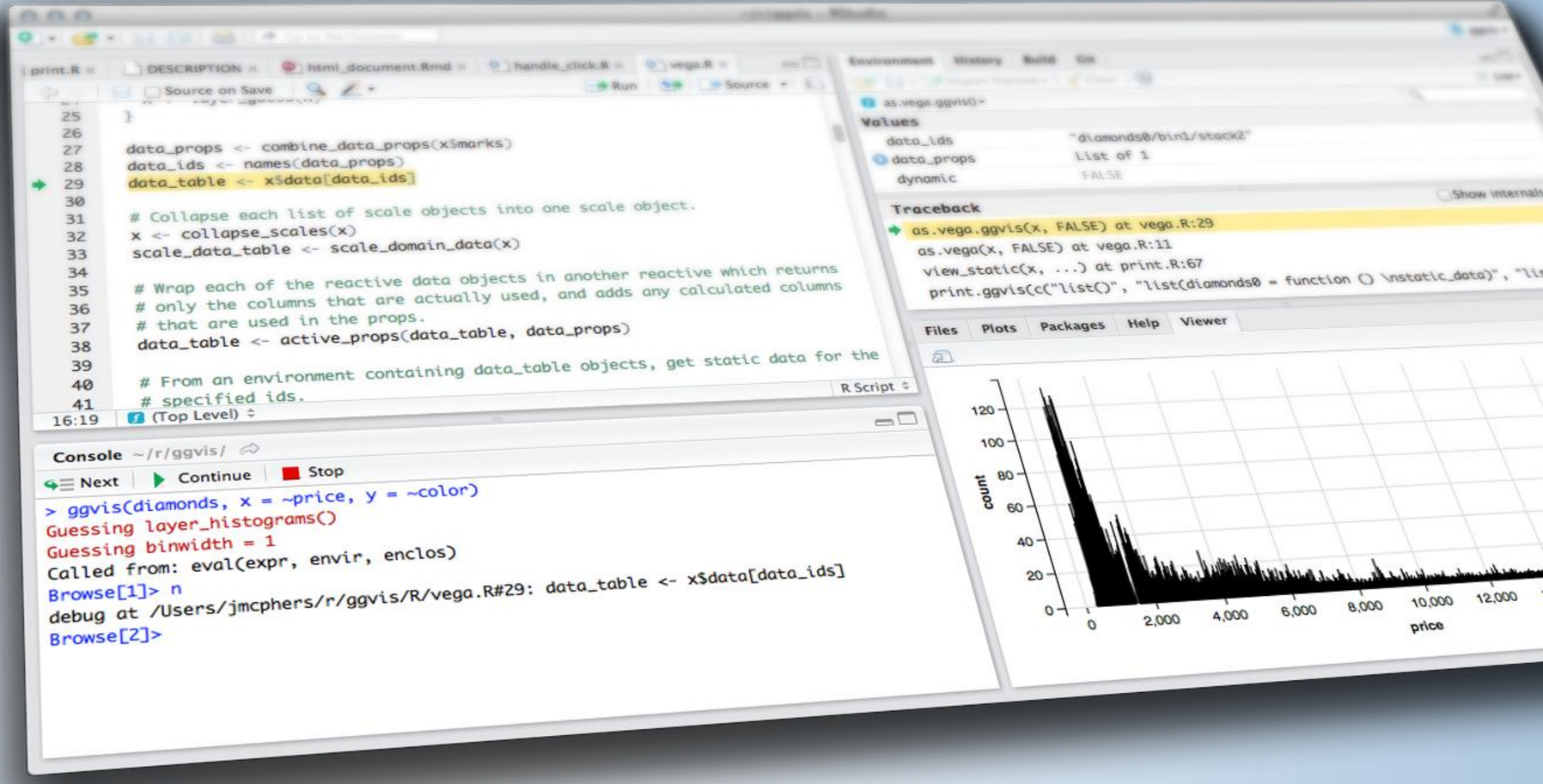


MODULES & BOOKMARKING

Shiny from posit

OUTLINE

- Modules
  - Anatomy of a Shiny module
    - User interface
    - Server function
    - Calling the module
    - Packaging modules
  - Exercise: Modularize this!
  - Combining modules
- Bookmarking
  - How to bookmark
  - Bookmarking options
  - Customizing bookmarks

What is a Shiny module?

MODULES

- A module is a self-contained, composable component of a Shiny app
  - self-contained like a function
  - can be combined to make an app
- Have their own UI and server (in addition to the app UI and server)

MODULES

- Useful for reusability
  - rather than copy and paste code, you can use modules to help manage the pieces that will be repeated throughout a single app or across multiple apps
  - can be bundled into packages
- Essential for managing code complexity in larger apps

LIMITATIONS TO JUST FUNCTIONALIZING

- It’s possible to write UI-generating functions and call them from your app’s UI, and you write functions for the server that define outputs and create reactive expressions
- However you must make sure your functions generate input and output IDs that don’t collide since input and output IDs in Shiny apps share a global namespace, meaning, each ID must be unique across the entire app
- **Solution:** Namespaces! Modules add namespacing to Shiny UI and server logic

“The thing to keep in mind about defining shiny UI, is that the main tool for managing complexity is the same tool as everywhere else in R: ***the function****. Since Shiny user interfaces are defined using function calls, take advantage of that fact by reusing and encapsulating UI creation logic into functions. The same can be said for defining server logic--write functions, and when those functions get too complicated, split them into more functions.”*

–Joe Cheng, 7/9/2015

“Roughly, hygienic macro expansion is desirable for the same reason as lexical scope: both enable local reasoning about binding so that program fragments compose reliably.”

–Matthew Flatt

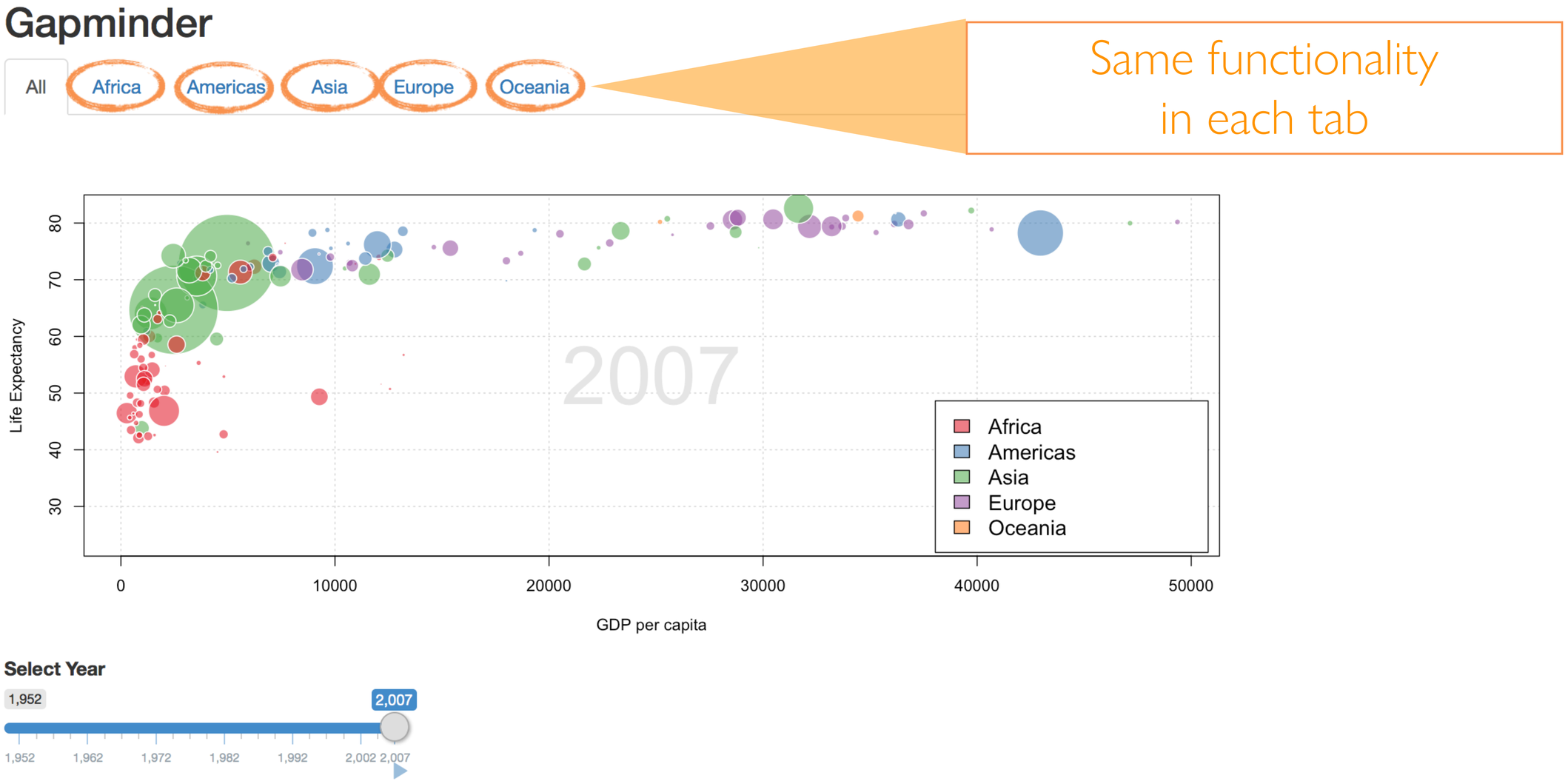
Shiny Modules

*“Roughly, ~~hygienic macro expansion~~ is desirable for the same reason as lexical scope: both enable **local reasoning** about binding so that program fragments **compose reliably**.”*

–Matthew Flatt

gapminder_app.R

DEMO



Shiny from RStudio™

LADDER OF PROGRESSION

- Step 1. Use modules to break large monolithic apps into manageable pieces
- Step 2. Create reusable modules
- Step 3. Nest modules

Anatomy of a Shiny module

WHAT'S IN A MODULE?

```
library(shiny)
name_of_module_UI <- function(id, label = "Some label") {
  # Create a namespace function using the provided id
  ns <- NS(id)
  # UI elements go here
  tagList(
    ...
  )
}

name_of_module <- function(input, output, session, ...) {
  # Server logic goes here
}
```

User interface
controls the layout and appearance of module

Server function
contains instructions needed to build module

MODULE VS. APP

- Similarities:
  - Inputs in UI can be accessed in server with input$
  - Outputs in UI can be defined in server with output$
- Differences:
  - Inputs/outputs cannot be directly accessed from outside the module namespace
  - If a module needs to use a reactive expression, take the reactive expression as a function parameter. If a module wants to return reactive expressions to the calling app, then return a list of reactive expressions from the function
  - If a module needs to access an input that isn't part of the module, the containing app should pass the input value wrapped in a reactive expression

User interface

MODULE UI

- A function
- Takes, as input, an id that gets pre-pended to all HTML element ids with a helper function: `NS()`
- Can also have additional parameters

Server function

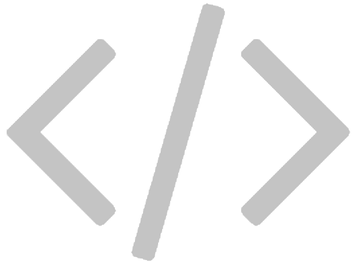
MODULE SERVER

- Includes the code needed for your module
- Looks almost identical to the app server function, except that you may have additional parameters
- App server function is automatically invoked by Shiny; module server function must be invoked by the app author

Calling the module

CALLING THE MODULE IN YOUR APP

- In the app UI:
  - Include the module UI with `name_of_module_UI("id", ...)`
  - Can also include other UI elements that are not included in the module
- In the app server:
  - Include the module server with `callModule(name_of_module, "id", ...)`
  - Can also include other UI elements that are not included in the module
- The id must match and must be unique among other inputs/outputs/modules at the same "scope" (either top-level ui/server, or within a parent Shiny module)



DEMO

gapminder.R

ui:

```
ui <- fluidPage(
  …
  titlePanel("Gapminder"),
  tabsetPanel(id = "continent",
    tabPanel("All", gapModuleUI("all")),
    tabPanel("Africa", gapModuleUI("africa")),
    tabPanel("Americas", gapModuleUI("americas")),
    tabPanel("Asia", gapModuleUI("asia")),
    tabPanel("Europe", gapModuleUI("europe")),
    tabPanel("Oceania", gapModuleUI("oceania"))
  )
)
```

server:

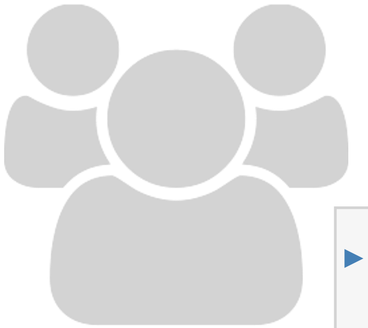
```
server <- function(input, output) {
  callModule(gapModule, "all", all_data)
  callModule(gapModule, "africa", africa_data)
  callModule(gapModule, "americas", americas_data)
  callModule(gapModule, "asia", asia_data)
  callModule(gapModule, "europe", europe_data)
  callModule(gapModule, "oceania", oceania_data)
}
```

Packaging modules

PACKAGING MODULES

- **Inline code:** Put the UI and server function code of the module directly in your app
  - If using app.R (single-file) syle, include module code in that file, before the app's UI and server logic
  - If using ui.R / server.R (two-file) style, include module code in the global.R file
  - If defining many modules / modules containing a lot of code, may result in bloated global.R/app.R file.
- **Standalone R file:**
  - Save module code in a separate .R file and then call `source("path-to-module.R")` from app.R or global.R
  - Probably the best approach for modules that won't be reused across applications
- **R package:** Useful if your modules are intended to be reused across apps
  - R package needs to export and document your module's UI and server functions
  - R packages can include more than one module

Modularize this!



EXERCISE

- See `movies_20.R` for a simpler version of the movie browser:
  - only select x, y, and z (for colors) variables
  - and alpha level and size of points
  - three tabs: one for each title type, showing a scatterplot and data table
- Note that this app is created by repeating the plotting and data table code chunks three times each
- Modularize the app using `moviesmodule_template.R`

10m 00s

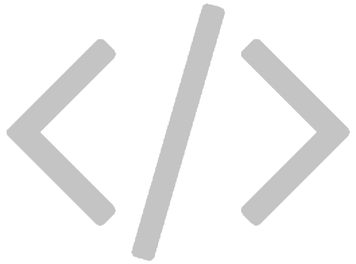
SOLUTION

- App: movies_21.R
- Module: moviesmodule.R

Combining modules

COMBINING MODULES

- When building an app that uses modules that depend on each other, avoid violating the sanctity of the module's namespace (similar to a function's local environment)
- If results of Module 1 will be used as inputs in Module 2, then Module 1 needs to return those results as an output, so that Module 2 does not have to “reach in and grab them”



DEMO

left_right_01.R

Choose a dataset

pressure

Number of records to return

10

```
 temperature      pressure
Min.   :  0   Min.   :0.0002
1st Qu.: 45   1st Qu.:0.0120
Median : 90   Median :0.1800
Mean   : 90   Mean   :1.5997
3rd Qu.:135   3rd Qu.:1.5750
Max.   :180   Max.   :8.8000
```

Choose a dataset

cars

Number of records to return

10

```
     speed           dist
Min.   : 4.0   Min.   : 2.0
1st Qu.: 7.0   1st Qu.:10.0
Median : 8.5   Median :16.5
Mean   : 8.0   Mean   :15.9
3rd Qu.:10.0   3rd Qu.:21.0
Max.   :11.0   Max.   :34.0
```

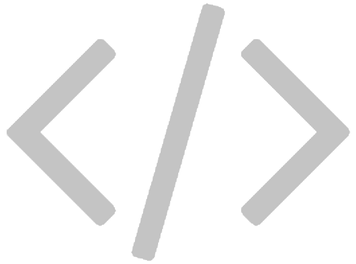
Module 1: Dataset chooser

DEMO

```
# Module 1 UI
dataset_chooser_UI <- function(id) {
  ns <- NS(id)

  tagList(
    selectInput(ns("dataset"), "Choose a dataset", c("pressure", "cars")),
    numericInput(ns("count"), "Number of records to return", 10)
  )
}

# Module 1 Server
dataset_chooser <- function(input, output, session) {
  dataset <- reactive({
    req(input$dataset)
    get(input$dataset, pos = "package:datasets")
  })

  return(list(
    dataset = dataset,
    count = reactive(input$count)
  ))
}
```

Why not dataset()**?**

Why wrapped with reactive()**?**



Module 2: Dataset summarizer

DEMO

```
# Module 2 UI
dataset_summarizer_UI <- function(id) {
  ns <- NS(id)

  verbatimTextOutput(ns("summary"))
}

# Module 2 Server
dataset_summarizer <- function(input, output, session, dataset, count) {
  output$summary <- renderPrint({
    summary(head(dataset(), count()))
  })
}
```

App combining the two modules

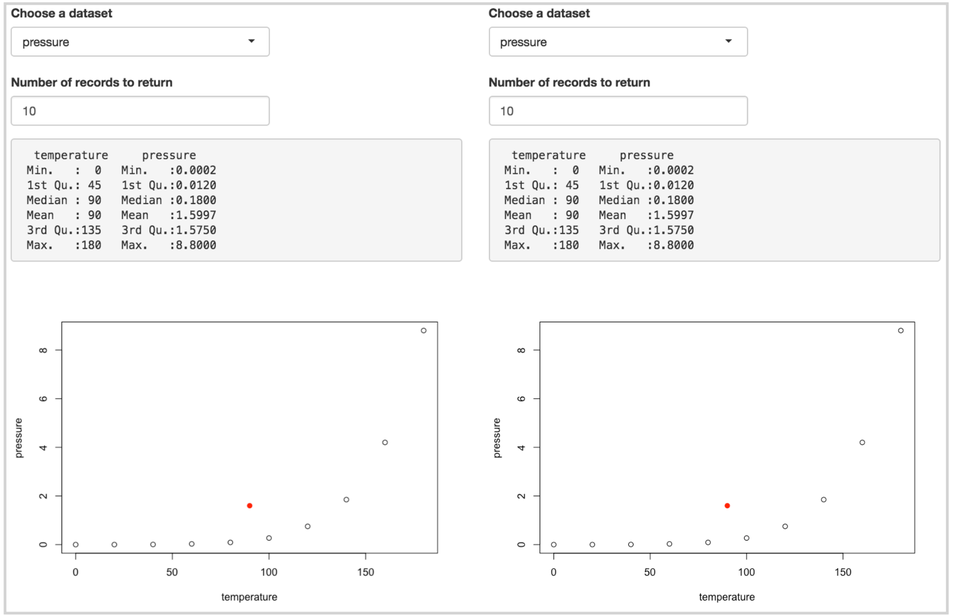
DEMO

```
# App UI
ui <- fluidPage(
  fluidRow(
    column(6,
      dataset_chooser_UI("left_input"),
      dataset_summarizer_UI("left_output")
    ),
    column(6,
      dataset_chooser_UI("right_input"),
      dataset_summarizer_UI("right_output")
    )
  )
)

# App server
server <- function(input, output, session) {
  left_result <- callModule(dataset_chooser, "left_input")
  right_result <- callModule(dataset_chooser, "right_input")

  callModule(dataset_summarizer, "left_output",
             dataset = left_result$dataset, count = left_result$count)
  callModule(dataset_summarizer, "right_output",
             dataset = right_result$dataset, count = right_result$count)
}
```

EXERCISE

- Start with left_right_01.R
- Update the 2nd module, dataset summarizer, so that it also returns mean of each of the two variables
- Add a 3rd module, dataset plotter, that makes a scatterplot of the selected dataset, and overlays a red point at (mean of x, mean of y) that it takes from the results of the dataset summarizer
- Add this module into your app such that the plot is printed underneath the summary

Desired outcome:



10m 00s

Module 2: Dataset summarizer (updated)

SOLUTION

```
# Module 2 UI
dataset_summarizer_UI <- function(id) {
  ns <- NS(id)

  verbatimTextOutput(ns("summary"))
}

# Module 2 Server
dataset_summarizer <- function(input, output, session, dataset, count) {

  selected_data <- reactive({ head(dataset(), count()) })

  output$summary <- renderPrint({
    summary( selected_data() )
  })

  mean_x <- reactive({ mean(selected_data()[,1]) })
  mean_y <- reactive({ mean(selected_data()[,2]) })

  return(list(
    mean_x = mean_x,
    mean_y = mean_y
  ))
}
```

SOLUTION

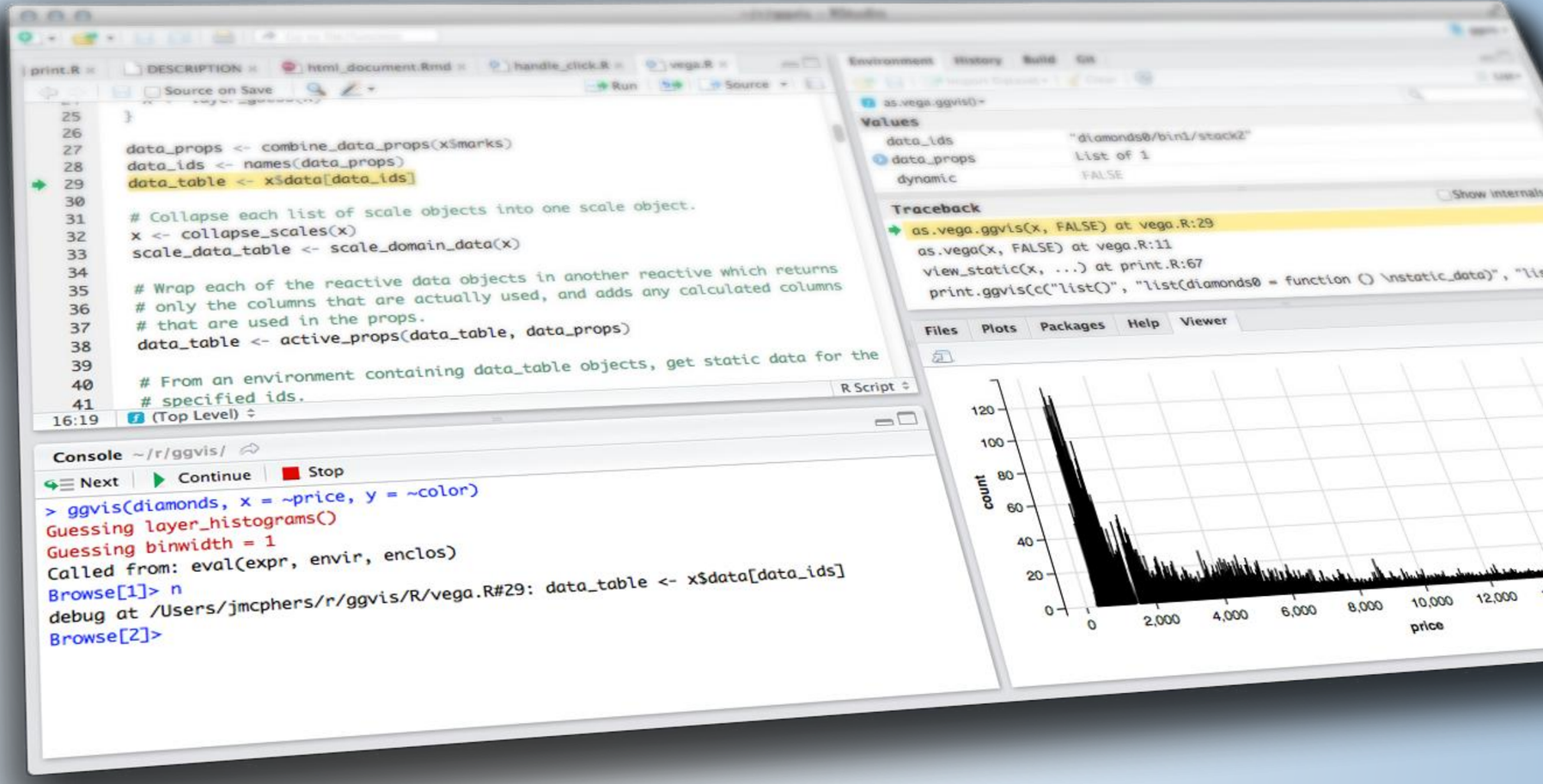
Module 3: Dataset plotter (new)

```
# Module 3 UI
dataset_plotter_UI <- function(id) {
  ns <- NS(id)

  plotOutput(ns("scatterplot"))
}

# Module 3 Server
dataset_plotter <- function(input, output, session, dataset, count,
                            mean_x, mean_y) {
  output$scatterplot <- renderPlot({
    plot(head(dataset(), count()))
    points(x = mean_x(), y = mean_y(), pch = 19, col = "red")
  })
}
```

SOLUTION

App combining all three modules

See left_right_02.R for details



BOOKMARKING

Shiny from posit

Motivation

MOTIVATION

- Capture the results you're currently seeing in a Shiny app, by snapshotting inputs and state
- Creates a URL that can be shared with others, or bookmarked for your own future use
- Applications:
  - Collaboration across researchers: Biostatistician creates an app, passes on to the biologist to explore, biologist wants to communicate specific findings back to the biostatistician
  - Sharing results with a coworker/manager: Instead of sending someone to your app and then asking them to click this, check that, choose this, etc.
  - Education:
    - If using a Shiny app to teach concepts, students might be asked to play around with a distribution by tweaking inputs, and submit the output
    - Or an instructor might want to provide multiple scenarios in an app for students to explore

How to bookmark

A VERY SIMPLE APP

```
ui <- fluidPage(
    textInput("txt", "Enter text"),
    checkboxInput("caps", "Capitalize"),
    verbatimTextOutput("out")
  )

server <- function(input, output, session) {
  output$out <- renderText({
    ifelse(input$caps, toupper(input$txt), input$txt)
  })
}

shinyApp(ui, server)
```

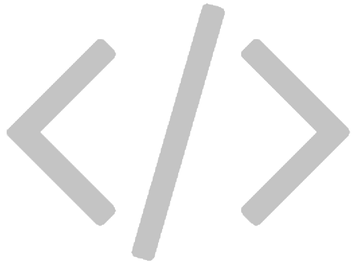
ENABLING BOOKMARKING

```
ui <- function(request) {
  fluidPage(
    textInput("txt", "Enter text"),
    checkboxInput("caps", "Capitalize"),
    verbatimTextOutput("out"),
    bookmarkButton()
  )
}

server <- function(input, output, session) {
  output$out <- renderText({
    ifelse(input$caps, toupper(input$txt), input$txt)
  })
}

shinyApp(ui, server, enableBookmarking = "url")
```

UI portion must be a function that takes one argument

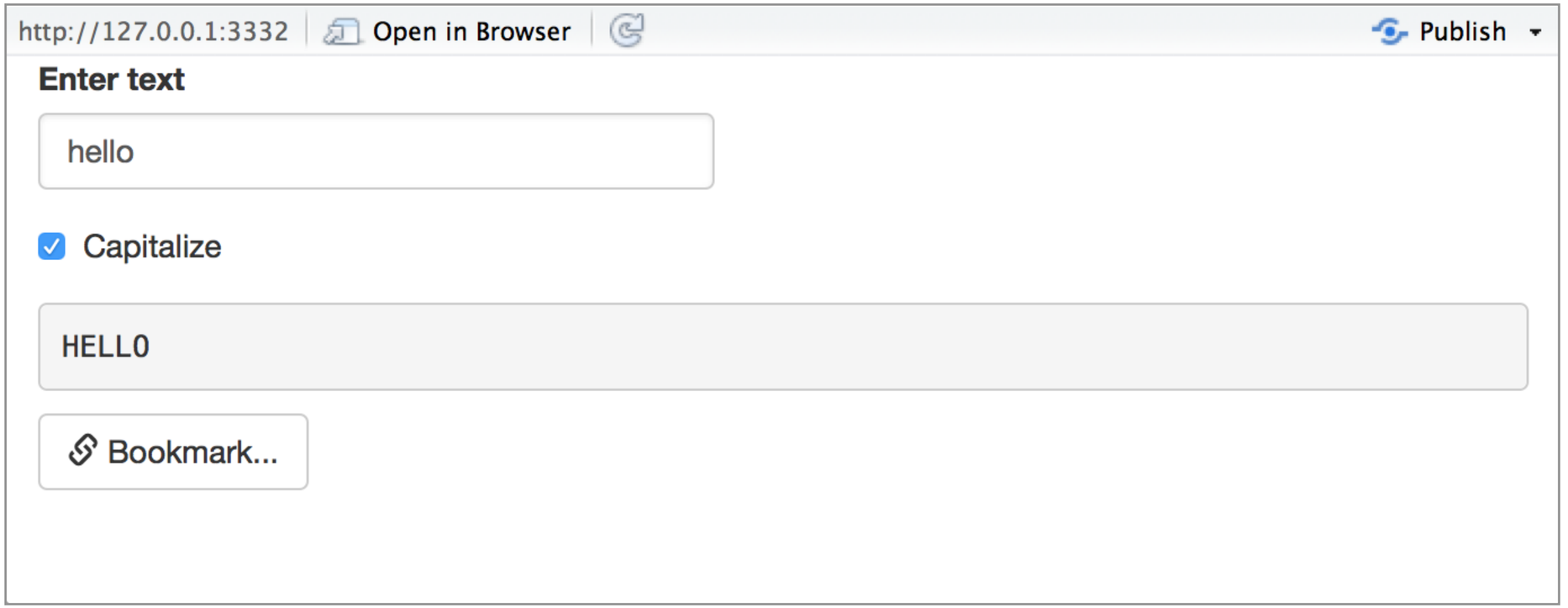
A button for bookmarking

There must be a call to enableBookmarking()



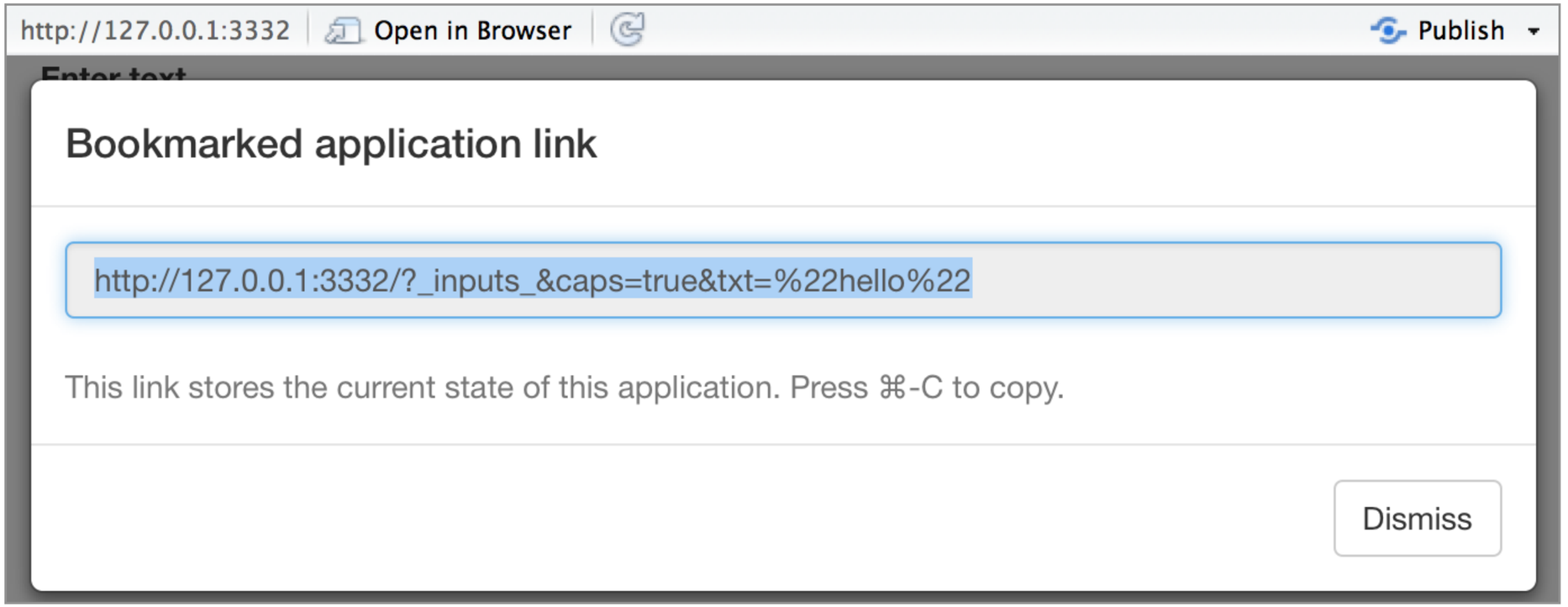
bookmark_01.R

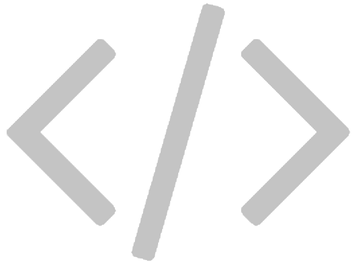
DEMO



bookmark_01.R

DEMO





DEMO

bookmark_01.R

HOW BOOKMARKING WORKS

- Bookmarked state automatically saves the input values (except passwords) such that when the application is restored using that state, the inputs are seeded with the saved values
- File inputs are saved only when state is saved to server (not with URL encoding)

Bookmarking options

TYPES OF BOOKMARKING

URL-encoded	Saved-to-server
No state on server	State saved on server
URLs may be long & values revealed in URL	Always a short URL
URL length limits data (~2K characters on some browsers)	No limits on data
Can’t store uploaded files	Can store uploaded files
Best for simple apps without too much state to serialize	Appropriate for large amounts of state, incl files and directories if necessary

ENABLING BOOKMARKING

```
ui <- function(request) {
  fluidPage(
    textInput("txt", "Enter text"),
    checkboxInput("caps", "Capitalize"),
    verbatimTextOutput("out"),
    bookmarkButton()
  )
}

server <- function(input, output, session) {
  output$out <- renderText({
    ifelse(input$caps, toupper(input$txt), input$txt)
  })
}

shinyApp(ui, server, enableBookmarking = "server")
```

Only change for saving to server

DEPLOYED EXAMPLES

URL-encoded
https://gallery.shinyapps.io/113-bookmarking-url/
https://gallery.shinyapps.io/113-bookmarking-url/?_inputs_&n=200

Customizing bookmarks

EXCLUDING INPUTS

- Inputs that should not be bookmarked can be excluded with setBookmarkExclude() in the server function, which takes in a vector containing the names of the inputs

```
server <- function(input, output, session) {
  setBookmarkExclude(c("x", "y"))
}
```

MULTIPLE TABS

- It is possible to bookmark which tab in a tabset is active
  - This requires providing IDs for `tabsetPanel()`, `navbarPage()`, or `navlistPanel()`
- If you have multiple tabs, you likely want the bookmark button to show up on each tab
  - This also requires providing IDs for `bookmarkButton()`, and also excluding the bookmarking of the bookmark buttons themselves

bookmark_02.R

Two tabs with checkboxes, and bookmark button on each tab

DEMO

```
server <- function(input, output, session) {

  # Need to exclude the buttons from themselves being bookmarked
  setBookmarkExclude(c("bookmark1", "bookmark2"))

  # Trigger bookmarking with either button
  observeEvent(input$bookmark1, {
    session$doBookmark()
  })
  observeEvent(input$bookmark2, {
    session$doBookmark()
  })
}
```

To trigger bookmarking from each button, use **observeEvent()** for each button that calls session**$doBookmark()**.

INPUT DEPENDENCE

- **Fully input-dependent apps:** The state of the outputs at time *t* is fully determined by the state of the inputs at time *t*
  - Bookmarking should “just work”
- **Partly input-dependent apps:** The state of the outputs at time *t* is only partly determined by the state of the inputs at time *t* — other things may influence it, including inputs at previous times, or external data
  - These apps are not fully reactive
  - Bookmarking requires additional tools to save & restore desired state

bookmark_03.R

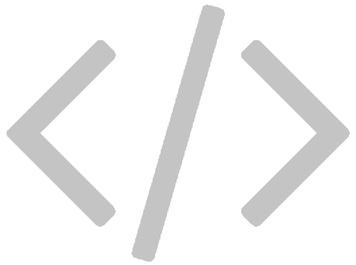
DEMO

Displays the sum of all previous slider values added

```
ui <- fluidPage(
    sidebarPanel(
      sliderInput("n", "Value to add", min = 0, max = 100, value =
50),
      actionButton("add", "Add"),
   ),
    mainPanel(
      h4("Sum:", textOutput("sum"))
    )
  )

server <- function(input, output, session) {
  vals <- reactiveValues(sum = 0)

  observeEvent(input$add, {
    vals$sum <- vals$sum + input$n
  })
  output$sum <- renderText({
    vals$sum
  })
}

shinyApp(ui, server, enableBookmarking = "url")
```

State of output not fully determined by state of inputs since previous input values matter as well

Each time “Add” button is clicked, add input$n to vals$sum, then render text of the sum value



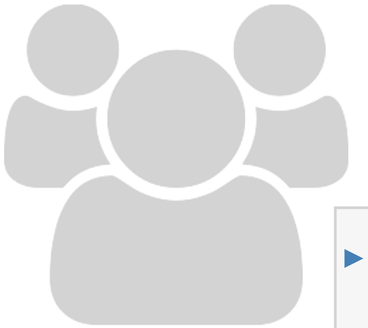
DEMO

bookmark_04.R

```r
ui <- function(request) {
  fluidPage(
    sidebarPanel(
      sliderInput("n", "Value to add", min = 0, max = 100, value = 50),
      actionButton("add", "Add"),
    ),
    mainPanel(
      h4("Sum:", textOutput("sum"))
    )
    bookmarkButton()
  )
}

server <- function(input, output, session) {
  vals <- reactiveValues(sum = 0)

  onBookmark(function(state) {
    state$values$currentSum <- vals$sum
  })
  onRestore(function(state) {
    vals$sum <- state$values$currentSum
  })

  observeEvent(input$add, {
    vals$sum <- vals$sum + input$n
  })
  output$sum <- renderText({
    vals$sum
  })
}

shinyApp(ui, server, enableBookmarking = "url")
```

When app is bookmarked, save vals$sum in the bookmark state values

When app is restored, retrieve vals$sum from the bookmark state values



EXERCISE

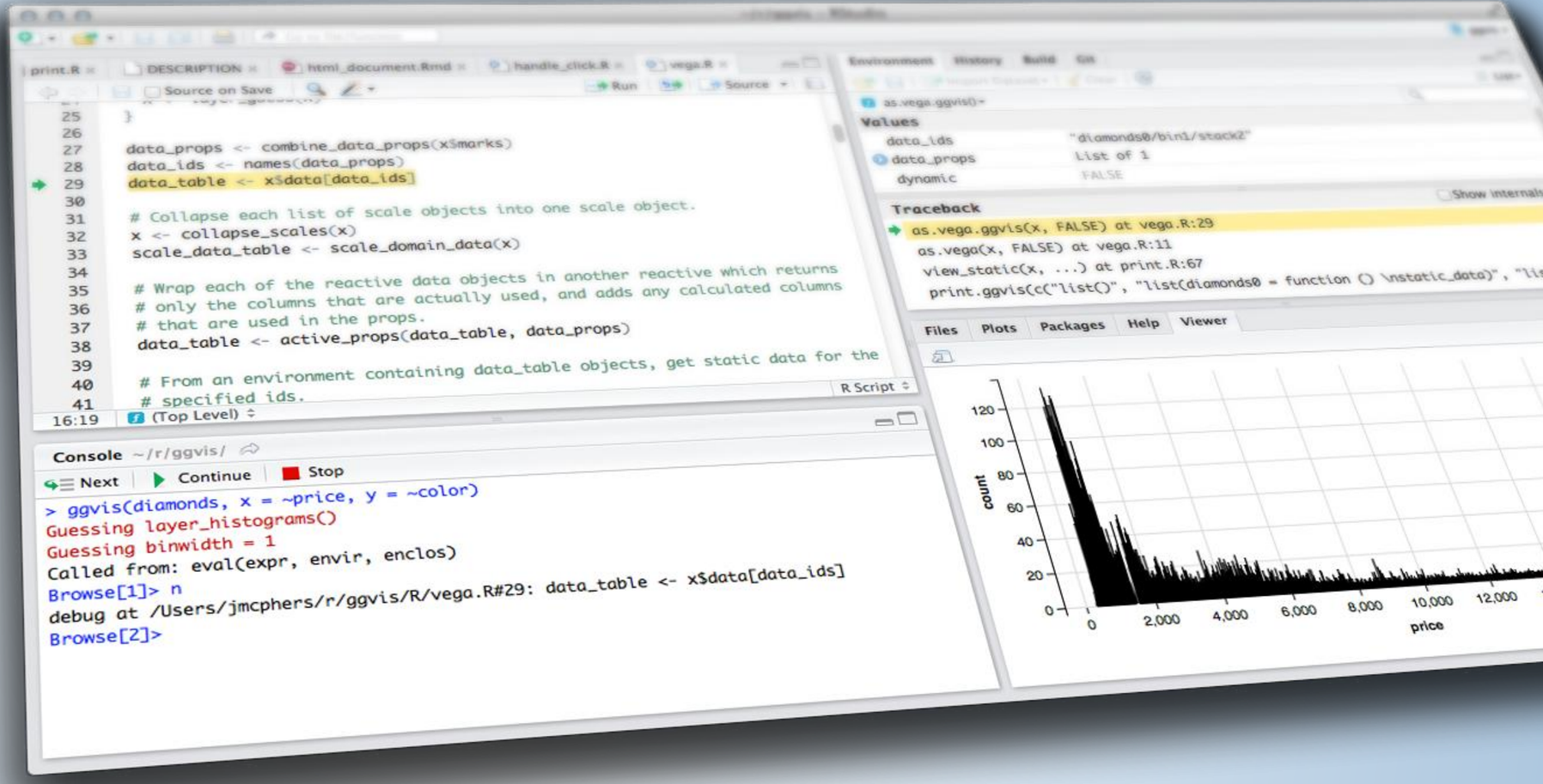
- Run bookmark_04.R
- There is a bug, what is it?
- Can you fix it?

3m 00s

SOLUTION

Solution to the previous exercise

bookmark_05.R



BOOKMARKING

Shiny from posit