIONS



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

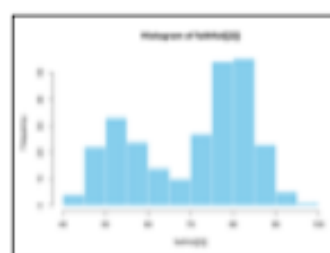


**dataTableOutput**(outputId, icon, ...)



**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)

'data.frame': 3 obs. of 2 variables:  
 \$ Sepal.Length: num 5.1 4.9 4.7  
 \$ Sepal.Width : num 3.5 3 3.2

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

**verbatimTextOutput**(outputId)

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.10	3.50	1.40	0.20	setosa
2	4.90	3.00	1.40	0.20	setosa
3	4.70	3.20	1.30	0.20	setosa
4	4.60	3.10	1.30	0.20	setosa
5	5.00	3.40	1.40	0.20	setosa
6	5.40	3.60	1.70	0.40	setosa

**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, ...)

# RECAP

```
render*({ [code_chunk] })
```

- ▶ These functions make objects to display
- ▶ Results should always be saved to output\$
- ▶ They make an observer object that has a block of code associated with it
- ▶ The object will rerun the entire code block to update itself whenever it is invalidated






# EXERCISE

- ▶ Run the app in movies\_12.R.
- ▶ Try entering a few different plot titles and observe that the plot title updates however the sampled data that is being plotted does not.
- ▶ Given that the renderPlot() function reruns each time input\$plot\_title changes, why does the sample stay the same?

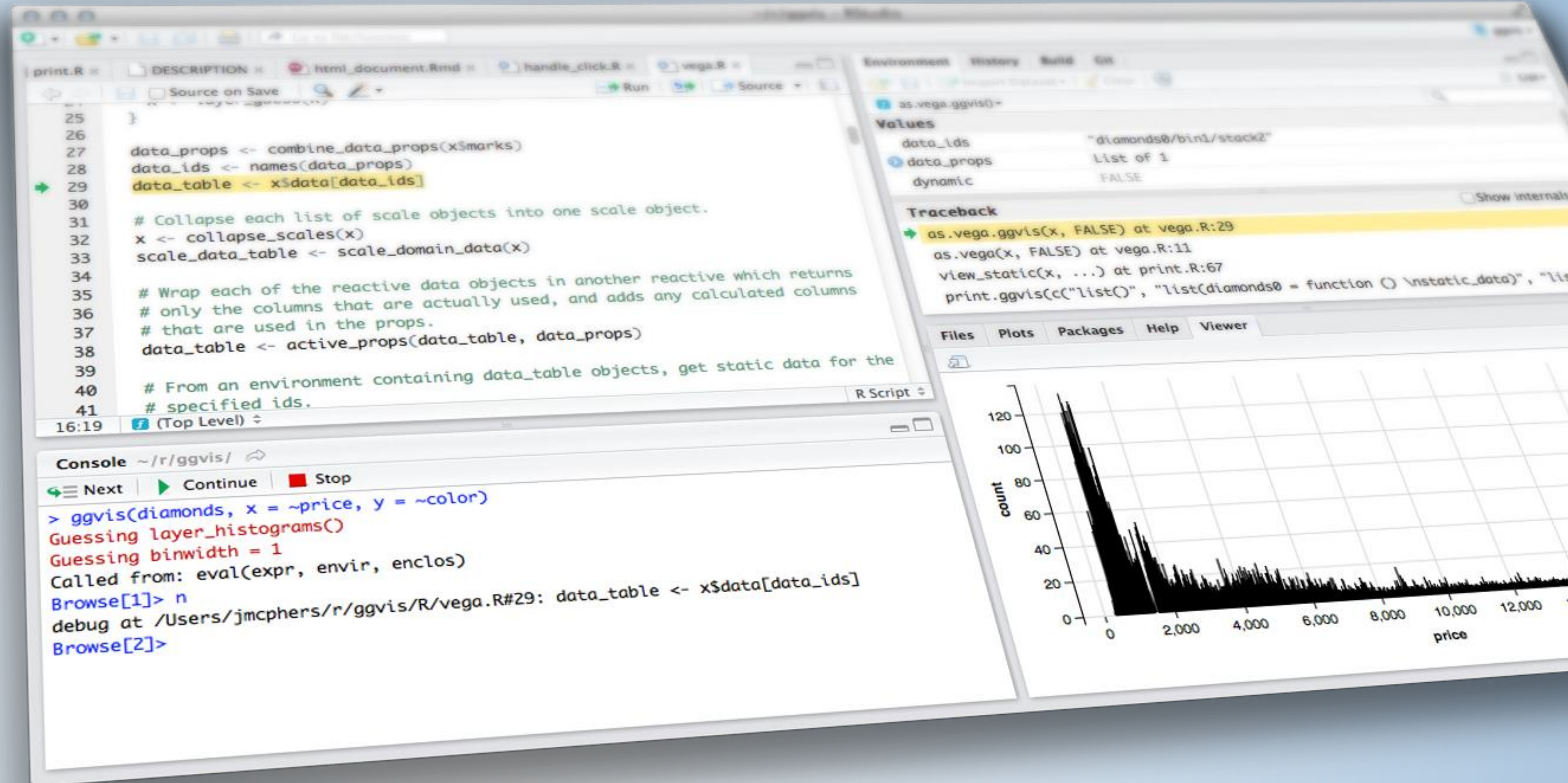
1<sub>m</sub> 00<sub>s</sub>

# SOLUTION



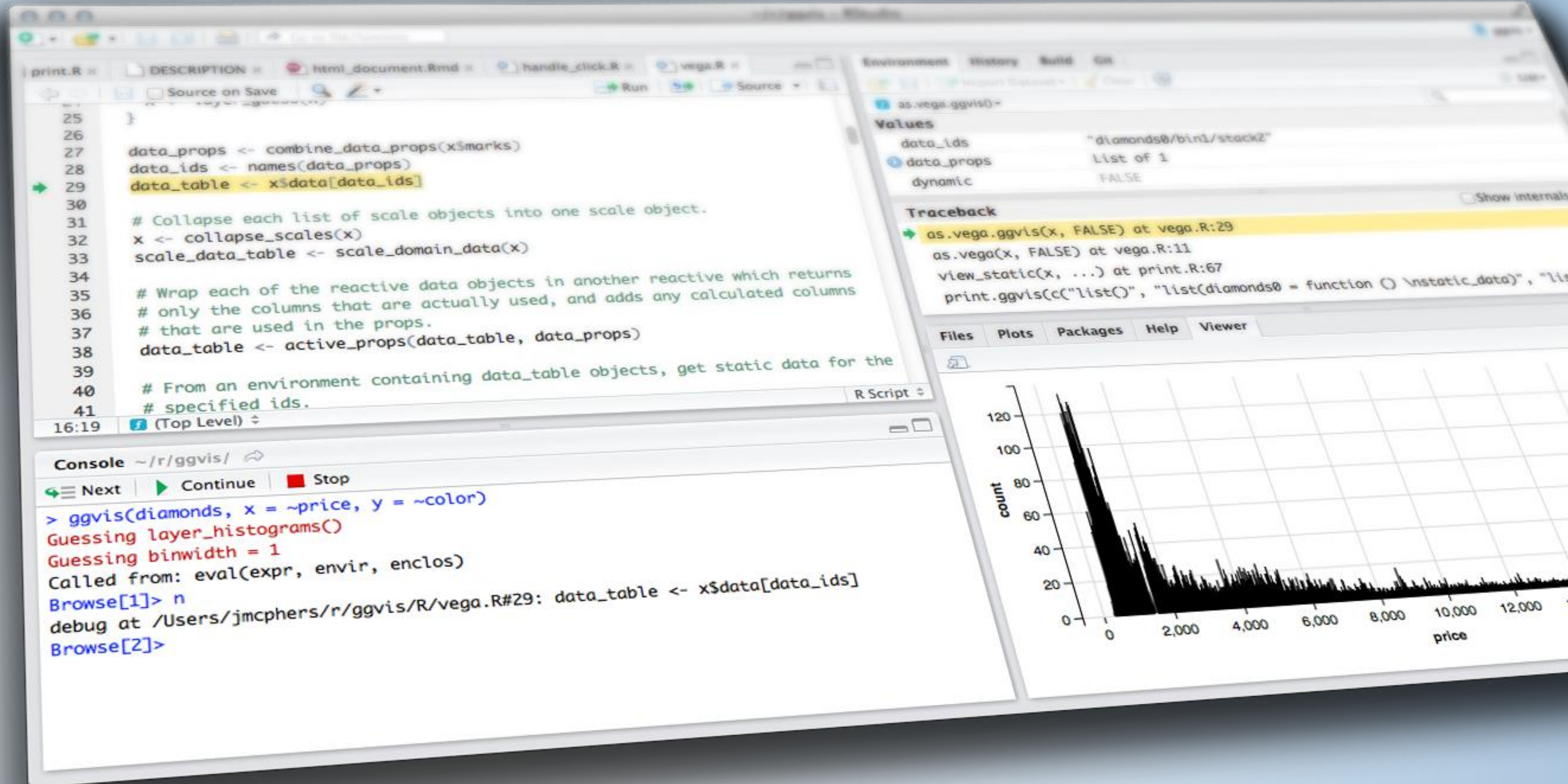
Because the data frame that is used in the plot is defined as a reactive expression with a code chunk that does not depend on `input$plot_title`.





# CLASS BREAK





# UNDERSTANDING UI

Web application UI is ultimately HTML/CSS/JavaScript

Shiny allows R users write user interfaces using a simple, familiar-looking API...

...but no limits for advanced users

# Ladder of progression



# LADDER OF UI PROGRESSION

Step 1. Shiny built-in inputs/outputs and layouts (sidebarLayout, navbarPage, tabsetPanel)

Step 2. Use functions from external packages (shinythemes, shinydashboard, shinybs)

Step 3. Use tag objects, write UI functions

**Our focus today**

Step 4. Author HTML templates

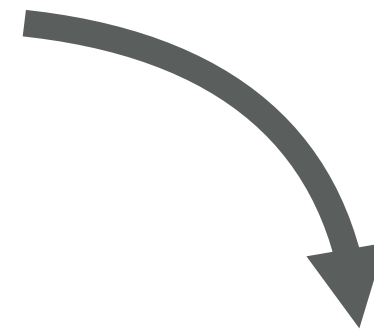
Step 5. Create custom inputs/outputs, wrap existing CSS/JS libraries and frameworks

High level  
view

# MULTIPLE LEVELS OF ABSTRACTION

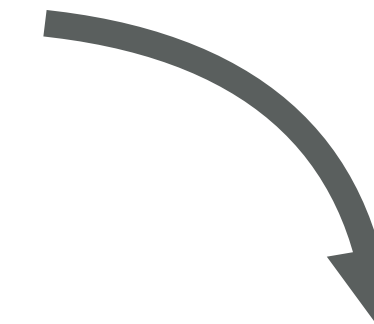
**High-level funcs**

`fluidRow(...)`



**htmltools tags**

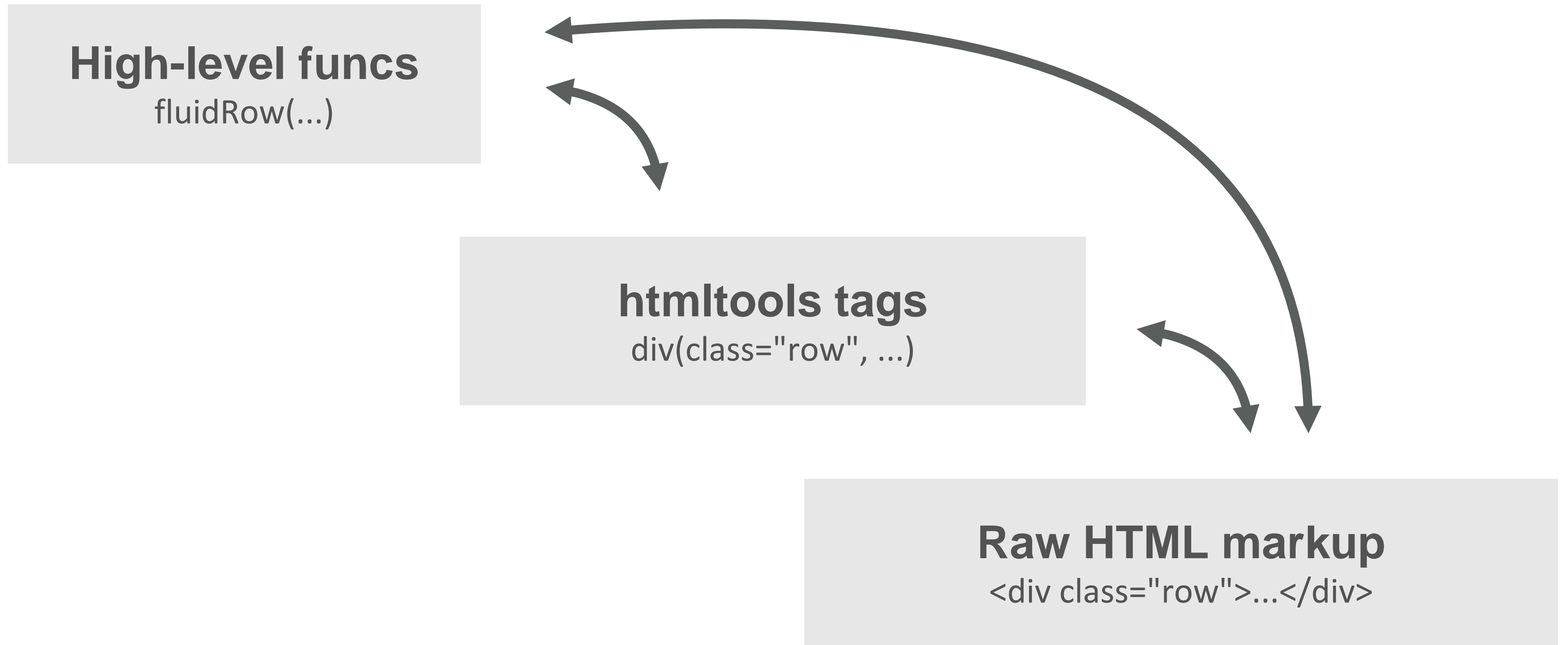
`div(class="row", ...)`

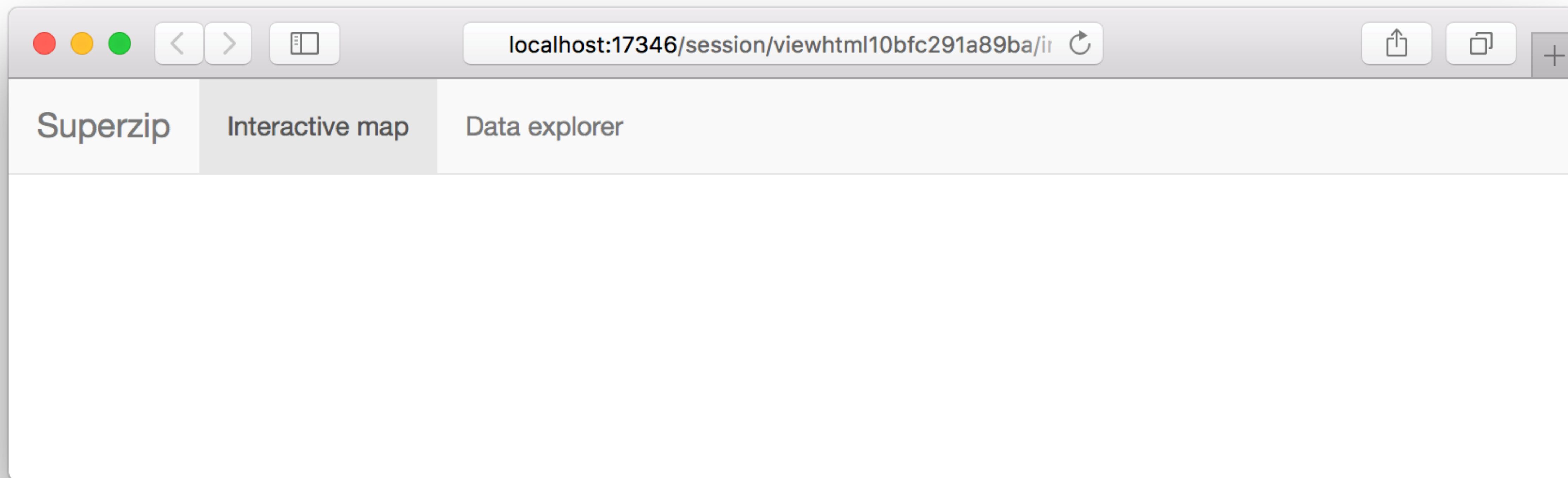


**Raw HTML markup**

`<div class="row">...</div>`

# MIX AND MATCH FREELY





# RAW HTML

## Pros

Can do anything that's possible in a web page

Comfortable for designers, web developers

## Cons

Unfamiliar for many R users

Potentially lots of HTML needed for conceptually simple tasks

CSS/JavaScript dependencies must be handled manually

```
<nav class="navbar navbar-default navbar-static-top" role="navigation">
  <div class="container">
    <div class="navbar-header">
      <span class="navbar-brand">Superzip</span>
    </div>
    <ul class="nav navbar-nav shiny-tab-input" id="nav">
      <li class="active">
        <a href="#tab-5158-1" data-toggle="tab" data-value="1">Home</a>
      </li>
      <li>
        <a href="#tab-5158-2" data-toggle="tab" data-value="2">About</a>
      </li>
      <li>
        <a href="#tab-5158-3" data-toggle="tab" data-value="3">Contact</a>
      </li>
    </ul>
  </div>
</nav>
<div class="container-fluid">
  <div class="tab-content">
    <div class="tab-pane active" data-value="Interactive map">
      <div class="outer">
        <div id="map" style="width:100%; height:100%; border:1px solid #ccc;">
          <div class="panel panel-default draggable" id="content">
            <div class="map">
              <img alt="A map showing the location of Superzip." data-bbox="100 100 900 900"/>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```



# HTMLTOOLS OBJECTS

## HTML-generating R functions

### Pros

- All the power of HTML, but looks like R
- Automated CSS/JS dependency handling
- More composable, programmable than HTML

### Cons

- Easy to misplace commas
- Almost as verbose as raw HTML

```
nav(class="navbar navbar-default navbar-static-top", role="navigation",
  div(class="container",
    div(class="navbar-header",
      span(class="navbar-brand", "Superzip")
    ),
    ul(class="nav navbar-nav shiny-tab-input", id="nav",
      li(class="active",
        a(href="#tab-5158-1", `data-toggle`="tab", `data-v`="1")
      ),
      li(
        a(href="#tab-5158-2", `data-toggle`="tab", `data-v`="2")
      ),
      li(
        a(href="#tab-5158-3", `data-toggle`="tab")
      )
    )
  )
)
```

# HIGH LEVEL FUNCTIONS

Functions that return htmltools objects

## Pros

- Less code, clearer intent
- Anyone can make their own

## Cons

- Still have to watch out for commas
- Less flexible

```
navbarPage("Superzip", id = "nav",  
  tabPanel("Interactive map", ...),  
  tabPanel("Data explorer", ...)  
)
```

# Using Shiny built-ins

# SHINY UI BUILT-INS

**Bootstrap grid framework** – fluidPage, fixedPage, fluidRow, column

**Containers** – wellPanel, absolutePanel, fixedPanel

**Navigation panels** – tabsetPanel, navlistPanel, navbarPage

**Fill layouts** (Shiny 0.13+) – fillPage, fillRow, fillCol

**Modals and notifications** (Shiny 0.14+) – showModal, modalDialog

# BOOTSTRAP GRID FRAMEWORK

Every page has 12 invisible columns

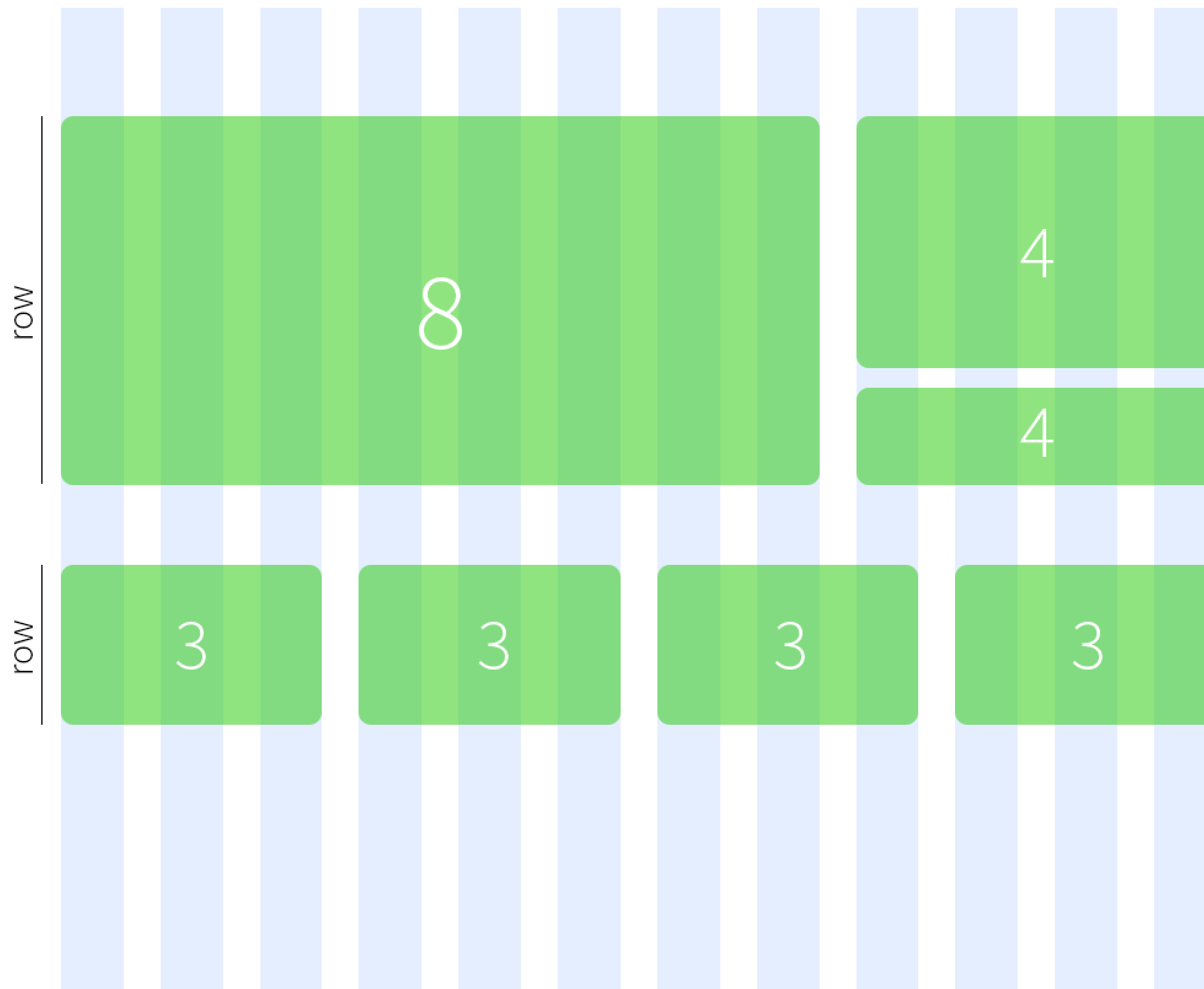
Each column of content must span an integral number of columns

Simple R API for implementing Bootstrap grid

`fluidPage(...)` wraps the entire page

`fluidRow(...)` wraps each row's column

`column(width, ...)` wraps each column's content





# FLUID PAGE

```
ui <- fluidPage(  
  fluidRow(  
    column(8, item1),  
    column(4, item2, item3),  
  ),  
  fluidRow(  
    column(3, item4),  
    column(3, item5),  
    column(3, item6),  
    column(3, item7)  
  )  
)
```

# EXERCISE



- ▶ Modify `ui_01.R` to display the two outputs next to each other (instead of above and below)
- ▶ Assign the left output to be 5 columns wide, and the right output to be 7 columns wide
- ▶ See what happens as you change the width of the browser window

3<sub>m</sub> 00<sub>s</sub>



# SOLUTION

Solution to the previous exercise

ui\_02.R



# DEMO

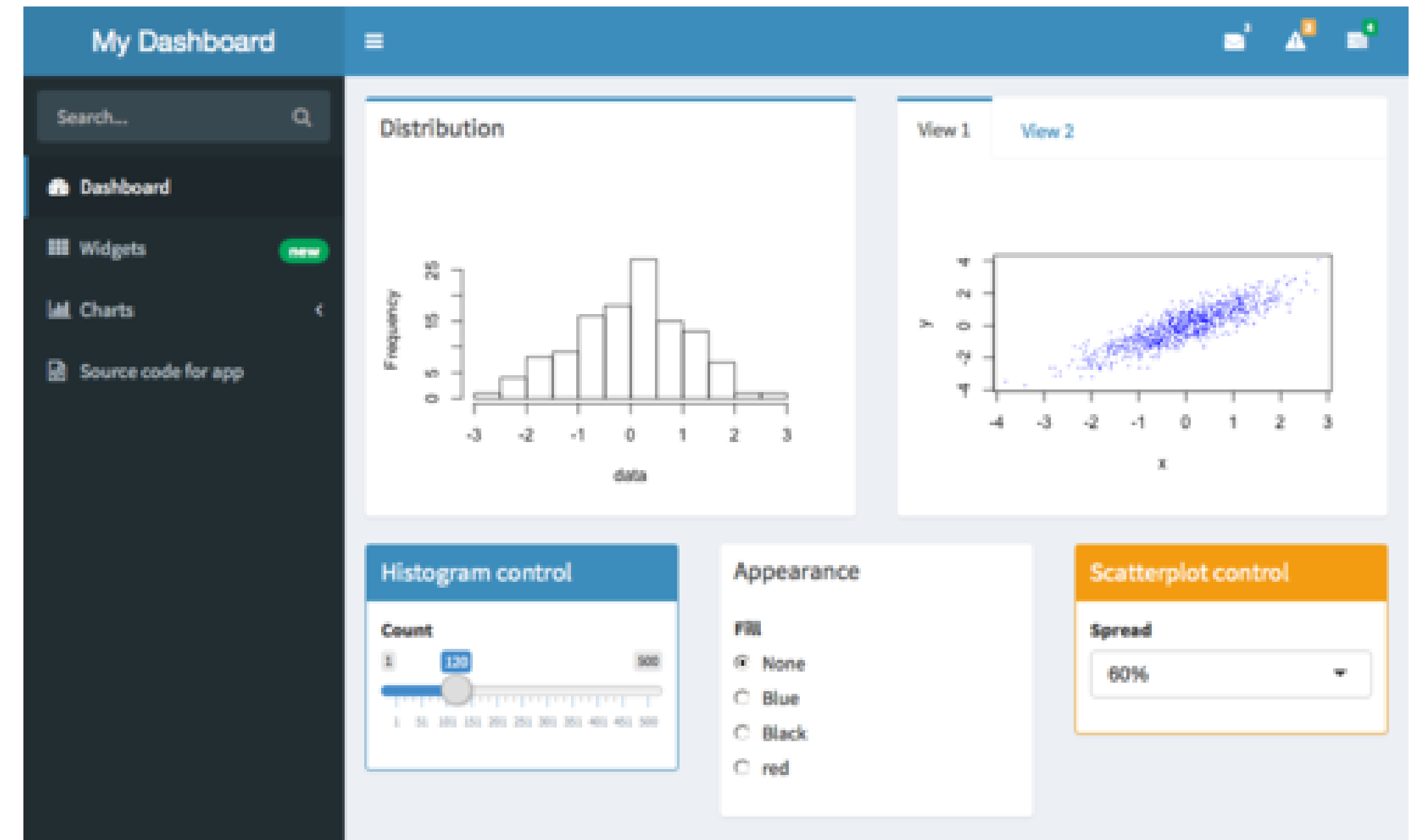
Layouts

<https://shiny.rstudio.com/gallery/>

Using external  
packages

# EXTERNAL PACKAGES

shinydashboard





# EXTERNAL PACKAGES

shinydashboard

shinythemes

The image displays a collage of Shiny dashboard examples. The primary example is a 'Flatly' theme dashboard with a dark blue header containing tabs: 'Flatly', 'Navbar 1', 'Plot', and 'Table'. Below the header, there are three sub-tabs: 'Tab 1', 'Tab 2' (highlighted in green), and 'Tab 3'. The 'Table' tab contains a table with two columns, 'speed' and 'dist', and four rows of data. Below the table is a 'Verbatim text output' box showing the text 'general, 30, NULL'. Further down are five headers labeled 'Header 1' through 'Header 5'. Overlaid on this are two smaller dashboard snippets. One snippet, titled 'United', shows a 'File input' with 'Browse...' and 'No file se' buttons, a 'Text input' with the value 'general', a 'Slider input' with a range from 1 to 100 and a value of 30, and a 'Default actionButton' with a 'Search' button and an 'Action button' with a CSS class. Another snippet, titled 'Darkly', shows a similar set of controls but with a dark background and lighter text.

Flatly Navbar 1 Plot Table

Tab 1 Tab 2 Tab 3

Table

speed	dist
4.00	2.00
4.00	10.00
7.00	4.00
7.00	22.00

Verbatim text output

```
general, 30, NULL
```

Header 1  
Header 2  
Header 3  
Header 4  
Header 5

Header 4  
Header 5

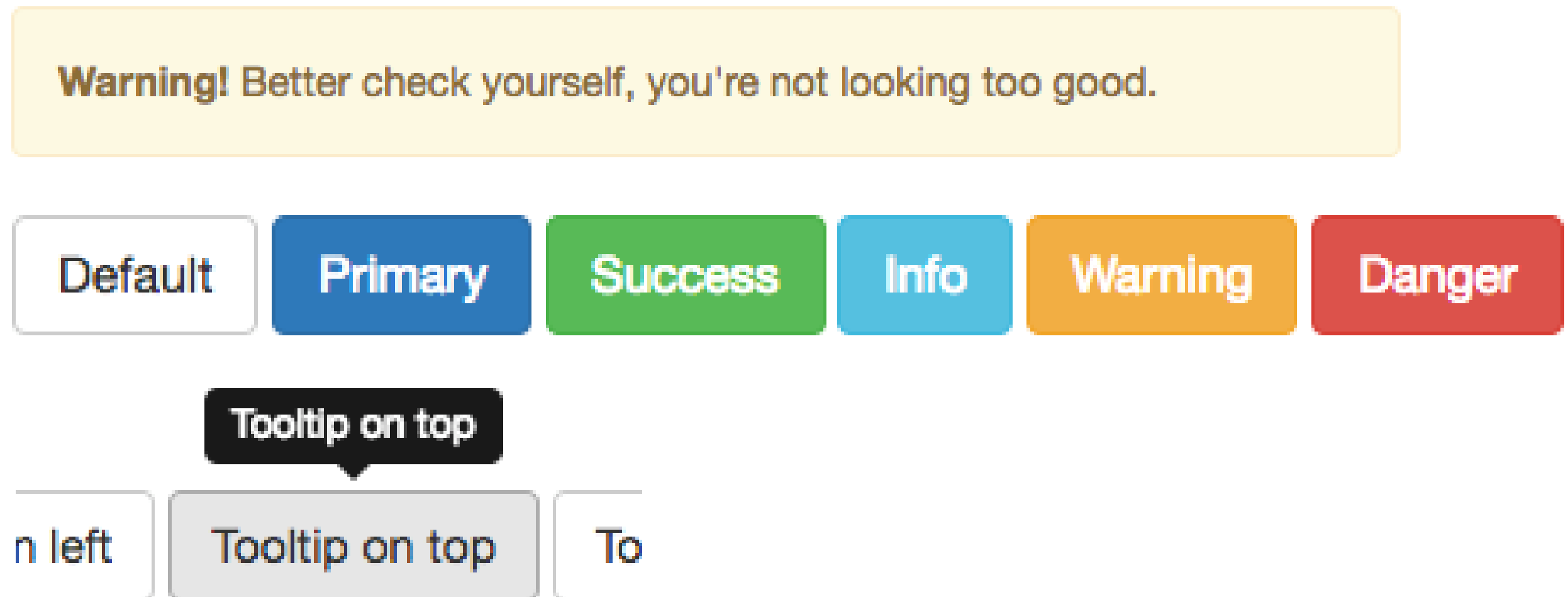
Header 3

# EXTERNAL PACKAGES

shinydashboard

shinythemes

shinyBS (@ebailey78)



# EXTERNAL PACKAGES

shinydashboard

shinythemes

shinyBS (@ebailey78)

shinytoastr (@gaborcsardi)



Cannot access database



Clock mismatch detected



Using the test database



Database updated

# EXTERNAL PACKAGES

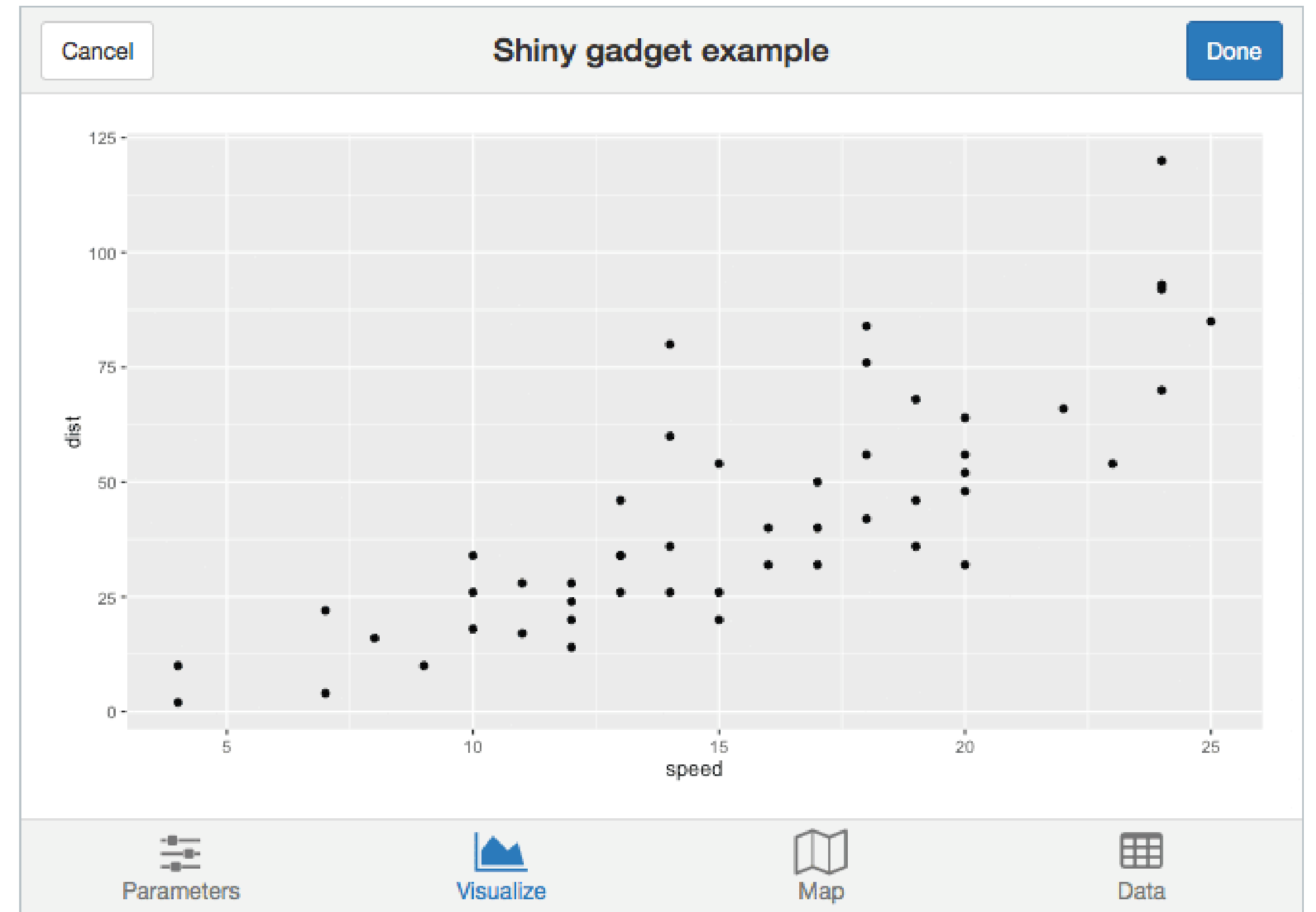
shinydashboard

shinythemes

shinyBS (@ebailey78)

shinytoastr (@gaborcsardi)

miniUI (for mobile devices or Shiny Gadgets)



# EXTERNAL PACKAGES

- bslib: <https://github.com/rstudio/bslib>
- Demo: <https://testing-apps.shinyapps.io/themer-demo/>

The image displays two side-by-side screenshots of the 'themer-demo' Shiny application, illustrating the 'Dark' and 'Light' themes.

**Left Screenshot (Dark Theme):**

- Theme customizer:** Shows 'Main colors' and 'Overall theme' set to 'Dark'.
- Input Widgets:** Includes a slider input (range 30-70), a selectize input (AL), a selectize input (multiple=T) (AZ, AK, CA), and a date input (2020-12-24).
- Console Output:** Shows the values bound to each input widget: sliderInput (int [1:2] 30 70), selectizeInput (chr "AL"), selectizeMultiInput (chr [1:3] "AZ" "AK" "CA"), dateInput (Date[1:1], format: "2020-12-24"), and dateRangeInput (Date[1:2], format: "2020-12-24" "2020-12-31").
- Action Buttons:** Primary (blue), Secondary (default) (grey), Success (green), Info (blue), Warning (yellow).

**Right Screenshot (Light Theme):**

- Theme customizer:** Shows 'Main colors' and 'Overall theme' set to 'Light'.
- Input Widgets:** Includes a slider input (range 30-70), a selectize input (AL), a selectize input (multiple=T) (AK, AR, AL), and a date input (2020-12-24).
- Console Output:** Shows the values bound to each input widget: sliderInput (int [1:2] 30 70), selectizeInput (chr "AL"), selectizeMultiInput (chr [1:3] "AK" "AR" "AL"), dateInput (Date[1:1], format: "..."), and dateRangeInput (Date[1:2], format: "...").
- Action Buttons:** Primary (blue), Secondary (default) (grey), Success (green), Info (blue), Warning (yellow), Danger (red), Dark (dark blue), Light (light blue).

# EXTERNAL PACKAGES

shinyjs  
(@daattali)

Perform many  
UI-related  
JavaScript  
operations  
from R

Function	Description
<code>show / hide / toggle</code>	Display or hide an element (optionally with an animation).
<code>hidden</code>	Initialize a Shiny tag as invisible (can be shown later with a call to <code>show</code> ).
<code>enable / disable / toggleState</code>	Enable or disable an input element, such as a button or a text input.
<code>disabled</code>	Initialize a Shiny input as disabled.
<code>reset</code>	Reset a Shiny input widget back to its original value.
<code>delay</code>	Execute R code (including any <code>shinyjs</code> functions) after a specified amount of time.
<code>alert</code>	Show a message to the
<code>html</code>	Change the text/HTML of an element.
<code>onclick</code>	Run R code when a specific element is clicked. Was originally developed with the sole purpose of running a <code>shinyjs</code> function when an element is clicked, though any R code can be used.
<code>onevent</code>	Similar to <code>onclick</code> , but can be used with many other events instead of click (for example, listen for a key press, mouse hover, etc).
<code>addClass / removeClass / toggleClass</code>	add or remove a CSS class from an element.
<code>runjs</code>	Run arbitrary JavaScript code.
<code>extendShinyjs</code>	Allows you to write your own JavaScript functions and use <code>shinyjs</code> to call them as if they were regular R code. More information is available in the section "Calling your own JavaScript functions from R" below.



# EXERCISE



- ▶ Modify movies\_12.R to use a Bootstrap theme
  - ▶ Use the "Live theme selector" feature in shinythemes in your own app
  - ▶ Once you've decided on a theme, remove the theme selector and apply your chosen theme permanently
- ▶ See shinythemes instructions at:  
<https://rstudio.github.io/shinythemes/>

5<sub>m</sub> 00<sub>s</sub>



# SOLUTION

Solution to the previous exercise

`movies_13.R`

Using htmltools  
tag objects

# AN API FOR COMPOSING HTML

When Shiny was born, it came with a sub-package for composing HTML

These functions were so useful, we extracted them out into a separate package: `htmltools`

Now used by R Markdown and `htmlwidgets` as well

# HTML BASICS

```
<a href="https://www.posit.co">Posit</a>
```



Posit



# HTML BASICS

```
<a href="https://www.posit.co">Posit</a>
```

End tag

Start tag

Child content

Attribute name

# ANATOMY OF A TAG

```
<a href="https://posit.co/">Posit</a>
```

Tag name

Attribute value

Creates an **anchor** whose  
**hyperlink reference** is the URL  
`https://www.posit.co`

# ANATOMY OF A TAG

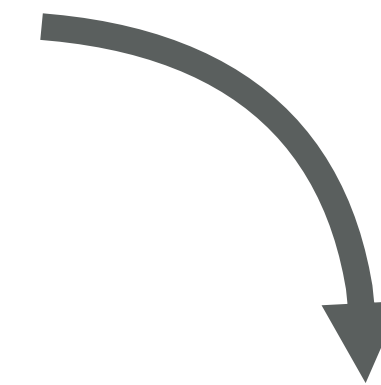
Text can contain tags

Tags can optionally contain text and/or other tags

Each start tag can have zero or more attributes

# HTML TO BROWSER UI

```
<div class="panel panel-default">  
  <div class="panel-heading">  
    <h3 class="panel-title">Panel title</h3>  
  </div>  
  <div class="panel-body">  
    Panel content  
  </div>  
</div>
```



**Panel title**

Panel content

# LOOKS LIKE R, MEANS HTML

```
<div class="panel panel-default">  
  <div class="panel-heading">  
    <h3 class="panel-title">  
      Panel title  
    </h3>  
  </div>  
  <div class="panel-body">  
    Panel content  
  </div>  
</div>
```

```
div(class="panel panel-default",  
  div(class="panel-heading",  
    h3(class="panel-title",  
      "Panel title",  
    )  
  ),  
  div(class="panel-body",  
    "Panel content"  
  )  
)
```



# USING TAG FUNCTIONS

Many common tags are exported as functions by `htmltools` and `shiny` (`p`, `h1-h6`, `a`, `br`, `div`, `span`, `img`)

All other tags can be accessed via the `tags` object. E.g., `<li>Item 1</li>` → `tags$li("Item 1")`

If you have lots of HTML to write, you can use the `withTags` function—it makes the `tags$` prefix optional.

```
withTags(  
  ul(  
    li("Item 1"), li("Item 2")  
  )  
)
```

# USING TAG FUNCTIONS

All tag functions behave the same way

Call the function to create a tag object

Named arguments become attributes

Unnamed arguments become children

# TAG ATTRIBUTES

Any valid HTML attribute name can be used (use quotes if the name has dashes, e.g. "data-toggle"="dropdown")

Valid tag attribute values are:

NULL (omit the attribute)

NA (the attribute should be included with no value)

Single-element character vector (or something to be coerced to character)

```
tags$input(type = "checkbox",  
  disabled = if (disabled) NA # else NULL  
)
```

# TAG CHILDREN

Valid tag children are:

Tag objects

Single-element character vectors (treated as text)

NULL (silently ignored)

Raw HTML (see ?htmltools::HTML)

Lists of valid tag children (recursive!)

# USING TAGS

Tags are made using normal R functions that take normal parameters and return normal values! You can do R-like things to them:

```
tags$ul(lapply(1:10, tags$li))
```

Print tag objects at the console to see their HTML source

Call `print(x, browse = TRUE)` to see their rendered view instead

Use `htmltools::browsable()` to make an object show its rendered view when printed, by default

If your top-level object is a list, you'll need to wrap in `tagList(...)` to get the right behavior at the console (or in an R Markdown doc)

# EXERCISE



- ▶ Open `ui_03.R`.
- ▶ Replace `includeHTML("youtube_thumbnail.html")` with the equivalent `htmltools` tag objects.
  - ▶ Hint: Take a look inside `youtube_thumbnail.html`.
- ▶ If you get that working, take the next step and define an R function that takes a YouTube URL, a title, and a description, and returns a thumbnail frame like the one you created.

5<sub>m</sub> 00<sub>s</sub>





# SOLUTION

## Solutions to the previous exercise

ui\_04.R

ui\_05.R

Using

*raw* HTML

# USING RAW HTML

Incorporate tiny amounts of HTML using inline string literals wrapped in `HTML()`

```
div(HTML("This is <strong>HTML</strong>"))
```

For chunks of (static) HTML, use `includeHTML` (or similar `includeCSS`, `includeScript`)

```
div(includeHTML("file.html"))
```

Or go the other way, with the [HTML Templates](#) feature: start with HTML, and embed R expressions that yield tag objects

Using

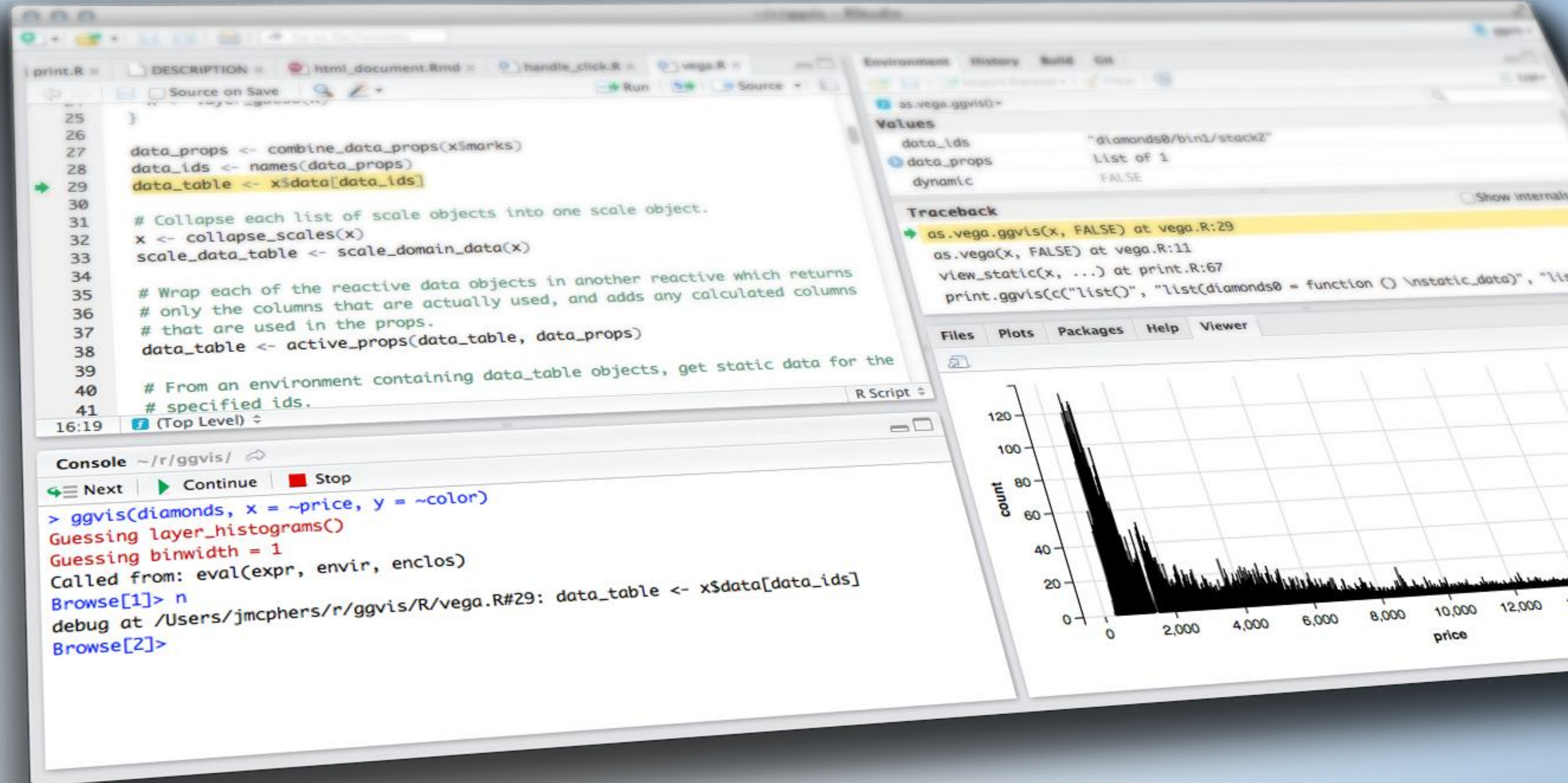
Shiny UI Editor



# DEMO

## Shiny UI Live Demo





# REACTIVE PROGRAMMING & UNDERSTANDING UI



# EXERCISE




- ▶ Create the Repo for your homework assignment
  - ▶ Click “New” on the course [GitHub page](#)
    - ▶ Name the repository your hw1 + your Andrew ID
      - ▶ ie: hw1-gla
    - ▶ Click “Create repository”
      - ▶ Do not create a .gitignore file
      - ▶ Clone the repo to your computer
- ▶ In RStudio create a new project in an existing directory
  - ▶ Select the cloned repository
    - ▶ If you’ve already started your homework simply move all your work into this repository

3<sub>m</sub> 00<sub>s</sub>

# EXERCISE



- ▶ Publishing a Shiny App
  - ▶ Sign up on <https://www.shinyapps.io/>
    - ▶ Go to the tokens page
      - ▶ Copy your tokens with secret to clip board
  - ▶ In Rstudio in any of the movies apps, go to the publish button 
    - ▶ Follow the steps to add your shiny apps io token.
  - ▶ Publish the app

3<sub>m</sub> 00<sub>s</sub>

# HOMework



## Homework 1