Interactive Apps with Shiny

INFO 201

Joel Ross

Winter 2017

TECHNOLOGY LAB —

Microsoft hosts the Windows source in a monstrous 300GB Git repository

Virtualized file system approach makes Git work better for huge repositories.

PETER BRIGHT - 2/6/2017, 3:20 PM

Deadlines This Week

- Tue 02/21 (today): Assignment 7
- Thu 02/23: Project Proposal
- Fri 02/24: First Peer Evaluation
- Tue 02/28: Assignment 8 **(individual)**
- Thu 03/09: Project

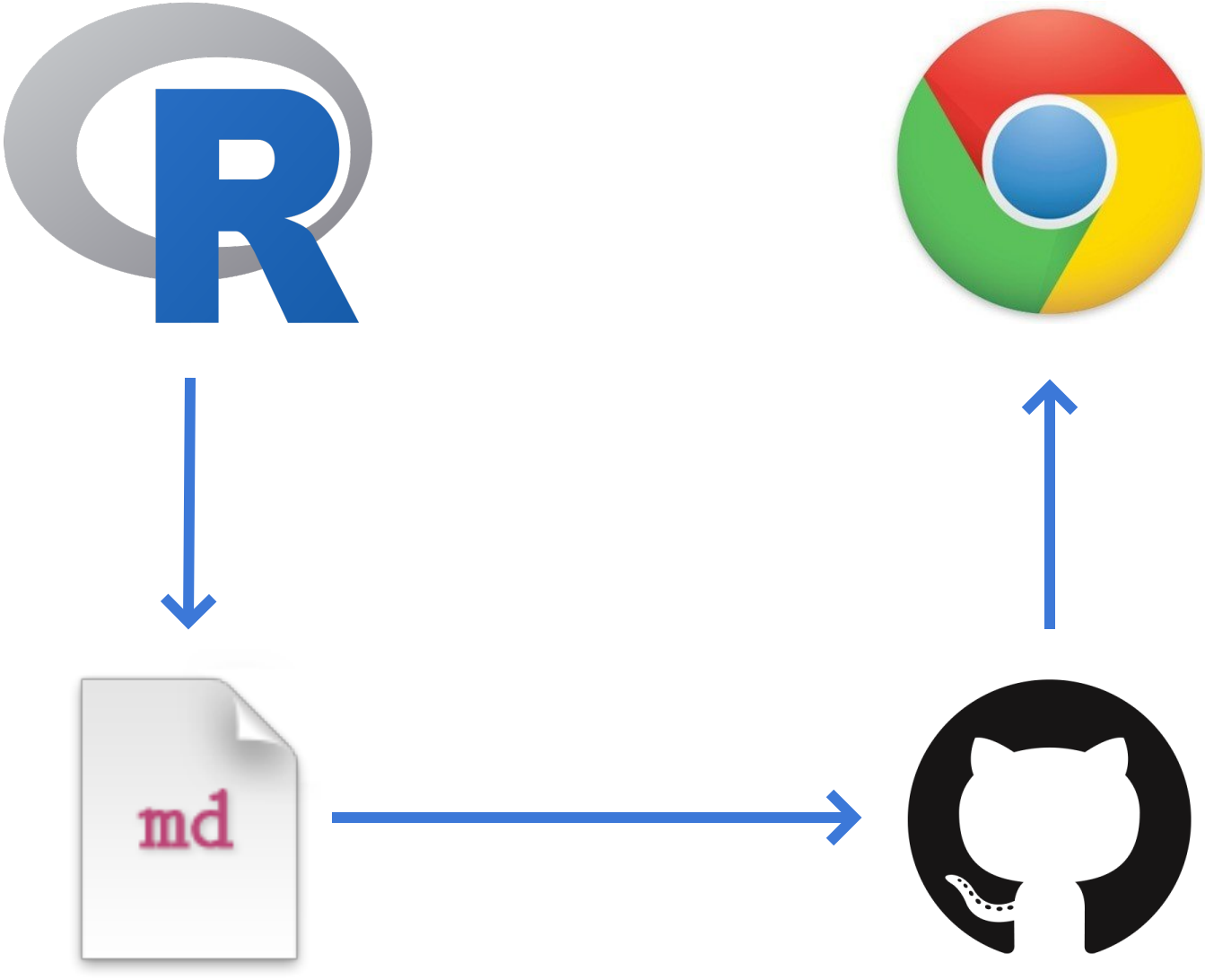
Today's Objectives

By the end of class, you should be able to

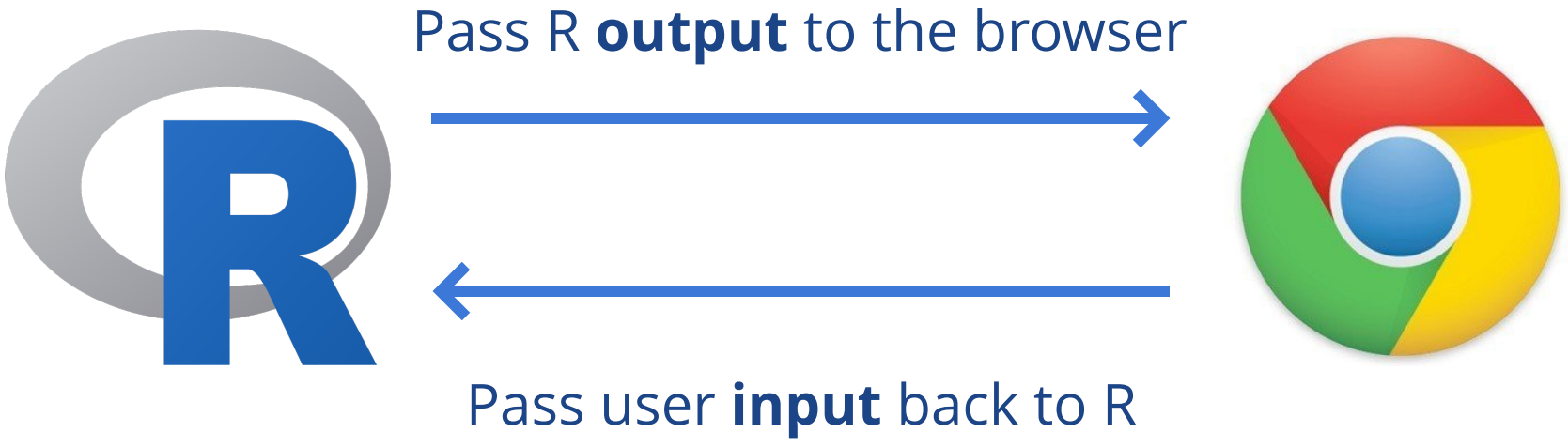
- Use the **Shiny** framework to make interactive, data-driven web applications
- Create beautiful interactive **user interfaces**
- Make Shiny interfaces **reactive**

Why would you want a data report to be ***interactive***?

Current Approach



Interactive Approach



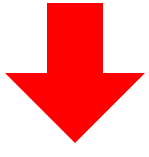
http://shiny.rstudio.com/

Shiny

Shiny is an R package that provides a ***framework*** for declaring interactive web applications.

Define your Shiny application in a file called `app.R` inside its own folder.

```
install.packages("shiny")  # once per machine
library("shiny")  # load the package
```



Application Structure

Shiny applications have two parts:

User Interface (UI)

- R code declaring how the app will appear in the *browser*

Server

- R code declaring data that will be displayed in the UI (**outputs**) and that the user can *interact* with (**inputs**).

  Somewhat like an R script the user will be able to run from the browser.

UI

The **UI** for a Shiny application is defined as a ***value*** (an "object") that is returned by a **layout function**.

```
# define a UI for the app
my.ui <- fluidPage(
  # formatted content
  h1("Hello Shiny"),

  # control widget
  textInput('user.name', label="What is your name?")
)
```

creates and returns a UI page

first-level header

Text input box

arguments to fluidPage() function

Server

The **Server** for a Shiny application is defined as a ***function*** with two arguments: a `list` of **input** values, and a `list` of **output** values.

```
# The server is a function that takes
# `input` and `output` arguments

my.server <- function(input, output) {

  # use values from `input`
  # assign values to `output`

  # we'll fill this in soon
}
```

A regular ol' function

Shiny App

Connect the **UI** and **Server** with the `shinyApp()` function. This will open up the app in R Studio's built-in browser!

```
# create and run the Shiny app
shinyApp(ui = my.ui, server = my.server)
```

- Can also hit the "Run App" button in RStudio
- If you change the UI or Server, simply reload the browser (no need to restart the App)
- Hit the Stop Sign to stop the App!

UI Elements

Shiny UIs can contain 4 different kinds of **content elements**. Content elements are created with **functions**, and specified as arguments to the layout function:

```
# "psuedocode" example
ui <- fluidPage(contentElement1(), contentElement2(),
                contentElement3(), contentElement4())
```

The types of content elements are:

- **UI layouts** (allows for nesting!)
- **styled content**
- **control widgets**
- **reactive outputs**

Styled Content

Styled content is created using functions named after **HTML elements**, which correspond to Markdown stylings:

- `p()` for creating paragraphs, the same as plain text in Markdown)
- `h1()`, `h2()`, `h3()` etc for creating headings, the same as `# Heading 1`, `## Heading 2`, `### Heading 3` in Markdown
- `em()` for creating *emphasized* (italic) text, the same as `_text_` in Markdown
- `strong()` for creating **strong** (bolded) text, the same as `**text**` in Markdown
- `a(text, href='url')` for creating hyperlinks (anchors), the same as `[text](url)` in Markdown
- `img(text, src='url')` for including images, the same as `![text](url)` in Markdown

Control Widgets

Control widgets are dynamic elements a user can interact with. Each stores a **value**, which ***automatically*** updates as user interacts with the widget.

```
# creates a text box
textInput('text.key', label = "Enter Text")

# creates a slider
sliderInput('slide.key', label = "Pick a number",
            min = 1, max = 20, value = 12)
```

"key" for the stored value

widget description

widget-specific arguments

The Server

The **server** is a function that manipulates `input` and `output` lists. The "value" from a widget is available in the `input` list.

```
ui <- fluidPage(
   textInput('user.name', label = "Enter Name")
)


server <- function(input, output) {

  # use input$user.name

}
```

← **value in input list**

Server: Render Functions

Assign the results of **render functions** to the `output` list. These functions take a **reactive expression** (an un-named function) as an argument.

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

  output$message <- renderText({

    my.mesage <- paste("Hello", input$user.name)
    return(my.message)
  })

}
```

returns text value to display

save in output list

like a function (just the block)

result of expression is text to display

end expression
end render function

*Reactive expressions are re-run every time an **input** they reference changes!*

UI: Reactive Outputs

Show the values from the *server's* `output` list by using **reactive output** functions in the UI.

```
ui <- fluidPage(
   textOutput('message')
)


server <- function(input, output) {

   output$message <- renderText({

     # ... determine message here
     return(my.message)
   })
}
```

Reactivity

Changes to **control widgets** update the value in `input`, which ***notifies*** the **reactive expression** to update the value in `output`, which is displayed by **reactive outputs**.

(1) user gives value to widget

(3) new output is shown

(2) change reruns the expression

```
ui <- fluidPage(
  textInput('user.name'),

  textOutput('message')
)


server <- function(input, output) {

  output$message <- renderText({

      my.message <- input$user.name
      return(my.message)
    })
}
```

Module 15 exercise-1

Sidebar Layouts

UI layouts can include *other layouts* as content elements, allowing you to create complex page organizations.

```
ui <- fluidPage(   # UI is a fluid page

  # include panel with the title (also sets browser title)
  titlePanel("My Title"),

  # layout the page in two columns
  sidebarLayout(

    sidebarPanel(  # specify content for the "sidebar" column
      p("sidebar panel content goes here")
    ),

    mainPanel(     # specify content for the "main" column
      p("main panel content goes here")
    )
  )
)
```

Multiple Views

Sometimes we want multiple displayed variables of the same data (model)

```
ui <- fluidPage(
  sliderInput('num', label="How many numbers", min=1, max=100, value=50),
  plotOutput('hist'),
  verbatimTextOutput('counts')
)

server <- function(input, output) {
  # render a histogram plot
  output$hist <- renderPlot({
    uniform.nums <- runif(input$num, 1, 10)  # random 1-10
    return( hist(uniform.nums, breaks=10) )  # built-in plotting
  })

  # render the counts
  output$counts <- renderPrint({
    uniform.nums <- runif(input$num, 1, 10)  # random 1-10
    counts <- factor(cut(uniform.nums, breaks=1:10))  # factor
    return( summary(counts) )  # simple vector of counts
  })
}

shinyApp(ui, server)
```

What is odd about this app?

Reactive Variables

If we want a variable to be shared between *reactive expressions*, we need to make that a **reactive variable** using the `reactive()` function:

```
server <- function(input, output) {
  # define a reactive variable
  uniform.nums <- reactive({
    return( runif(input$num, 1, 10) )
  })

  # render a histogram plot
  output$hist <- renderPlot({
    return( hist(uniform.nums()) )
  })

  # render the counts
  output$counts <- renderPrint({
    counts <- factor(cut(uniform.nums(), breaks=1:10))
    return( summary(counts) )
  })
}
```

just like a render function

call reactive variable AS A FUNCTION

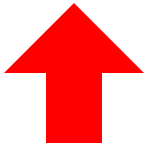
Module 15 exercise-2

Publishing Shiny Apps

Because Shiny apps require an R interpreter session to run the server, we can't just publish on GitHub Pages.

Instead, we can use free app hosting through

https://www.shinyapps.io/



Will need to

1. sign up for an account
2. install software
3. setup access token
4. publish app!

Action Items!

- Be comfortable with **module 15**
- Deadlines: Project Proposal, Peer Evaluation

Thursday: Advanced layouts and interactivity