Building web applications with Shiny in R

Ram Thapa
Biostatistics Core

Bio-Data Club
2019/01/25

Shiny is RStudio's framework for building interactive web applications in **R**

Shiny is an **R** Package to deploy web apps using an **R** backend

We don't have to know any HTML, CSS, or JavaScript

Fairly easy to use for someone who is not a programmer

Great tool for interactive visualization

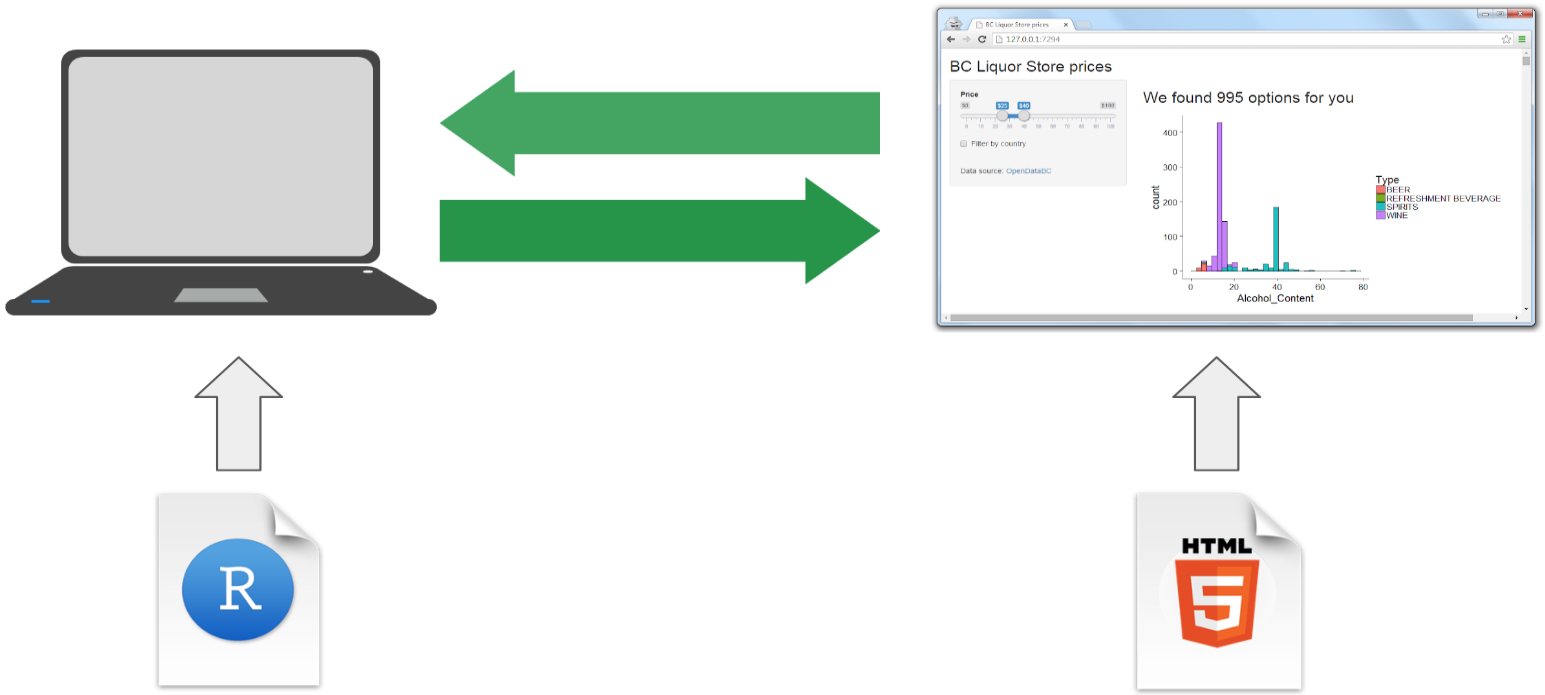
Anatomy of a Shiny app

There are two main components of Shiny app

1. **User Interface** or **UI**: this defines a webpage that the user interacts with, it controls layout and appearance of the app. This is the part which is visible to the user and thus supports user interactivity.

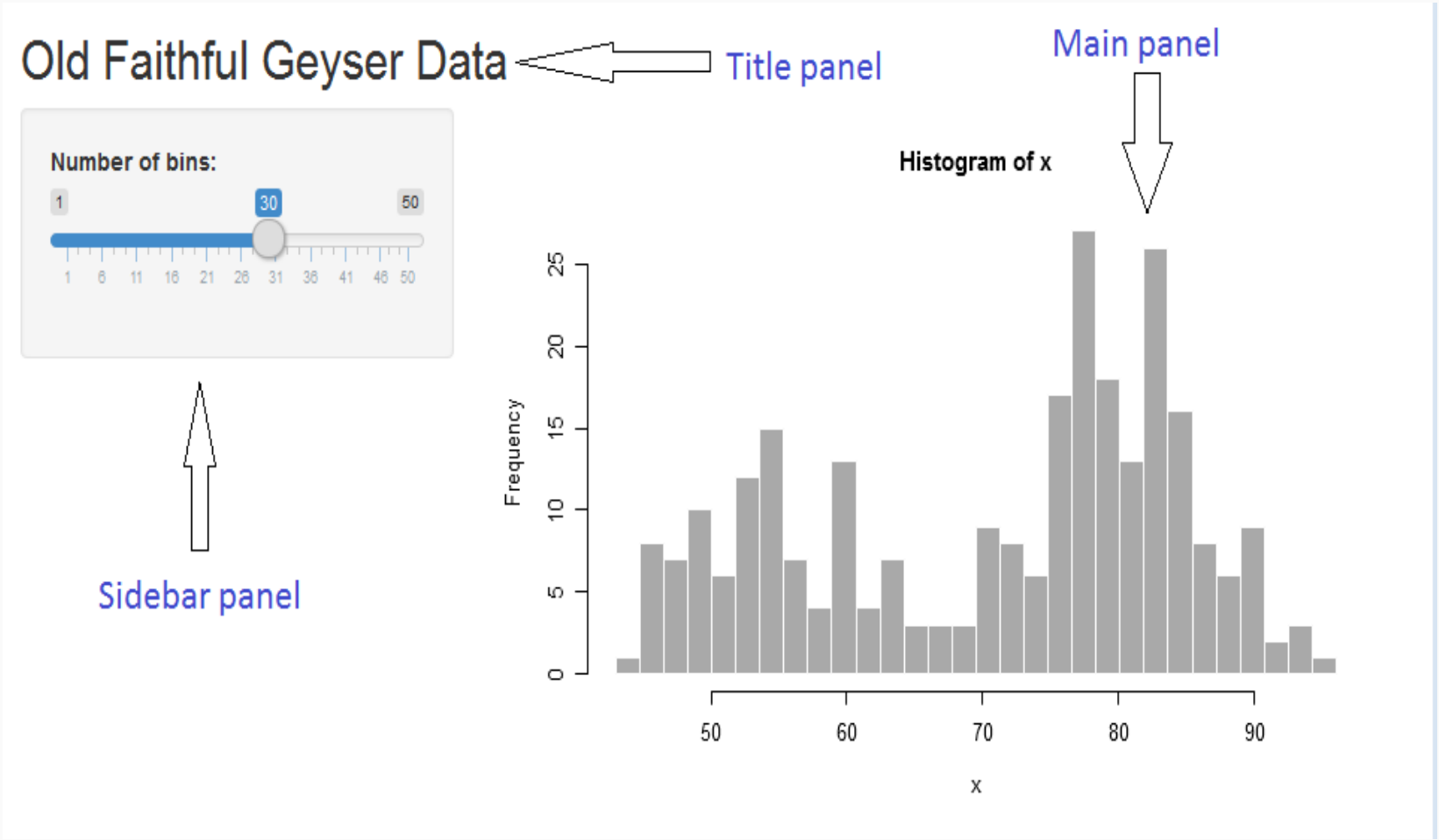
2. **Server**: this contains all the code for performing all the calculations and manipulations related to the output

```
library(shiny)

ui ← fluidPage()

server ← function(input, output, session) {
  # server code here

  # ...
}

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


BC Liquor Store prices
We found 995 options for you
Price
Filter by country
Data source: OpenDataBC
count
Alcohol_Content
Type
BEER
REFRESHMENT BEVERAGE
SPIRITS
WINE
HTML
Server code
User interface (UI)

User interface definition

```
library(shiny)

# Define UI for application that draws a histogram
shinyUI(
  fluidPage(
  # Application title
  titlePanel("Old Faithful Geyser Data"),
  # Sidebar with a slider input for number of bins
  sidebarLayout(
    sidebarPanel(
       sliderInput(inputId = "bins",
                   label = "Number of bins:",
                   min = 1,
                   max = 50,
                   value = 30)
    ),
    # Show a plot of the generated distribution
    mainPanel(
       plotOutput(outputId = "distPlot")
    )
  )
))
```


Old Faithful Geyser Data
Title panel
Main panel
Number of bins:
1
30
50
1 6 11 16 21 26 31 36 41 46 50
Sidebar panel
Histogram of x
Frequency
0
5
10
15
20
25
50
60
70
80
90
x

Define the `ui` using the `fluidPage()` function which renders an HTML file to display the components of the application

Within the `fluidPage()`, we can define :

- Title panel with `titlePanel()`
- Sidebar panel with `sidebarPanel()`
- Main panel with `mainPanel()`

There are some additional panels which can be added to sidebarPanel and mainPanel depending upon the layout and requirements of the app

Server definition

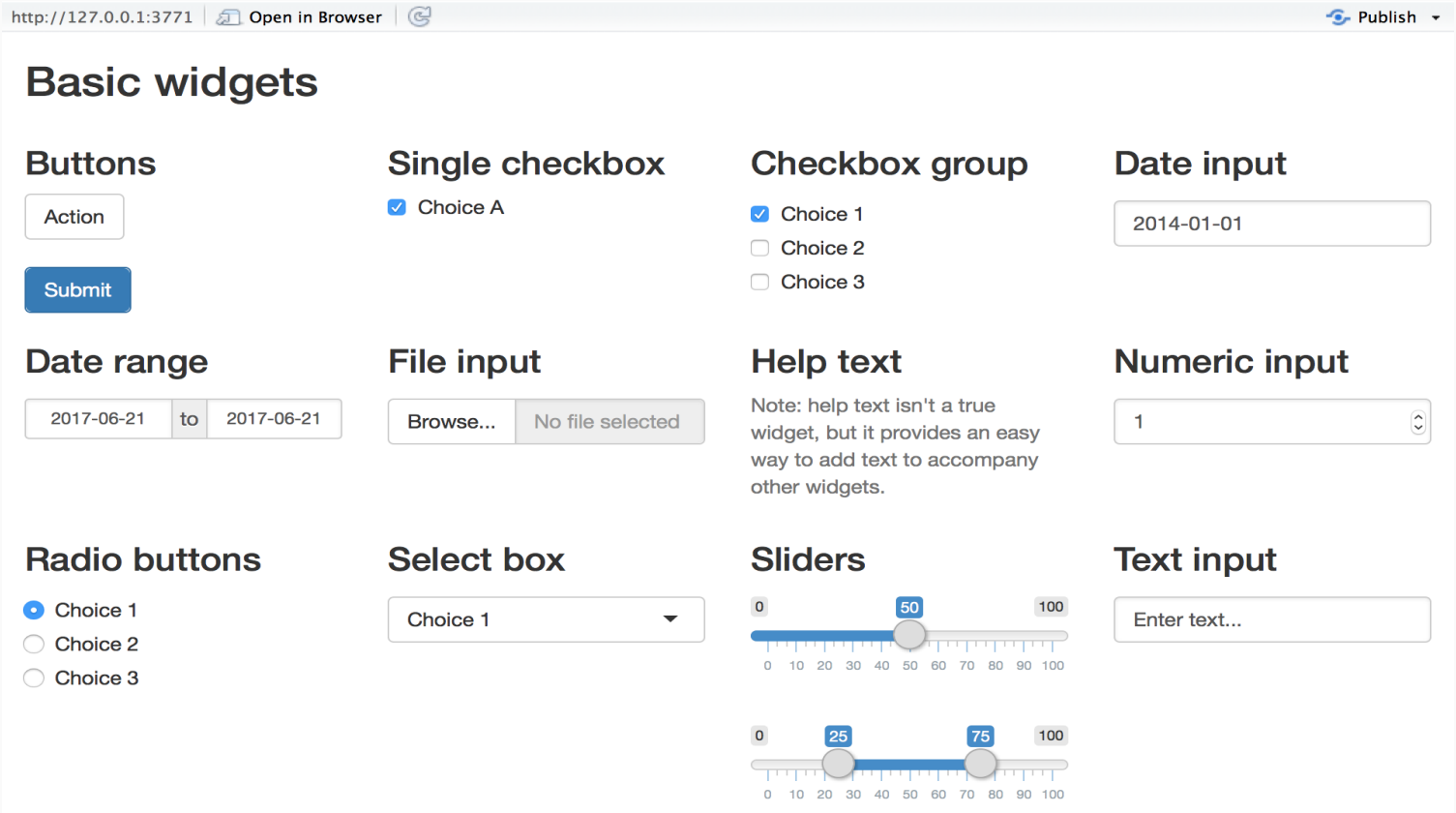
```
library(shiny)

# Define server logic required to draw a histogram
shinyServer(function(input, output) {

  output$distPlot ← renderPlot({

    # generate bins based on input$bins from ui.R
    x    ← faithful[, 2]
    bins ← seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })
})
```

The server function works off the input and output elements that were defined in the `UI`

Input options

Input options (usually) go in the `ui.R` file

Input is defined through input functions called **widgets**. These are text elements a user can interact with, like scroll bars or radio buttons



All input functions have `inputId` and `label` as the first two arguments

```
sliderInput(inputId = "bins",
            label = "Number of bins:",
            min = 1,
            max = 50,
            value = 30)
```

The server side uses `inputId` to access the value of the user input

In this example `inputId = "bins"` and on the server side we would use `input$bins` to use its value

```
output$distPlot ← renderPlot({

  # generate bins based on input$bins from ui.R
  x    ← faithful[, 2]
  bins ← seq(min(x), max(x), length.out = input$bins + 1)

  # draw the histogram with the specified number of bins
  hist(x, breaks = bins, col = 'darkgray', border = 'white')
})
```

Output options

Output options (also usually) go in the `ui.R` file

They define things like plots, tables and texts- anything **R** creates and users see

Examples: `plotOutput()`, `textOutput()`, `tableOutput()`

```
mainPanel(
  plotOutput(outputId = "distPlot")
)
})
```

Rules of server functions

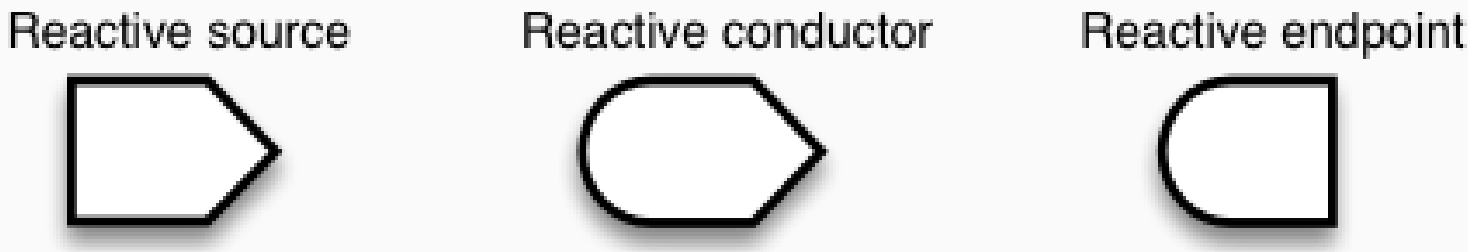
- Save objects to display to `output$<outputId>`
- Build objects to display with `render*()` function (`renderPlot()`, `renderTable()`, `renderText()`, etc)
- Use `input$<inputId>` to access value of input

Reactivity

One of the things that makes Shiny apps interactive is **reactivity**

Shiny uses **reactive programming** which gives Shiny the ability to compute outputs that **react** to changes in input from a user

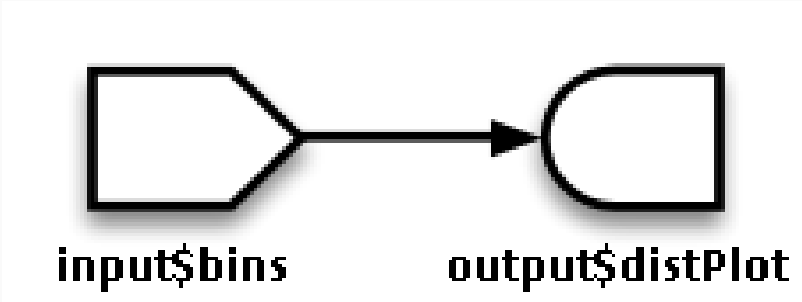
There are three types of reactive objects:



Reactive source: user input that comes through a browser interface

Reactive endpoint: something that appears in the user's browser such plot or table

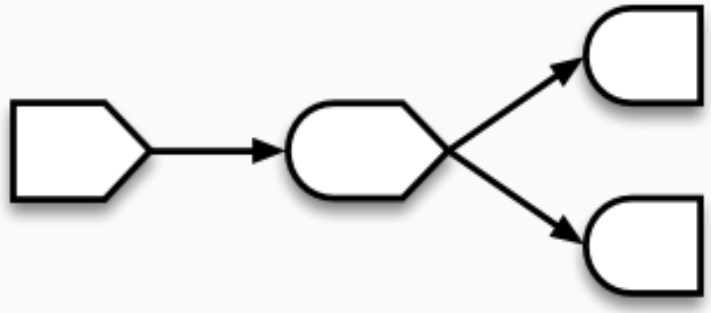
`input$<inputId>` object is the reactive source and `output$<outputId>` object is the reactive endpoint



One reactive source can be connected to multiple reactive endpoints and vice versa

Reactive conductor: reactive component between a source and an endpoint

A conductor can both be a dependent (child) and have dependents (parent)



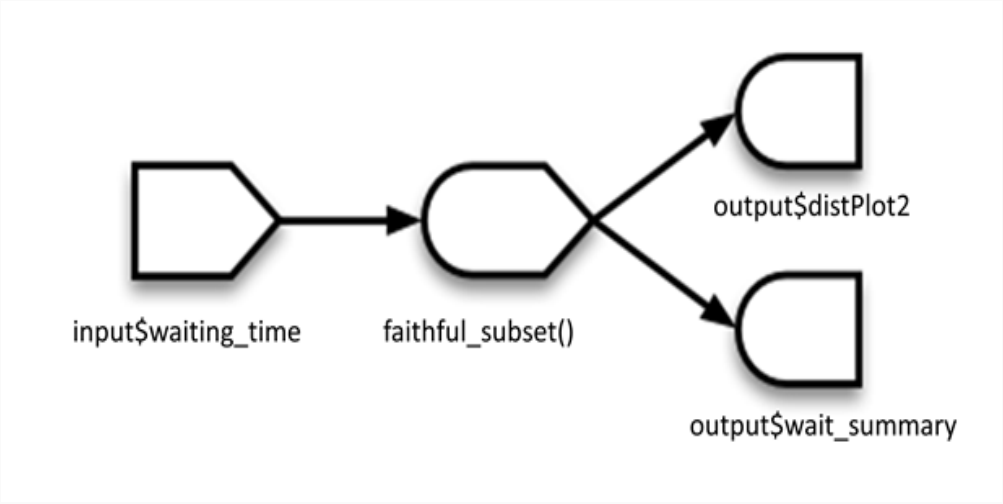
Reactive conductors can be useful for encapsulating slow or computationally expensive operations

If one source has multiple endpoints then computations will need to be done several times and reactive conducters can speed this up

`Reactive expressions` are an implementation of reactive conducters that take an `input$<inputId>` value, do some operation and cache the results

```
faithful_subset ← reactive({
  filter(faithful, waiting > input$waiting_time)
})
```

`reactive({})` creates cached expression that knows it is out of date only when input changes



- Reactive conductors let you not repeat yourself and help decompose large, complex calculations into smaller pieces
- Reactive expressions are lazy; they only get executed when their input changes and they are called by someone else
- Reactive expressions are useful for caching the results of any procedure that happens in response to user input

Accessing reactive value outside of reactive context throws error

```
server ← function(input, output, session){
  print(input$bins)
}
```

`Error : Operation not allowed without an active reactive context.`

`observe({ })` to access reactive varaible

```
server ← function(input, output, session){
  observe({
    print(input$bins)
  })
}
```

Reactives vs. observers

`reactive()`

- It **can be called** and **returns a value**, like a function
- It's **lazy** and doesn't execute its code until it is called (even if its reactive dependencies have changed), also like a function
- It's **cached**. The first time it's called, it executes the code and saves the resulting value. Subsequent calls can skip the execution and just return the value
- It's **reactive**. It is notified when its dependencies change. When that happens, it clears its cache and notifies it dependents.

```
function(input, output, session) {
  reactive({
    # This code will never execute!
    cat("The value of input$x is now ", input$x, "\n")
  })
}
```

To access reactive variable, we need to add `()` i.e. `x()`

`observe()`

- It **can't be called** and **doesn't return a value**
- It **eagerly respond** to changes their dependencies. When its dependencies change, it executes right away
- Since it can't be called and doesn't have a return value, there's no notion of caching that applies here
- It's **reactive**. It is notified when its dependencies change, and when that happens it executes

```
function(input, output, session) {

  # Executes immediately, and repeats whenever input$x changes
  observe({
    cat("The value of input$x is now ", input$x, "\n")
  })
}
```

Again note that you cannot assign variables from `observe()`

Most importantly:

- `reactive()` is for calculating values, without side effects
- `observe()` is for performing actions, with side effects

`reactive()`	**`observe()`**
Callable	Not callable
Returns a value	No return value
Lazy	Eager
Cached	N/A

Isolating reactions

Use `isolate()` to wrap an expression to block reactivity

It prevents the execution of a piece of code unless, of course, certain condition is met

```
# Updates every time input$x or input$y change
r1 ← reactive({
  input$x * input$y
})

# Updates only when input$x changes
r2 ← reactive({
  input$x * isolate({input$y})
})

# Never updates; it will always have its original value
r3 ← reactive({
  isolate({input$x * input$y})
})
```

The condition that `isolate()` takes in order to re-execute the piece of code is action button

Triggering reactions

`observeEvent(eventExpr, handlerExpr, ...)` is used to trigger a reaction

It just depends on specific reactive value/expression and ignore all others (“event handler”)

```
# ui
actionButton(inputId = "save_button", label = "Save CSV")

# server
function(input, output, session) {

  # only executes when input$save_button is pushed
  observeEvent(input$save_button, {
    write.csv(df(), "data.csv")
  })
}
```

Delaying reactions

`eventReactive(eventExpr, handlerExpr, ...)` is used for delayed compuation

```
# ui
actionButton(inputId = "go", label = "Get samples")

# server
function(input, output, session) {

  rv ← reactiveValues(data = rnom(50))

  observeEvent(input$go, {
    rv$data ← rnorm(input$num)
  })
}
```

observeEvent() vs eventReactive()

- `observeEvent()` is used to perform an action in response to an event
- `eventReactive()` is used to create a calculated value that only updates in response to an event

`observe()` and `reactive()` functions automatically trigger on whatever they access but `observeEvent()` and `eventReactive()` functions need to be explicitly told what triggers them

Shiny modules

A self-contained, composable component of a Shiny app

- self-contained like a function
- can be combined to make an app

Why use modules?

- Reuse
- Isolate

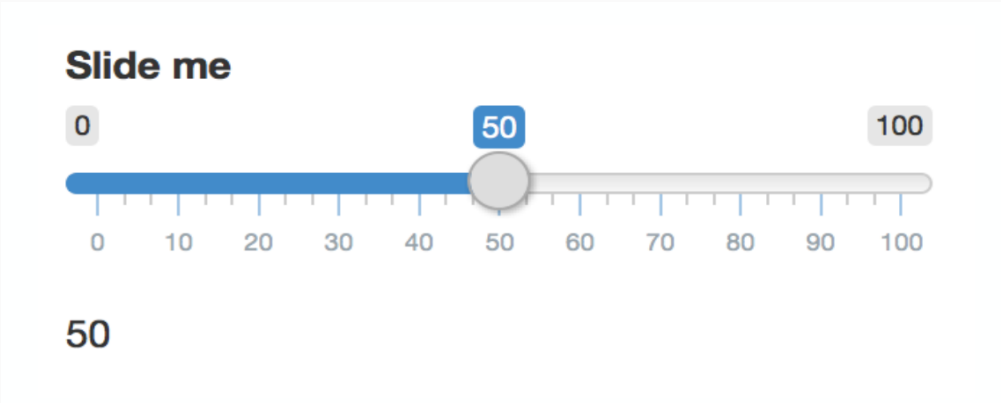
Useful for managing code complexity in larger apps

Anatomy of Shiny module

A pattern of code organized into two functions

- A function that creates UI elements
- A function that loads server logic

```
library(shiny)

name_of_module_UI ← function(){ #UI }

name_of_module ← function(){ #Server logic }
```




```
sliderTextUI ← function(){ #UI }

sliderText ← function(){ #Server logic }
```

Demo[1]

[1] slider_ex1-5

Module UI

Task 1 - Return Shiny UI

```
sliderTextUI ← function(sliderId, textId){

  tagList(
    sliderInput(sliderId, "Slide Me", 0, 100, 1),
    textOutput(textId)
  )
}
```

Module UI

Task 1 - Return Shiny UI

```
sliderTextUI ← function(sliderId, textId){

  tagList(
    sliderInput(sliderId, "Slide Me", 0, 100, 1),
    textOutput(textId)
  )
}
```

Task 2 - Assign module elements to a unique namesapce with `NS()`

namespace is a system for organizing objects with identical names

```
NS("Hello")
```

```
## function (id)
## {
##     if (length(id) == 0)
##         return(ns_prefix)
##     if (length(ns_prefix) == 0)
##         return(id)
##     paste(ns_prefix, id, sep = ns.sep)
## }
## <bytecode: 0x00000000155aeb50>
## <environment: 0x000000001583df40>
```

```
ns_fun ← NS("Hello")

ns_fun("World")
```

```
## [1] "Hello-World"
```

1. Add an id argument

2. Make a namespace function

3. Wrap all input and output IDs with namespace function

```
sliderTextUI ← function(id){ # add an id arg

  ns ← NS(id) # make a namesapce function

  tagList(
    sliderInput(ns("slider"), # wrap all input & output IDs with ns()
                "Slide Me", 0, 100, 1),
    textOutput(ns("number"))
  )
}
```

Module server

Handles the server logic for the module

```
sliderText ← function(input, output, session){
  output$num ← renderText({
    input$slider
    })
}
```

1. You must use all 3 arguments: input, output, session

2. Do not use `ns()` to refer to inputs and outputs from the module

Load the module server function in the app's server function with `callModule()`[1]

```
ui ← fluidPage(
  sliderTextUI("one")
)

server ← function(input, output){
  callModule(sliderText, "one")
}

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

- First argument of `callModule()` is module function
- Second argument is the namespace Id that is the same Id as UI module

[1] Demo: slider_ex6

Where to define the module functions?

- In the preamble of a single file app (app.R)
- In a file that is sourced in the preamble of a single file app
- In global.R
- In a file sourced by global.R
- In a package that the app loads

Passing reactive input to a module

Reactive expressions are the most portable format for passing reactive information between functions[1]

```
sliderText ← function(input, output,
                      session, show){
  output$number ← renderText({
    if (show()) input$slider # 3.
    else NULL
  })
}

ui ← fluidPage(
  checkboxInput("display", "Show Value")
  sliderTextUI("module")
)

server ← function(input, output) {
display ← reactive({input$display}) #1.
 callModule(sliderText, "module",
            display) # 2.
}
shinyApp(ui, server)
```

1. Wrap the input as a reactive expression
2. Pass the reactive expression, not the value, to the module, i.e do NOT use `()`
3. Treat the argument as a reactive expression within the module, i.e. do use `()`

[1] Demo: slider_ex7

Returning reactive output from a

```
sliderText ← function(input, output, se

 output$num ← renderText({input$slider}

reactive({input$slider}) # 1.
}

ui ← fluidPage(
 sliderTextUI("module"),
 h2(textOutput("value"))
)

server ← function(input, output) {

num ← callModule(sliderText, "module")

output$value ← renderText({num()}) # 2.
}

shinyApp(ui, server)
```

1. Return reactive output as a reactive expression or a list of reactive expressions[1]
2. Call value as a reactive expression, i.e. with `()`

[1] Demo: slider_ex8