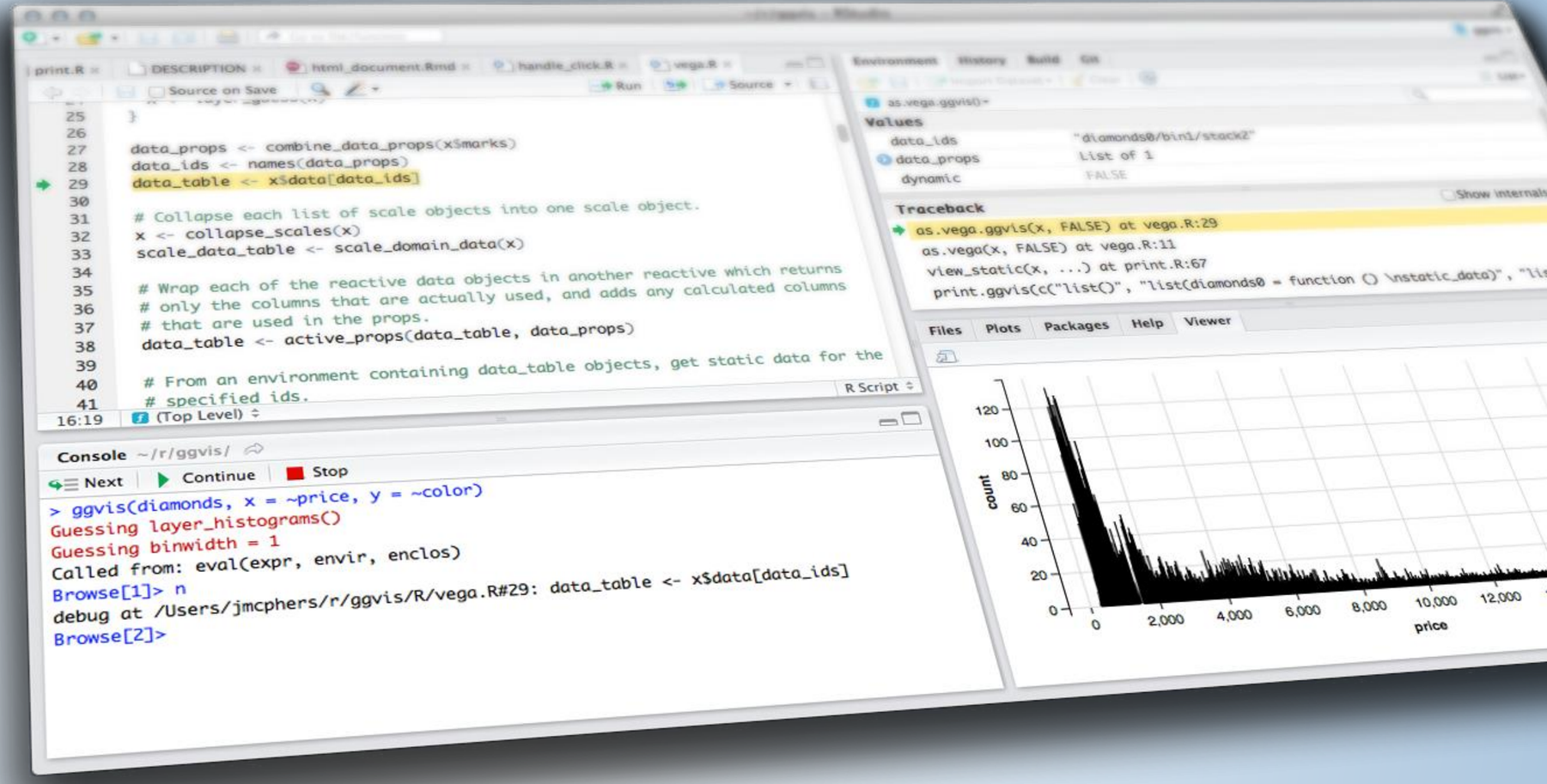


REACTIVE PROGRAMMING & DASHBOARDS

Shiny from posit

OUTLINE

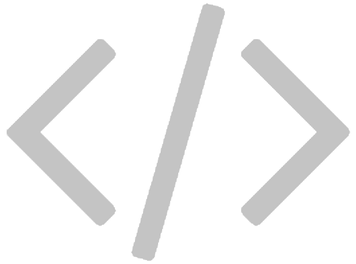
- Reactive Programming Part 2
  - Stop - trigger - delay
    - isolate()
    - observeEvent()
    - eventReactive()
  - Scheduling
    - Schedule with invalidateLater()
    - Monitor with reactivePoll()
    - reactiveFileReader()
  - Reactivity best practices
- Dashboards
  - What is in a dashboard?
  - Server
    - reactiveFileReader
    - reactivePoll
  - UI
    - Static vs. dynamic dashboards
    - Shiny pre-rendered
  - shinydashboard

Stop - trigger - delay

Stop with isolate()

ISOLATE

- Use isolate() to wrap an expression whose reactivity should be suppressed (i.e. the currently executing reactive expression/observer/output *shouldn't* be notified when something changes).



DEMO

Only update plot title when other components of the plot are also updated. See movies_14.R.

server:

```
pretty_plot_title <- reactive({ toTitleCase(input$plot_title) })

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

Plot title will update when any of the other **input**s in this chunk change

Plot title will **not** update when input$plot_title changes

Trigger with observeEvent()

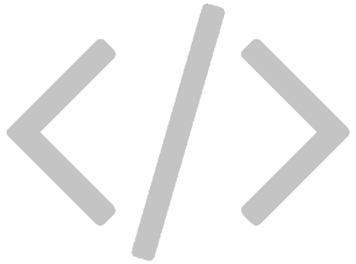
TRIGGERING A REACTION

- observeEvent() can be used to trigger a reaction
- It uses a different syntax

```
observeEvent(eventExpr, handlerExpr, …)
```

simple reactive value - **input$click**,
call to reactive expression - **df()**,
or complex expression inside **{}**

expression to call whenever
eventExpr is invalidated



DEMO

Write a CSV of the sampled data when action button is pressed. See movies_15.R.

ui:

```
actionButton(inputId = "write_csv", label = "Write CSV")
```

server:

```
observeEvent(eventExpr = input$write_csv,
             handlerExpr = {
               filename <- paste0("movies_", str_replace_all(Sys.time(), ":|\ ", "_"), ".csv")
               write.csv(movies_sample(), file = filename, row.names = FALSE)
               }
             )
```

ISOLATE VS. OBSERVEEVENT

- isolate() is used to stop a reaction
- while observeEvent() is used to perform an **action** in response to an event
  - Note: "recalculate a value" does not generally count as performing an action, we'll next discuss eventReactive() for that

Delay reactions with eventReactive()

OBSERVEEVENT VS. EVENTREACTIVE

- observeEvent() is to to perform an **action** in response to an event
- while eventReactive() is used to create a **calculated value** that only updates in response to an event
  - Just like a normal reactive expression except only invalidates in response to the given event.

```
observeEvent(eventExpr, valueExpr, …)
```

EXERCISE

- Change how the random sample is generated such that it is updated when the user clicks on an action button that says “Get new sample”.
- Use `movies_15.R` as the basis of the script and make the updates there.
- Run the app to ensure that the behavior is as described
- Compare your code / output with the person sitting next to / nearby you

5m 00s

SOLUTION

Solution can also be found in movies_16.R.

ui:

```
actionButton(inputId = "get_new_sample",
             label = "Get new sample")
```

server:

```
movies_sample <- eventReactive(eventExpr = input$get_new_sample,
                               valueExpr = {
                                 req(input$n_samp)
                                 sample_n(movies_subset(), inpu
                               },
                               ignoreNULL = FALSE
)
```

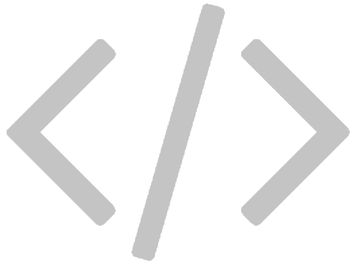
Initially perform the action/calculation and just let the user re-initiate it (like a "Recalculate" button)

Scheduling

Schedule with invalidateLater()

INVALIDATELATER

- If this is placed within an observer or reactive expression, that object will be invalidated (and re-execute) after the interval has passed
- The re-execution will reset the invalidation flag, so in a typical use case, the object will keep re-executing and waiting for the specified interval.
- It's possible to stop this cycle by adding conditional logic that prevents the `invalidateLater()` from being run.



DEMO

Tell the user how long they have been viewing your app for. See movies_17.R.

ui:

```
textOutput(outputId = "time_elapsed")
```

server:

```
# Calculate time difference between when app is first launched and now
beg <- reactive({ Sys.time() })
now <- reactive({ invalidateLater(millis = 1000); Sys.time() })
diff <- reactive({ round(difftime(now(), beg(), units = "secs")) })

# Print time viewing app
output$time_elapsed <- renderText({
  paste("You have been viewing this app for", diff(), "seconds.")
})
```

EXERCISE

- Change how the random sample is generated such that it is updated every 5 seconds
  - Don’t forget to remove now unused functionality for the action button to get a new sample
- Use `movies_17.R` as the basis of the script and make the updates there
- Run the app to ensure that the behavior is as described
- Compare your code / output with the person sitting next to / nearby you

5m 00s

SOLUTION

Solution can also be found in movies_18.R.

ui:

```
actionButton(inputId = "get_new_sample", label = "Get new sample")
```

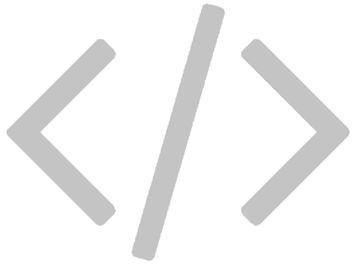
server:

```
# Get new sample every 5 seconds
movies_sample <- reactive({ invalidateLater(millis = 5000)
    req(input$n_samp)
    sample_n(movies_subset(), input$n_samp)
  })
```

Monitor with reactivePoll()

REACTIVEPOLL

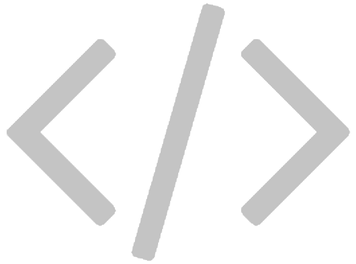
- reactivePoll() pairs a relatively cheap "check" function with a more expensive value retrieval function
  - **Check function:** is executed periodically and should always return a consistent value until the data changes
    - Note doesn't return `TRUE` or `FALSE`, instead it indicates change by returning a different value from the previous time it was called
  - **Value retrieval function:** is used to re-populate the data when the check function returns a different value
- Similar to invalidateLater(), but it's based on a change in a file as opposed to a periodic change



DEMO

Periodically check and report the names and dimensions of CSV files in the directory.

1. Write the check and value retrieval functions for `reactivePoll()`
2. Count and list CSV files in the directory every 5 seconds with `reactivePoll()`
3. Store CSV files in the directory as a data table in `output$csv_files`
4. Print `output$csv_files` in the UI, use tabs to reduce clutter



DEMO

1. Write the check and value retrieval functions for reactivePoll()

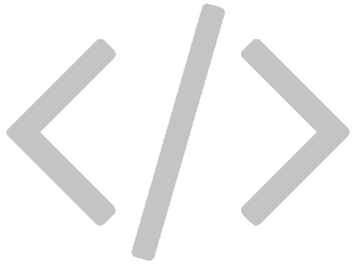
```
# Check function
count_files <- function(){ length(dir(pattern = "*.csv")) }

# Value retrieval function
list_files <- function(){
  files <- dir(pattern = "*.csv")
  if(length(files) == 0){ return( data.frame() ) }
  sapply(files, function(file) dim(read.csv(file))) %>%
    unlist() %>%
    t() %>%
    as.data.frame() %>%
    setNames(c("rows", "cols"))
}
```

There are many ways of doing this, don't focus too much on this code

DEMO

2. Count and list CSV files in the directory every 5 seconds with reactivePoll()

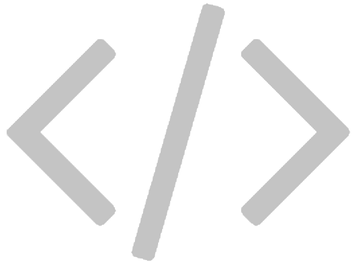
```
# Count and list CSV files in the directory every 5 seconds
csv_files <- reactivePoll(intervalMillis = 5000,
                          session,
                          checkFunc = count_files,
                          valueFunc = list_files)
```



DEMO

3. Store CSV files in the directory as a data table in output$csv_files

```
# Print CSV files in the directory
output$csv_files <- DT::renderDataTable(
    DT::datatable(data = csv_files(),
                  options = list(pageLength = 10),
                  rownames = TRUE)
```

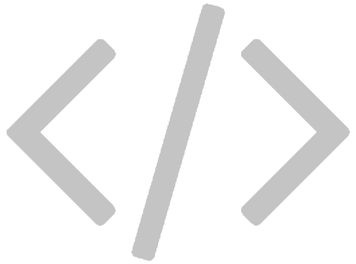


DEMO

4. Print output$csv_files in the UI, use tabs to reduce clutter

```
# Use tabs for the data tables to reduce clutter
tabsetPanel(
  # Show data table
  tabPanel("Plotted data", dataTableOutput(outputId = "moviestable")),

  # Show CSV files in directory
  tabPanel("Files in directory", dataTableOutput(outputId = "csv_files"))
)
```

This is new syntax we haven't seen before



DEMO

Putting it all together…

movies_19.R

See it in action: Change sample size, get new sample, write data to CSV, check out the “Files in directory” tab. Then, delete all CSV files in directory, and see the list update.

reactiveFileReader()

REACTIVEFILEREADER

- reactiveFileReader() works by periodically checking the file's last modified time
  - If the file has changed, it is re-read and any reactive dependents are invalidated
- Also similar to invalidateLater() but instead of periodic updates, updates are based on changes in a file

Reactivity best practices

EXERCISE

Is there something wrong with this? If so, what?

```
ui <- fluidPage(
  titlePanel("Add 2"),
  sidebarLayout(
    sidebarPanel( sliderInput("x", "Select x", min = 1, max = 50, value = 30) ),
    mainPanel( textOutput("x_updated") )
  )
)

server <- function(input, output) {
  add_2             <- function(x) { x + 2 }
  current_x         <- add_2(input$x)
  output$x_updated <- renderText({ current_x })
}
```

1 m 00 s

SOLUTION

Yup! See add_2.R.

```
ui <- fluidPage(
  titlePanel("Add 2"),
  sidebarLayout(
    sidebarPanel( sliderInput("x", "Select x", min = 1, max = 50, value = 30) ),
    mainPanel( textOutput("x_updated") )
  )
)

server <- function(input, output) {
  add_2             <- function(x) { x + 2 }
  current_x         <- reactive({ add_2(input$x) })
  output$x_updated <- renderText({ current_x() })
}
```

LESSON 1

Reactives are equivalent to no argument functions

Think about them as functions, think about them as variables that can depend on user input and other reactives

EXERCISE

observe() vs. reactive()

Which one should you use if you want to create an object that you can later use in a render function?

Which one if you want to update the minimum value of a slider input based on the choices a user makes in the app?

1 m 00 s

SOLUTION

observe() vs. reactive()

Which one should you use if you want to create an object that you can later use in a render function?

`reactive()`

Which one if you want to update the minimum value of a slider input based on the choices a user makes in the app?

`observe()`

LESSON 2

Reactives are for reactive values and expressions

Observers are for their side effects

EXERCISE

Is there something wrong with this? If so, what?

```
ui <- fluidPage(
 sidebarLayout(
    sidebarPanel(sliderInput("n", "Select n", min = 1,
                             max = 50, value = 30)),
    mainPanel(
      plotOutput("hist"),
      textOutput("med")
    )
  )
)
```

```
server <- function(input, output) {
  dist <- reactive({ rnorm(input$n) })
  output$hist <- renderPlot({
    hist(dist())
    med <- reactive({ median(dist()) })
    abline(v = med(), col = "red")
  })
  output$med <- renderText({
    paste("The median is", round(med(), 3))
  })
}
```

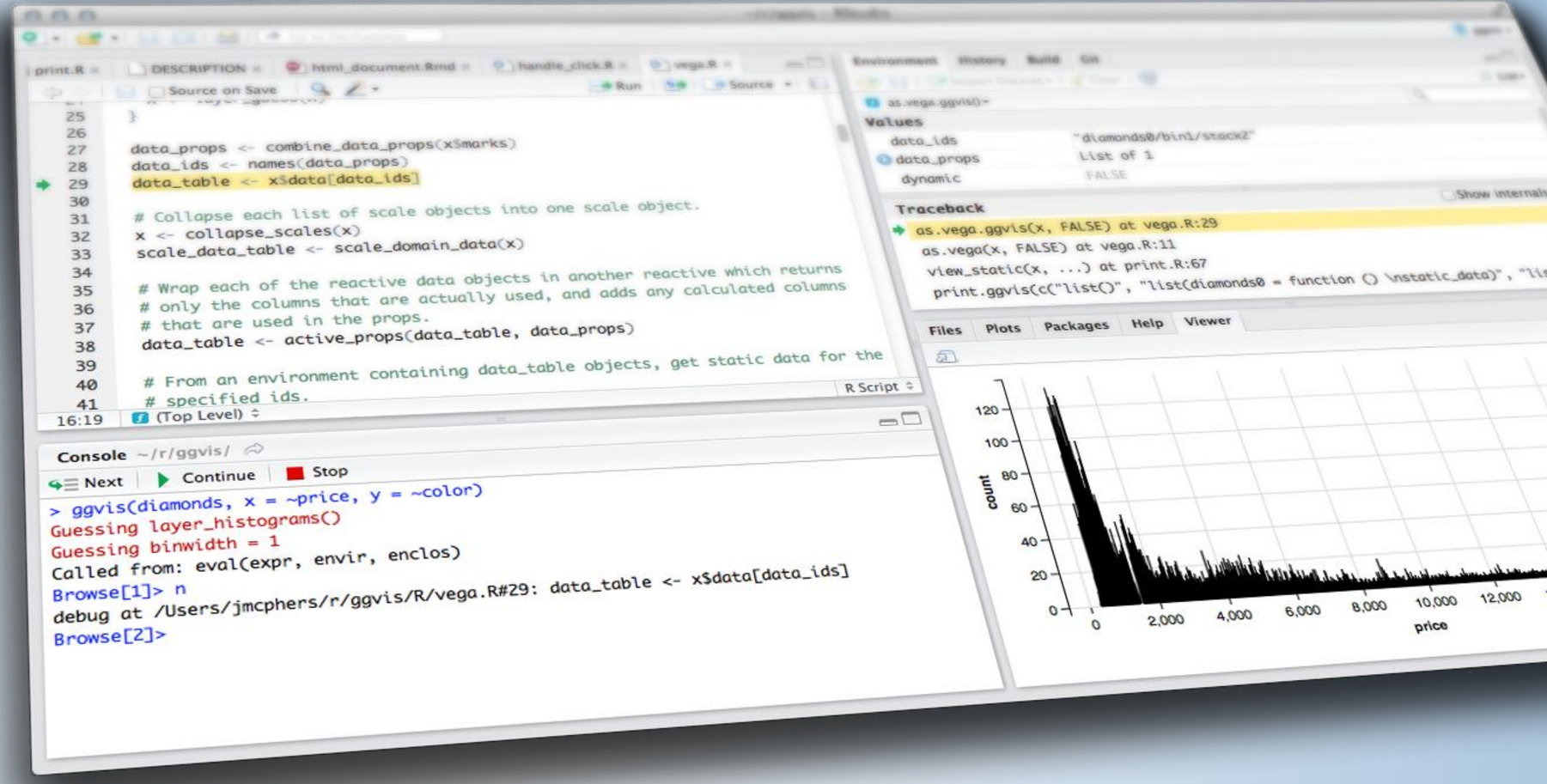
1 m 00 s

SOLUTION

Oh yeah! See hist_med.R.

```
ui <- fluidPage(
 sidebarLayout(
    sidebarPanel(sliderInput("n", "Select n", min = 1,
                             max = 50, value = 30)),
    mainPanel(
      plotOutput("hist"),
      textOutput("medtext")
    )
  )
)
```

```
server <- function(input, output) {
  dist <- reactive({ rnorm(input$n) })
  med <- reactive({ median(dist()) })
  output$hist <- renderPlot({
    hist(dist())
    abline(v = med(), col = "red")
  })
  output$medtext <- renderText({
    paste("The median is", round(med(), 3))
  })
}
```



DASHBOARDS

Shiny from posit

What is in a dashboard?

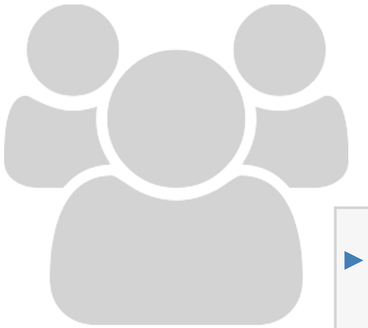
DASHBOARDS

- Automatically updating
  - Not just based on user gestures
  - But also when data source changes
- Many viewers looking at the same data
- May or may not be interactive

Server

MOTIVATION

- You have new data coming in — constantly, continuously, or on a schedule
- When new data comes in, it's automatically received, and transformed, aggregated, summarized, etc.
- May want to call attention to exceptional results



EXERCISE

- Why might this not be a good idea?

```
dataset <- reactive({
  result <- read.csv("data.csv")
  invalidateLater(5000)
  result
})

output$plot <- renderPlot({
  plot(dataset()) # or whatever
})
```

SOLUTION

Lots of overhead!

reactiveFileReader

REACTIVEFILEREADER

- Reads the given file ("data.csv") using the given function (read.csv)
- Periodically reads the last-modified time of the file
- If the timestamp changes, then (and only then) re-reads the file

Single file, on disk
(not database or web API)

```
dataset <- reactiveFileReader(
  intervalMillis = 1000,
  session = session,
  filePath = "data.csv",
  readFunc = read.csv
)

output$plot <- renderPlot({
  plot(dataset()) # or whatever
})
```

Must have data path as first argument

REACTIVEFILEREADER

```
dataset <- reactiveFileReader(
  intervalMillis = 1000,
  session = session,
  filePath = "data.csv",
  readFunc = read.csv,
  stringsAsFactors = FALSE
)

output$plot <- renderPlot({
  plot(dataset()) # or whatever
})
```

Add any named arguments

reactivePoll

REACTIVEPOLL

- `reactiveFileReader` is limited to files on disk. It doesn't work for non-file-based data sources like databases or web APIs
- `reactivePoll` is a generalization of `reactiveFileReader`
  - `checkFunc`: A function that can execute quickly, and merely determine if anything has changed
    - Should be fast as it will block the R process while it runs! The slower it is, the greater you should make the polling interval.
    - Should not return `TRUE` or `FALSE` for changed/unchanged. Instead, just return a value (like the timestamp, or the count); it's `reactivePoll`'s job, not yours, to keep track of whether that value is the same as the previous value or not.
  - `valueFunc`: A function with the (potentially expensive) logic for actually reading the data

EXERCISE

- When might we want to use reactivePoll on dashboards?

SOLUTION

When we are pulling from a database or Web API!

```
QueriedData <- reactivePoll(30000, session,
# This function checks the rows and when the rows are higher than previously, in those cases
it reads the table
  checkFunc = function(){
    # connect
    con <- poolCheckout(mysqldb)
    # Return the current numbers of rows in mysqltable
    rowcount <- dbGetQuery(con, "SHOW TABLE STATUS;") %>% filter(Name == "mysqltable") %>%
pull(Rows)
 # disconnect database
 poolReturn(con)
    },
  valueFunc = function() {
 # connect
    con <- poolCheckout(mysqldb)
    test_db <- dbReadTable(con, “mysqltable”)
   })
output$mytable  <- DT::renderDT({
    test_db <- QueriedData() %>% as.data.frame()
    DT::datatable(test_db)
})
```

Static vs. dynamic dashboards

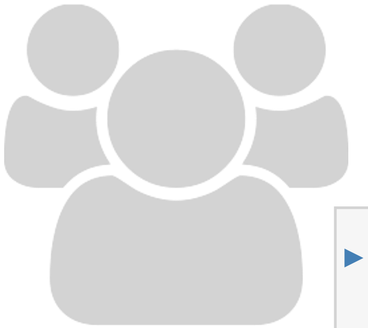
STATIC VS. DYNAMIC

- Static:
  - R code runs once and generates an HTML page
  - Generation of this HTML can be scheduled
- Dynamic:
  - Client web browser connects to an R session running on server
  - User input causes server to do things and send information back to client
  - Interactivity can be on client and server
  - Can update data in real time
  - User potentially can do anything that R can do

FLEX VS. SHINY DASHBOARD

flexdashboard	shinydashboard
R Markdown	Shiny UI code
Super easy	Not quite as easy
Static or dynamic	Dynamic
CSS flexbox layout	Bootstrap grid layout

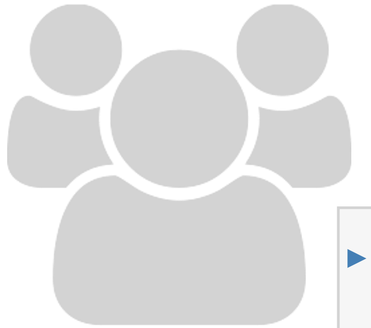
flexdashboard



EXERCISE

- library(flexdashboard)
- File → New file → R Markdown → From Template
- Create three plots that go in each of the panes using built-in R datasets or any data we have used in the course (or your own data)

3m 00s



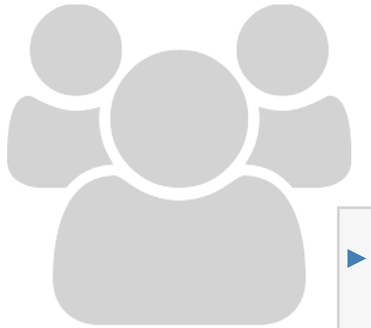
EXERCISE

- Open apps/flexdashboard_01.Rmd
- How is it different than Shiny apps we have been building so far, how is it similar?
- Make a change to the layout of the dashboard, see https://rstudio.github.io/flexdashboard/articles/using.html#layout-1 for help
- Change the theme of the dashboard, see https://rstudio.github.io/flexdashboard/articles/using.html#appearance-1 for help

5m 00s

SHINY DOCUMENTS

- Add `runtime: shiny` to header.
- Add `inputs` in code chunks.
- Add `renderXyz` functions in code chunks.
  - No need for `output$x <-` assignment, or for `xyzOutput` functions.



EXERCISE

- Continue working on apps/dashboards/flexdashboard_01.Rmd
- Add another UI widget, a radioButton, that allows the user to select whether the plot used to visualize the distribution of weight should be histogram or a violin plot

3m 00s

SOLUTION

Sample solution at apps/flexdashboard_02.Rmd

SHINY DOCUMENT DRAWBACKS

- Start-up time: knits document every time someone visits it
- Resizing can trigger re-knit
- Auto-reconnection doesn’t work (i.e. client browsers cannot automatically reconnect afer being disconnected due to network problems)

- **The solution:** Pre-rendered Shiny Documents

Shiny

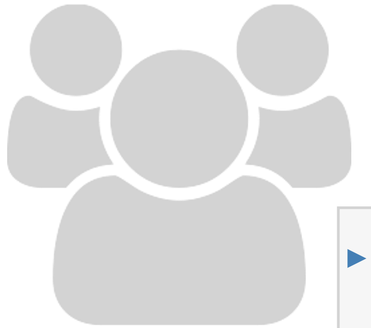
pre-rendered

SHINY PRE_RENDERED

- **Rendering phase:** UI code (and select other code) is run once, before users connect.
- **Serving phase:** Server code is run once for each user session.
- Each phase is run in a separate R sessions and can’t access variables from the other phase.

CONTEXTS FOR SHINY_PRERENDERED

- "render": Runs in rendering phase (like ui)
- "server": Runs in serving phase (like server)
- Additional contexts:
  - "setup": Runs in both phases (like global.R)
  - "data": Runs in rendering phase (any variables are saved to a file, and available to serving phase, useful for data preprocessing)
  - "server-start": Runs once in serving phase, when the Shiny document is first run and is not re-executed for each new user of the document, appropriate for
    - establishing shared connections to remote servers (e.g. databases, Spark contexts, etc.)
    - creating reactive values to be shared across sessions (e.g. with reactivePoll, reactiveFileReader)



EXERCISE

- Start with apps/flexdashboard_02.Rmd
- Turn your document into runtime: shiny_prerendered
- *Note:* You will need to use output$x <- assignment and xyzOutput functions

5m 00s

SOLUTION

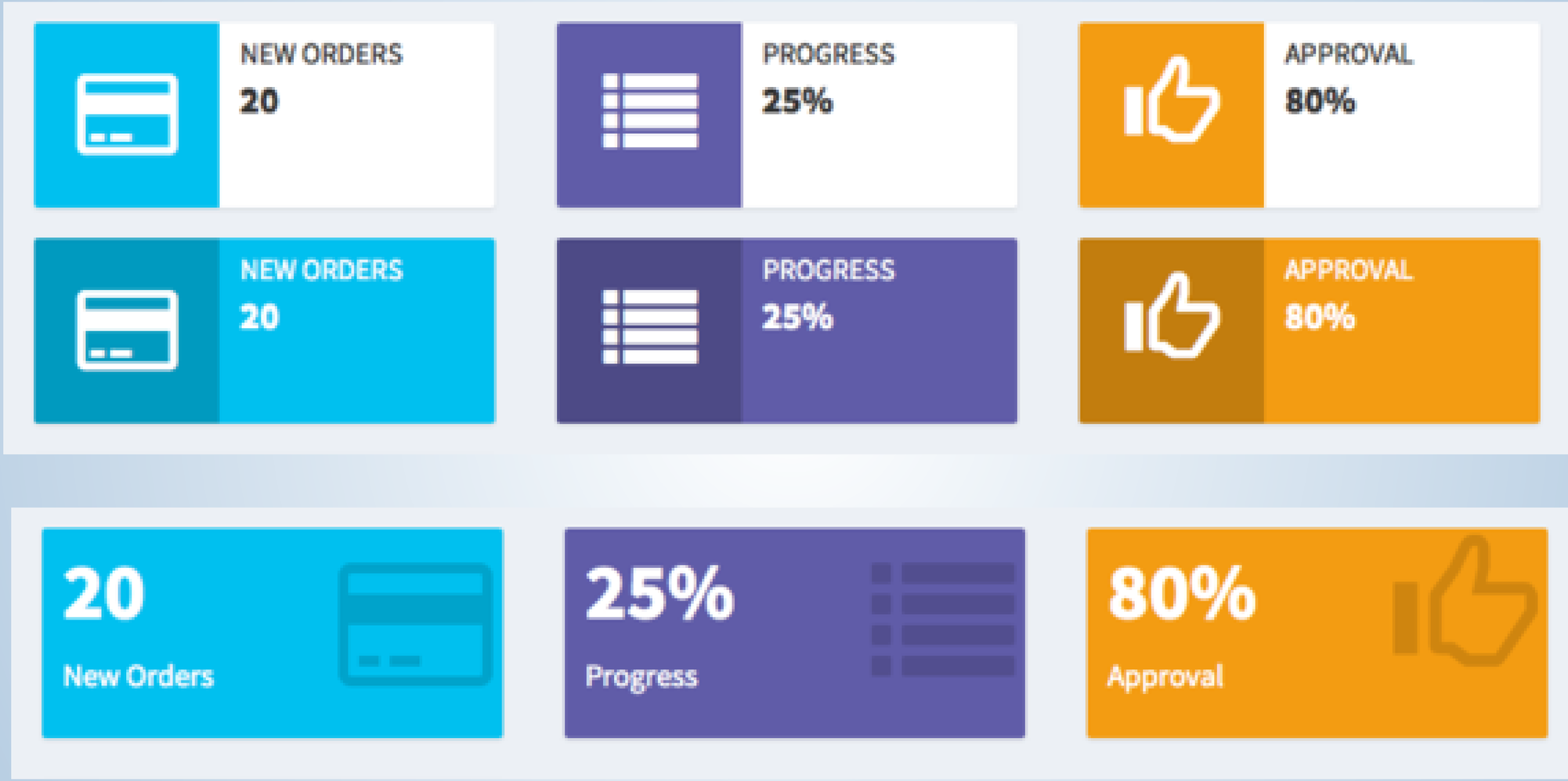
Sample solution at apps/flexdashboard_03.Rmd

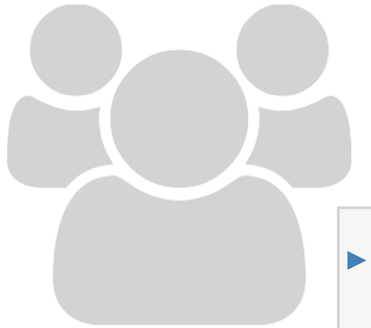
shinydashboard

FORMAT

- shinydashboard is an advanced layout of a typical shiny app
- The ui has more arguments
  - header
  - sidebarMenu
  - body (similar to fluid pages)
  - title
  - skin (color of the page)

Body


NEW ORDERS
20
PROGRESS
25%
APPROVAL
80%
NEW ORDERS
20
PROGRESS
25%
APPROVAL
80%
20
New Orders
25%
Progress
80%
Approval



EXERCISE

- Open starwars_01.R
  - Add an info or value box counting for mass and height respectively (lines 120 or 125)
    - Hint: First run the app to figure out what measurements might make sense
      - Stretch goal: Create the other kind of box

5m 00s

SOLUTION

See starwars_02.R

Tab1 Tab2 **First tabBox**

First tab content

Tab3 Tab2 Tab1

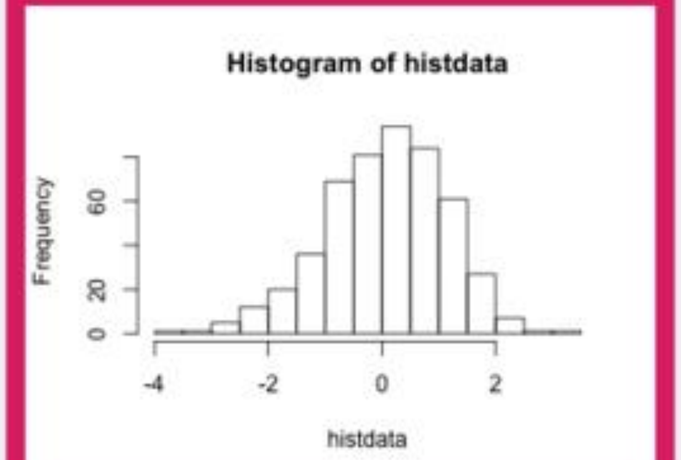
Note that when side=right, the tab order is reversed.

Tab1 Tab2 **tabBox status**

Currently selected tab from first box:

Tab1

Histogram



Inputs

Box content here

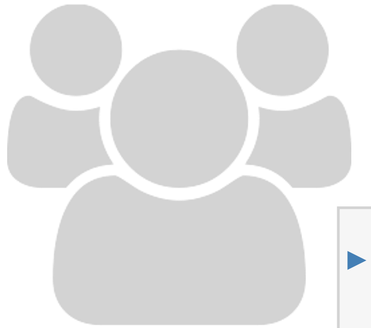
More box content

Slider input:

50

1 11 21 31 41 51 61 71 81 91 100

Text input:



EXERCISE

- Open starwars_02.R
  - Add a tabBox in the body that holds the output of both the plots for mass and height.
    - What arguments do you need to pass to the box so the table fits?
  - Stretch goal: Give the box a title

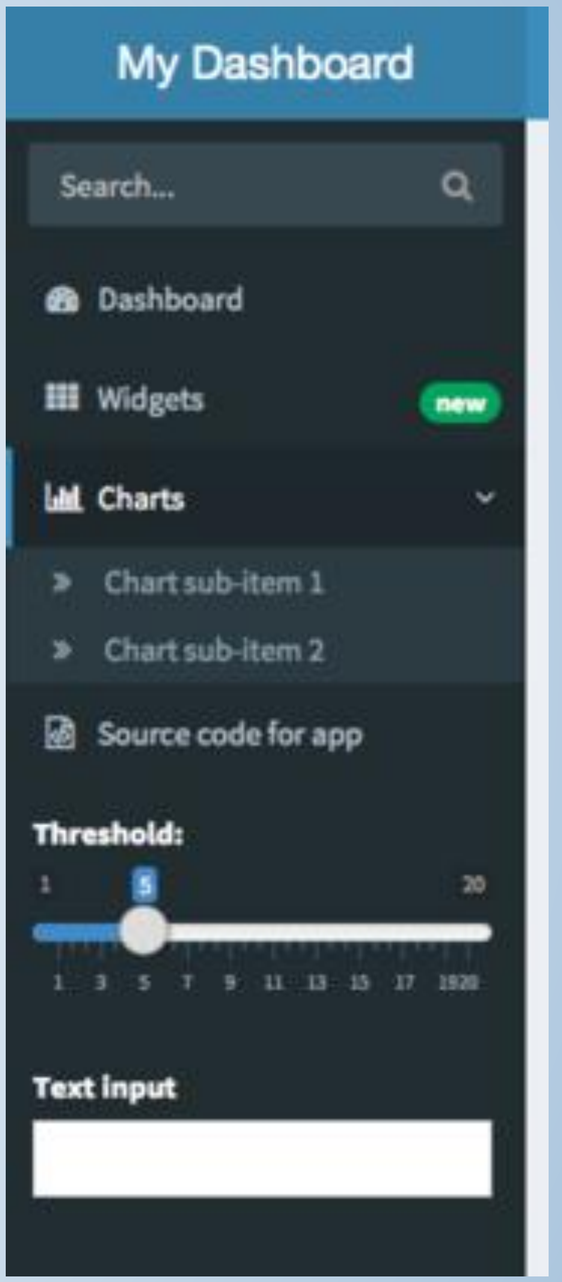
5m 00s

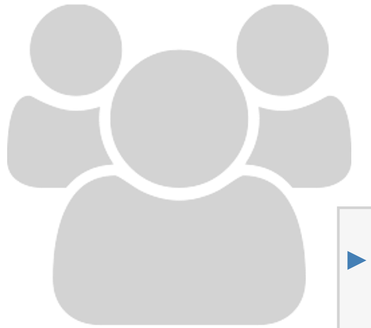
SOLUTION

See starwars_03.R

Menu


Simple tabs
Dashboard
Widgets
new
Widgets tab content


My Dashboard
Search...
Dashboard
Widgets
new
Charts
Chart sub-item 1
Chart sub-item 2
Source code for app
Threshold:
1
5
20
1 3 5 7 9 11 13 15 17 1920
Text input



EXERCISE

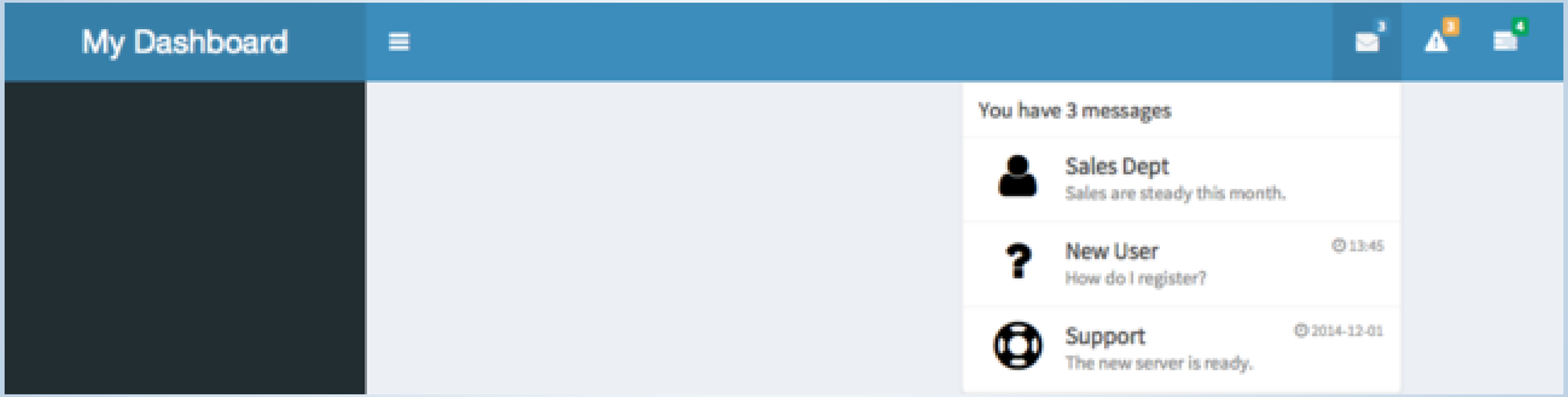
- Open starwars_03.R
  - Add a new menu item that allows users to access the table page

5m 00s

SOLUTION

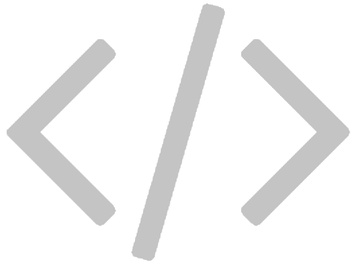
See starwars_04.R

Header


My Dashboard
You have 3 messages
Sales Dept
Sales are steady this month.
New User
13:45
How do I register?
Support
2014-12-01
The new server is ready.

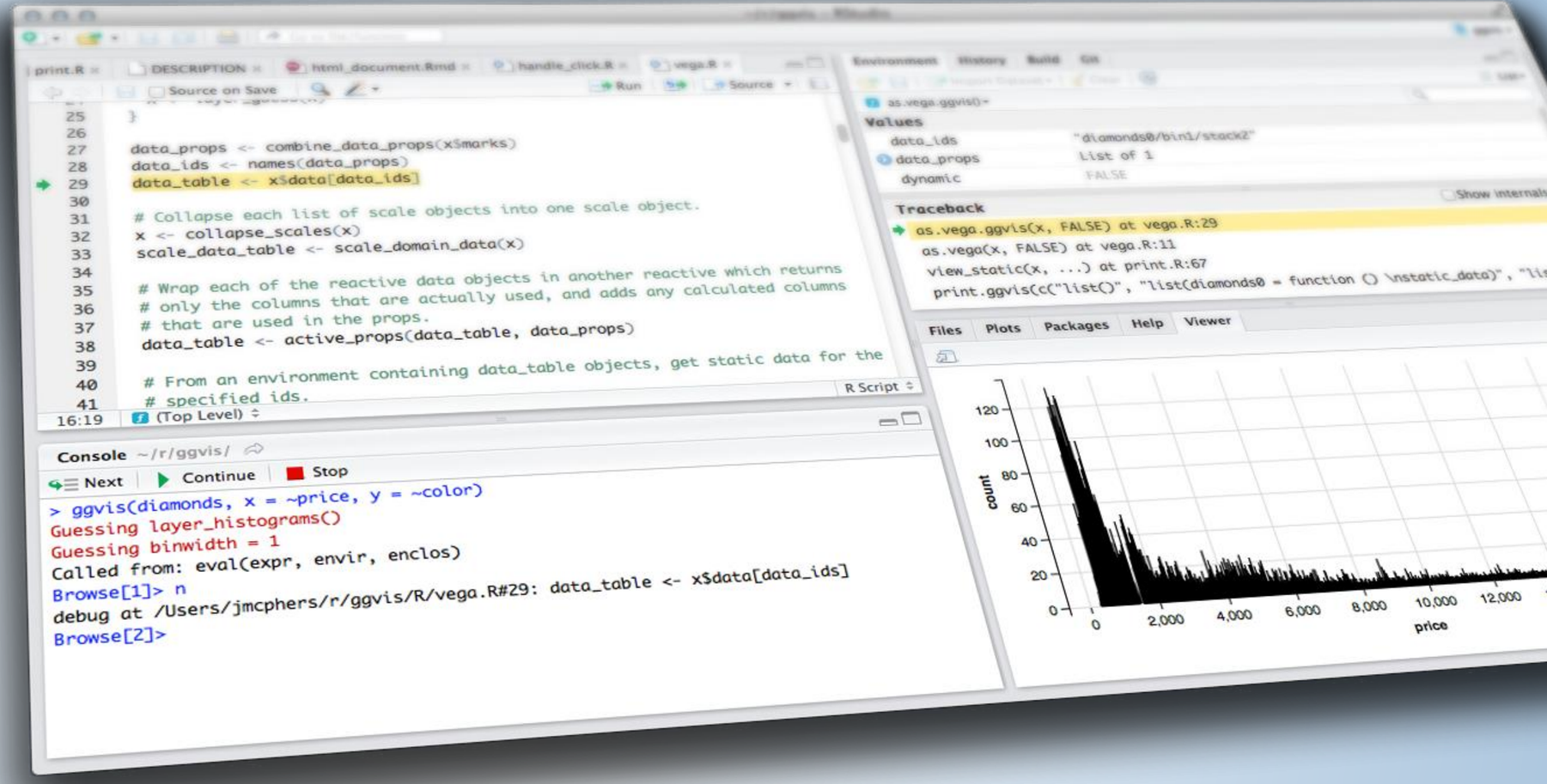
HEADER

- Headers have three types of information that can be displayed
  - messageItem - text information along with date/time information
  - notificationItem - basic text information
  - taskItem - show progress towards a goal
- All of these items can be dynamically updated and rendered in the server function
  - For examples see the shinydashboard docs



DEMO

starwars_04.R



DASHBOARDS

Shiny from posit

HOMEWORK

Creating multiple types of visuals from the same data is an important way to convey information to application users. Students will create a shiny or flexdashboard with:

- Three (3) input/filters
- Three (3) single numeric based boxes/gauges
- One (1) datatable
- Three (3) interactive (plotly) and reactively responsive charts.
- These elements should be places throughout a dashboard with at least three (3) pages or tabs with an analytical themes or question about the data.
- On the server side your plots and tables must utilize the reactive function for all datasets.