RStudio®

Building Interactive Dashboards and Applications with Shiny

Materials: github.com/nrohr/learn-shiny
Questions? financial-services@rstudio.com

Nick Rohrbaugh
RStudio
nick@rstudio.com

Shiny

Shiny lets us build fully interactive web applications from R code

No knowledge of HTML, CSS, JavaScript, etc. required - only R!

Extremely flexible and customizable

Shiny website: shiny.rstudio.com

Shiny articles: https://shiny.rstudio.com/articles/

RStudio Cheat Sheets: https://rstudio.com/resources/cheatsheets/

Mastering Shiny book by Hadley: https://mastering-shiny.org/

Why Shiny? (Shiny vs. point-and-click)

Shiny	***Point-and-click dashboarding tools***
+ Open-source, no lock-in + Code-based = reproducible, repeatable, extensible, trackable + Powerful - can do anything you can do in R (or Python)	- Often closed ecosystems, vendor lock-in - Can be hard to reproduce work or track changes over time - Can be limited to basic slicing and dicing of data
- Steeper learning curve (but less if you know R!). Full customization requires deeper knowledge of code (potentially CSS/HTML)	+ Typically easier to learn

Parts of a Shiny app

1. Header - load packages and data, wrangle data, any other universal code
2. UI - add visual elements to the user interface. Define layout, inputs, outputs.
3. Server - Logic behind app, in R. R code that runs when users change inputs.
4. shinyApp() - run the app!

1. The Header

- The first part of your app.R file
- Everything in this section will run as soon as someone connects to your app
- Put things in here that you always need for every part of your app - loading data and packages, data wrangling/transformation steps, etc.

2. The UI (user interface)

- The second part of your app.R file
- Lives inside `ui( )`
- Define the visual layout of your app (where to put inputs and outputs)
- Define inputs and how they work
- Define outputs and what types they are

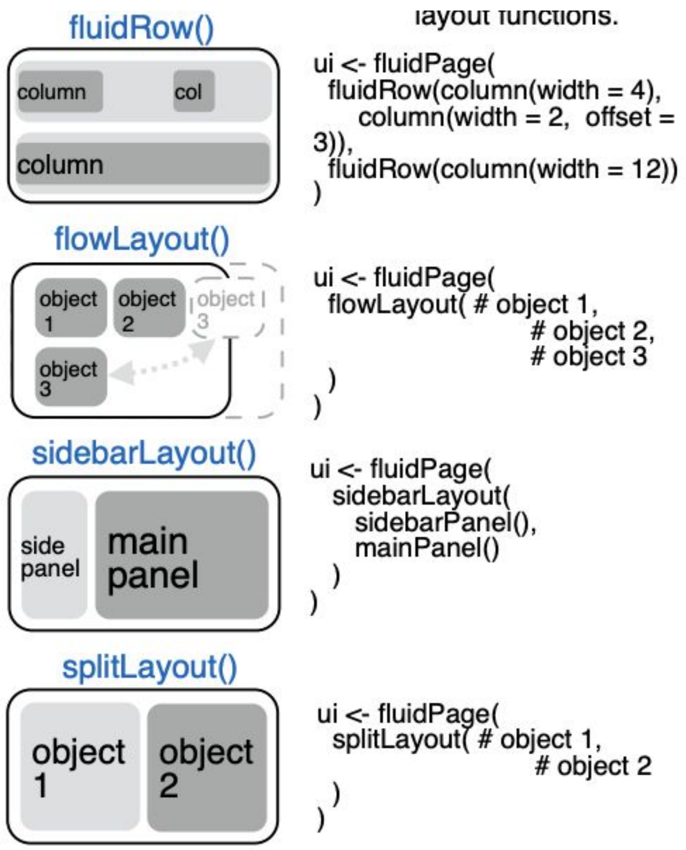
RStudio®

Layouts

Lots of ways to organize Shiny apps, but think in terms of rows, columns, and panels.

Layout is the outermost layer inside of the UI part of your Shiny app.

Not sure where to start? sidebarLayout() is a great starting point.



layout functions.

```
ui <- fluidPage(
  fluidRow(column(width = 4),
      column(width = 2,  offset =
3)),
  fluidRow(column(width = 12))
)
```

```
ui <- fluidPage(
  flowLayout( # object 1,
                          # object 2,
                          # object 3
  )
)
```

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

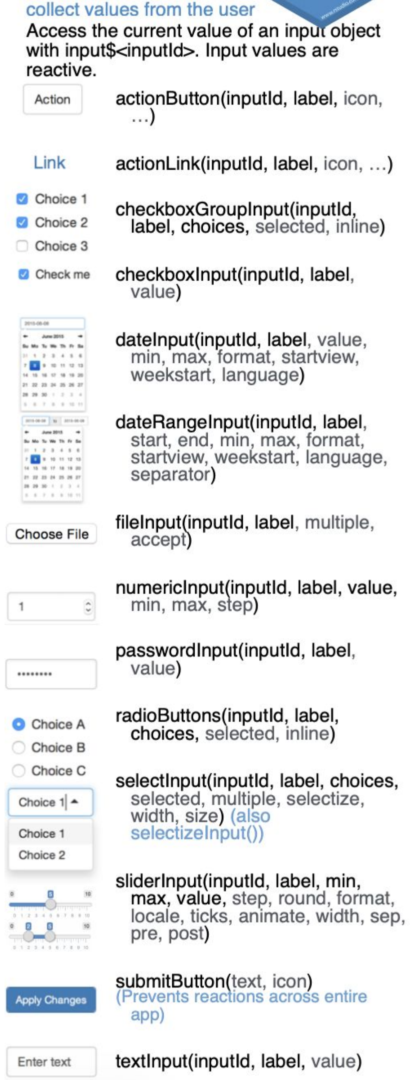
```
ui <- fluidPage(
  splitLayout( # object 1,
                         # object 2
  )
)
```

Inputs

Inputs (and outputs) are nested inside of your layout functions

Many different ways to collect user input:

- Select from drop-down menu (selectInput)
- Multiple choice w/ radio buttons (radioButtons)
- Choose dates on a calendar (dateInput + dateRangeInput)
- Checkboxes (checkboxInput)
- Upload files (fileInput)
- and more

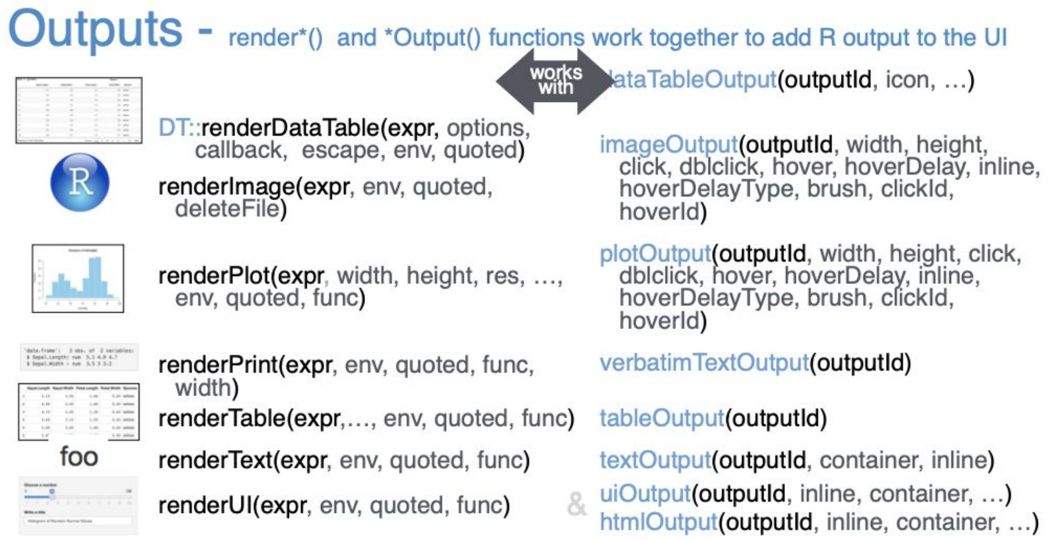


collect values from the user

Access the current value of an input object with input$<inputId>. Input values are reactive.

Action	actionButton(inputId, label, icon, …)
Link	actionLink(inputId, label, icon, …)
Choice 1 Choice 2 Choice 3	checkboxGroupInput(inputId, label, choices, selected, inline)
Check me	checkboxInput(inputId, label, value)
	dateInput(inputId, label, value, min, max, format, startview, weekstart, language)
	dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)
Choose File	fileInput(inputId, label, multiple, accept)
1	numericInput(inputId, label, value, min, max, step)
••••••••	passwordInput(inputId, label, value)
Choice A Choice B Choice C	radioButtons(inputId, label, choices, selected, inline)
Choice 1 Choice 1 Choice 2	selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (also selectizeInput())
	sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)
Apply Changes	submitButton(text, icon) (Prevents reactions across entire app)
Enter text	textInput(inputId, label, value)

Outputs



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

works with

DT::renderDataTable(expr, options, callback, escape, env, quoted)	ataTableOutput(outputId, icon, ...)
renderImage(expr, env, quoted, deleteFile)	imageOutput(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)
renderPlot(expr, width, height, res, ..., env, quoted, func)	plotOutput(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)
renderPrint(expr, env, quoted, func, width)	verbatimTextOutput(outputId)
renderTable(expr,..., env, quoted, func)	tableOutput(outputId)
renderText(expr, env, quoted, func)	textOutput(outputId, container, inline)
renderUI(expr, env, quoted, func)	uiOutput(outputId, inline, container, ...) & htmlOutput(outputId, inline, container, ...)

Two <u>required</u> parts to every output:

1. render() function in server()
2. Output() function in ui()

If we have a plotOutput() in our ui(), we need a renderPlot() in our server()

Output functions are simple and tell Shiny what type of thing we're displaying.

Render functions contain your R code (building a ggplot) and tell Shiny *how* to build something.

3. The server()

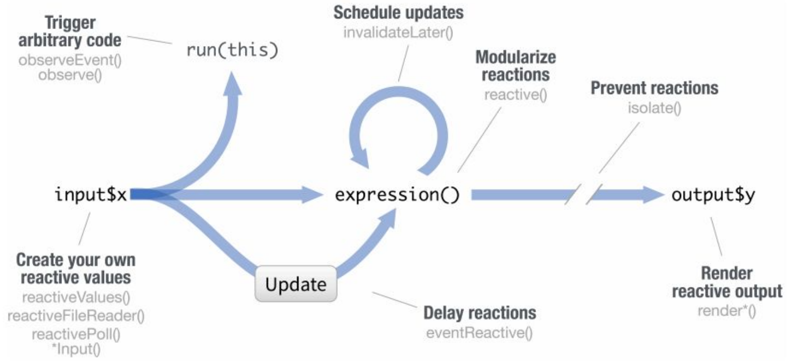
- The third part of your app.R file
- Lives inside `server( )`
- Most of your R code (copied from other files, scripts, notebooks, analyses) lives here.
- Tells Shiny *how* to build outputs.
- Can use values from user inputs by calling input$id
- ggplot2 code, data wrangling that requires user input

4. shinyApp()

- The fourth and final part of your app.R file
- Tells Shiny to run the app!
- Needs to know where to find UI and server parts

Reactivity

Reactive values work together with reactive functions. Call a reactive value from within the arguments of one of these functions to avoid the error Operation not allowed without an active reactive context.



Shiny apps are *reactive* - when something changes, it has downstream consequences.

When an input changes, anything that uses that input will automatically update.

You can build very complex reactive structures to get your app to update the right parts at the right time.

- Wait until users click a submit button before changing everything
- Automatically check if a file has been updated every *x* seconds