DATA SOCIETY®

R Shiny - day 2

“One should look for what is and not what he thinks should be.”
-Albert Einstein.

Module completion checklist

Objective	Complete
Introduce the concept of reactivity for interactive dashboards	
Summarize different objects in reactive programming with their flow and implementation	
Implement a simple reactive plot and a table in Shiny	
Build a Shiny app with reactive expressions	
Configure the app to isolate part of the app from regenerating	
Build a Shiny app to respond to actions using observeEvent()	
Configure the app to delay the response to events	
Outline different layouts to organize the Shiny widgets	
Host the app on the cloud-based service from RStudio	

Reactivity: a brief introduction

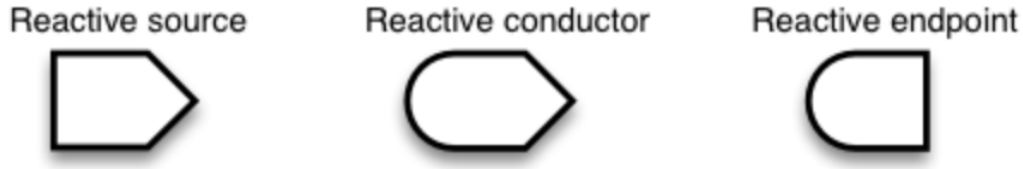
- In the previous class, we built a simple Shiny app with **static content** using different input and output widgets
- Today, we will build **interactive Shiny apps**
- We need to understand the concept of reactivity in order to build interactive dashboards

What is reactivity?

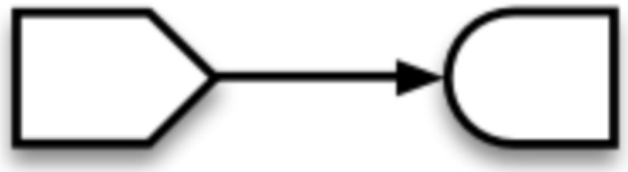
- Reactivity makes the R Shiny apps **responsive**
- It lets the app **update itself** whenever the user makes a change in the input
- With reactivity, we can:
  - create more **sophisticated and efficient Shiny apps**
  - avoid errors that come from misusing reactive values
- For every input the user changes, we get an updated output

Objects in reactive programming

- There are three main objects in reactive programming
  - **reactive sources:** it can only be a parent
  - **reactive conductors:** it can be both a parent and a child
  - **reactive endpoints:** it can only be a child



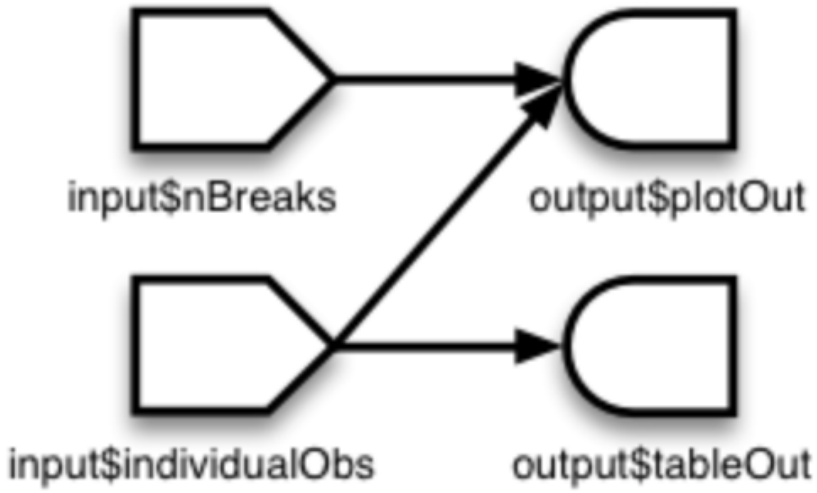
Reactive flow - 1



- The simplest reactive flow has **one source and one endpoint**

- The source is the user's input through the browser interface and it is usually taken from the `ui`
- It can be the user selecting an item, by clicking on the button or selecting radio buttons
- Reactive sources are accessible through the `input` object and endpoints are accessible through the `output` object
- The reactive endpoint usually lies in the `server` part and appears on the browser as a plot or table of values

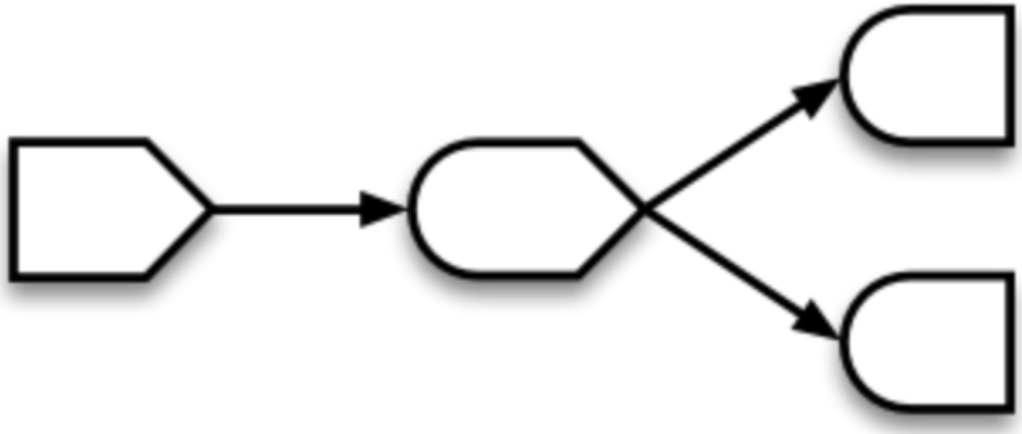
Reactive flow - 2



- A single source can be connected to **more than one endpoint**

- In this flow, whenever the value of `input$nBreaks` changes, the expression that generates the plot also automatically re-executes
- Whenever the value of `input$individualObs` changes, the plot and the table functions will automatically re-execute

Reactive flow - 3



- In this process, we have a reactive conductor in the flow which is in between the sources and the endpoints
- Reactive conductors are for encapsulating any reactive operations while the user's input changes every time

Implementation of reactive values

- We saw the three types of reactive objects
- Let's understand how to implement the objects in the R Shiny dashboard
- **Reactive values**
  - They are the implementation of **reactive sources**
  - They contain values that can be read by other reactive objects
  - The `input` object is a `ReactiveValues` object, which is a list of reactive values
  - The values in `input` are set by input from web browser
  - We can only call a reactive value from a function that is designed to work with it

Implementation of reactive expressions

- Reactive expressions are the implementation of the **reactive conductors**
- They can access reactive values or other reactive expressions and they return a value
- A reactive expression can be used for:
  - Accessing a database
  - Reading data from file
  - Downloading data
  - Performing expensive computation

Implementation of observers

- Observers are the implementation of **reactive endpoints**
- They can access reactive sources and reactive expressions and they do not return a value
- Observers send the data to the web browser for the users to view
- The output object is like a list of individual observers

Module completion checklist

Objective	Complete
Introduce the concept of reactivity for interactive dashboards	✔
Summarize different objects in reactive programming with their flow and implementation	✔
Implement a simple reactive plot and a table in Shiny	
Build a Shiny app with reactive expressions	
Configure the app to isolate part of the app from regenerating	
Build a Shiny app to respond to actions using observeEvent()	
Configure the app to delay the response to events	
Outline different layouts to organize the Shiny widgets	
Host the app on the cloud-based service from RStudio	

Reactive apps

- We already saw how to render a plot on the R Shiny app
- We will now render a reactive plot and a reactive table
  - **Plot:** get the input variable and number of bins from the user and create a histogram of the variable
  - **Table:** print the head or tail of `n` observations of the first 10 columns

Setting up Shiny apps for the day

- Always remember to first set up the app through the following steps before the actual coding
  - Set the folder structure for the new app
  - Create `server.R` and `ui.R` scripts with base structure
  - Set the working directory
  - Load the `shiny` package
  - Load the required dataset

- We have a new folder within the `shiny` folder with the name `day-2`
- `1-histogram` folder within the `day-2` folder is our first app today
- Find two `R` scripts with name as `server.R` and `ui.R`
- We will be using `CMP` dataset today

Base structure of server of the app

- The following is the base structure in `server.R`

```
#### Set working directory ####
session_info = sessionInfo()

platform = session_info$platform
directory = "af-werx"

if(length(grep("linux|apple", platform)) > 0){
  Sys.getenv("USER")
  dir = paste0("~/Desktop/", directory)
}else{
  username = Sys.getenv("USERNAME")
  dir=paste0("C:/Users/", username, "/Desktop/", directory)
}

# Set the working directory to data directory.
data_dir = paste0(dir, "/data")
setwd(data_dir)

# Load the shiny package.
library(shiny)

# Load the CMP dataset.
CMP = read.csv("ChemicalManufacturingProcess.csv", header = TRUE, stringsAsFactors = FALSE)

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

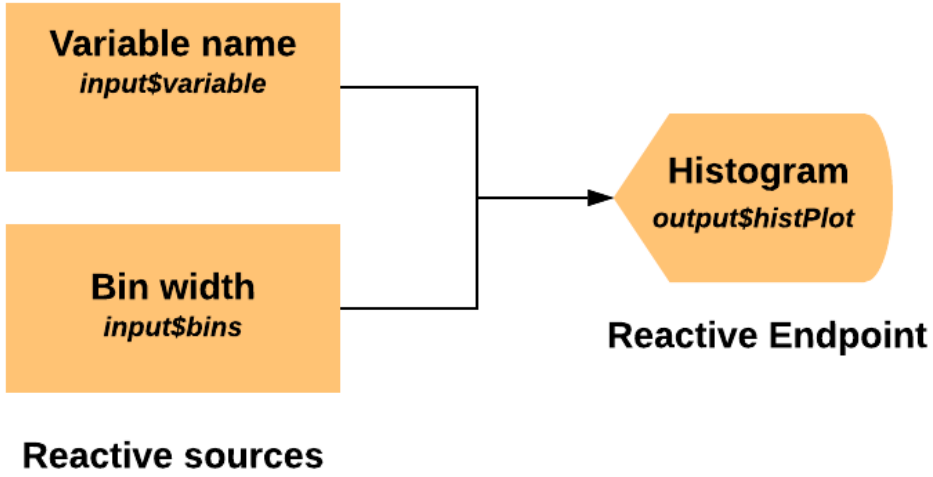
Base structure of ui of the app

- The following is the base structure in `ui.R`

```
#### Set working directory ####
session_info = sessionInfo()

platform = session_info$platform
directory = "af-werx"

if(length(grep("linux|apple", platform)) > 0){
  Sys.getenv("USER")
  dir = paste0("~/Desktop/", directory)
}else{
  username = Sys.getenv("USERNAME")
  dir=paste0("C:/Users/", username, "/Desktop/", directory)
}

# Set the working directory to data directory.
data_dir = paste0(dir, "/data")
setwd(data_dir)

# Load the shiny package.
library(shiny)

# Load the CMP dataset.
CMP = read.csv("ChemicalManufacturingProcess.csv", header = TRUE, stringsAsFactors = FALSE)


ui <- fluidPage(
)
```

Build a reactive histogram of the given variable

- We want to create a histogram of any column of the dataframe chosen by the user
- Within the UI, we will create two inputs
  - `selectInput` which has the column names of the `CMP` dataframe
  - `sliderInput` which gets the bin width value from the user
- We will also create a title panel to give the title to the app
- The output is a plot output which is rendered by `plotOutput` function in `ui` and `renderPlot` function in `server`



Build a reactive histogram of the given variable

- Adding the ui part of the app in the `ui.R` script

```
ui <- fluidPage(

  # Title of the app.
  titlePanel("Histogram"),

  # Create a slider input.
  sliderInput(inputId = "bins",  #<- input id
              label = "Number of bins:", #<- input label
              min = 1,  #<- min number in slider
              max = 50, #<- max number in slider
              value = 30), #<- default value in slider

  # Select input from a list of inputs.
  selectInput(inputId = "variable",  #<- input ID
              label = "Choose the variable:", #<- input label
              choices = as.list(names(CMP)), #<- input list
              selected = names(CMP)[1]),  #<- default value of input

  # Render the output as plot.
  plotOutput(outputId = "histPlot")
)
```

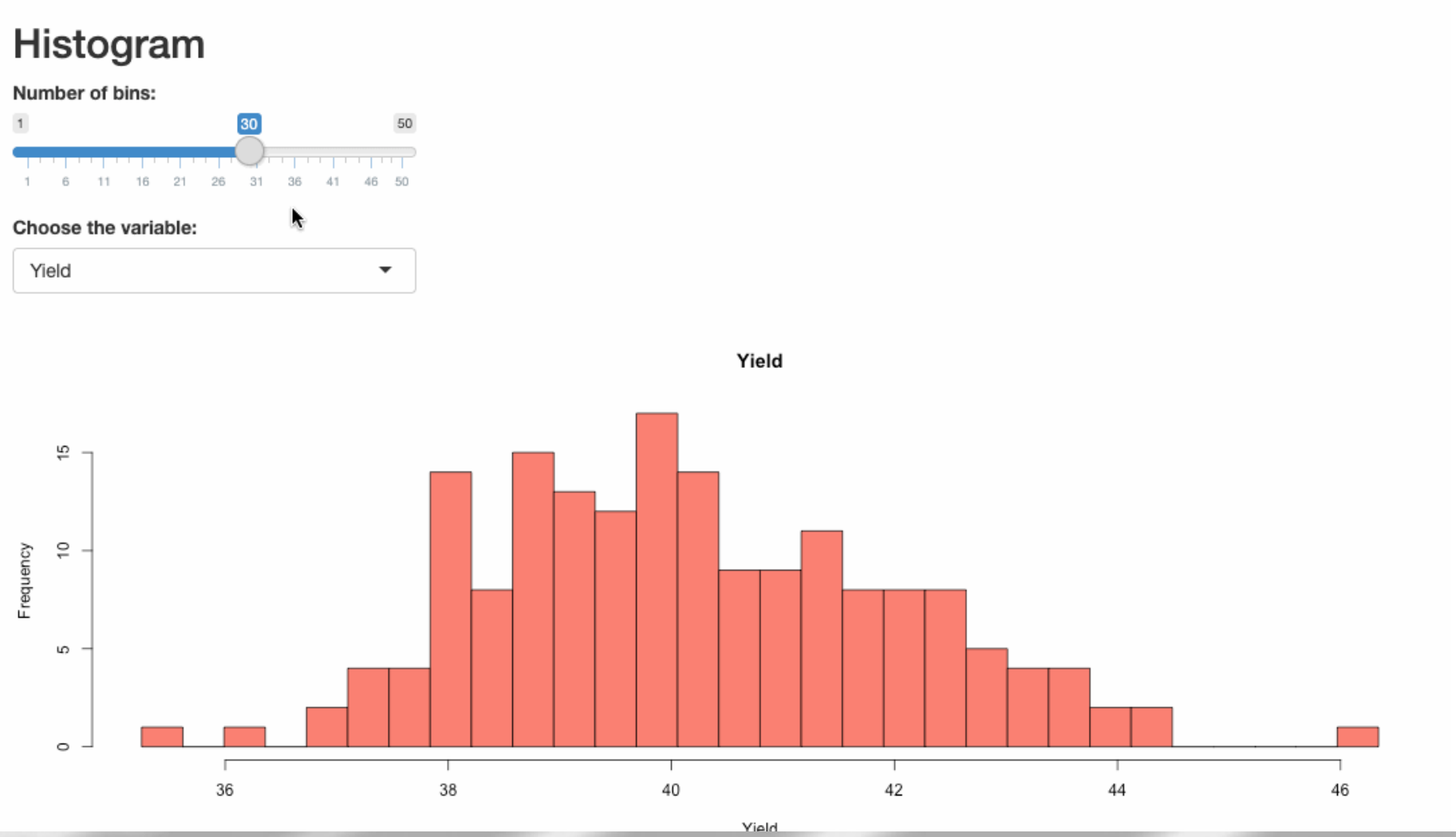
Build a reactive histogram of the given variable

- Adding the server part of the app in the `server.R` script

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

  # Create a renderPlot function.
  output$histPlot <- renderPlot({

    # Get the bin width from user and adjust the bin size.
    bins <- seq(min(CMP[[input$variable]]),  #<- min bin value is min value of variable chosen
                max(CMP[[input$variable]]), #<- max bin value is max value of variable chosen
                length.out = input$bins + 1) #<- length is value chosen by the user


    # Plot the histogram.
    hist(CMP[[input$variable]], #<- variable name chosen by user
         breaks = bins,         #<- bin width
         col = "salmon",        #<- color of the histogram
         main = input$variable, #<- title of the histogram
         xlab = input$variable) #<- x axis label

  })
}
```

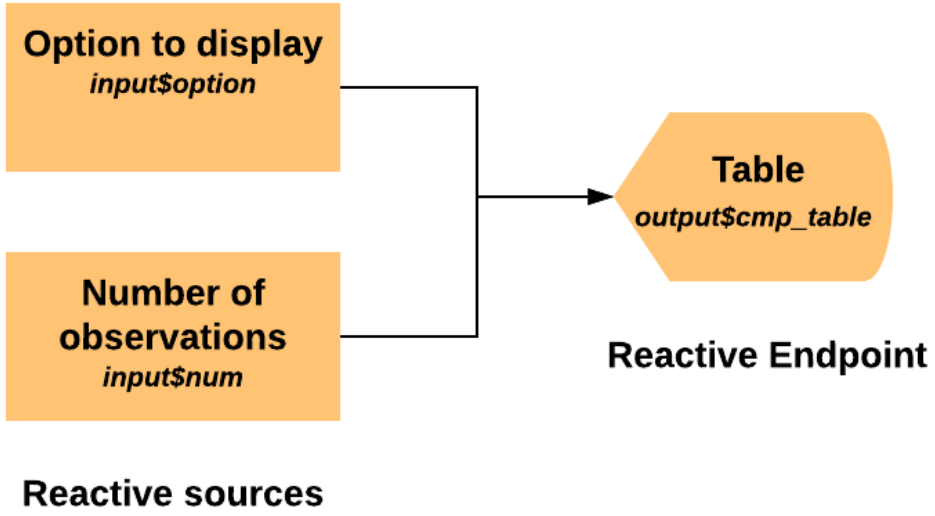
Build a reactive histogram of the given variable

- Click on `Run App` from the app script



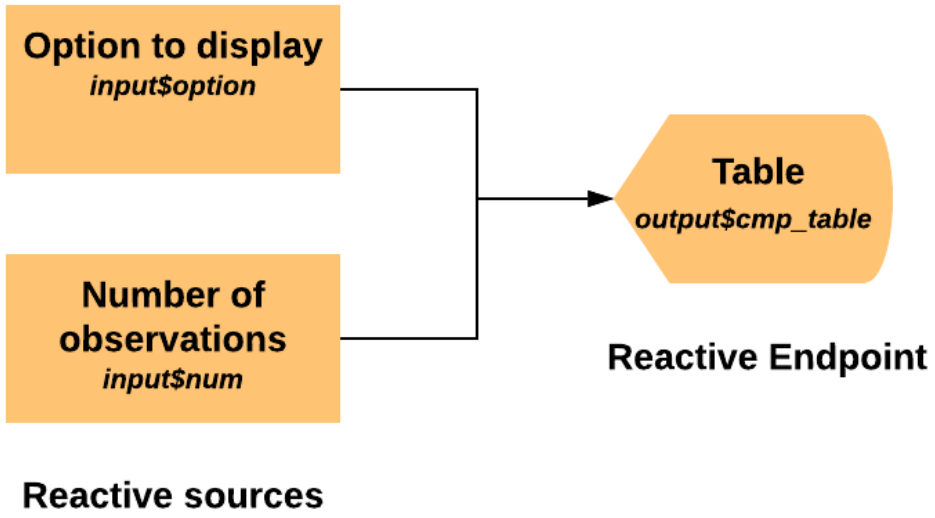
Build a reactive table

- We want to display head or tail of `n` observations of the first 10 variables of `CMP` dataset as a table
- Find the subfolder in `day-2` with name `2-table`
- Find the two `R` scripts named as `server.R` and `ui.R` that also have the base structure
- We will be using `CMP` dataset for this app



Build a reactive table

- Within the UI, we will create two inputs:
  - `radioButtons` which has the option to choose head or tail
  - `numericInput` which gets the number of observations to display
- We will also create a title panel to give the title to the app
- The output is a table output which is rendered by `tableOutput` function in `ui` and `renderTable` function in `server`



Build a reactive table of the given variable

- Adding the ui part of the app in the `ui.R` script

```
ui <- fluidPage(

  # Title of the app.
  titlePanel("Table"),

  # Radio button to choose the option.
  radioButtons("option",             #<- input ID
               "Choose the option", #<- input label
               c("Head", "Tail")),  #<- options

  # Numeric inputs.
  numericInput("num",                 #<- input ID
               "No of observations", #<- input label
               value = 10,           #<- default value
               min = 1,              #<- minimum value
               max = nrow(CMP),      #<- maximum value
               step = 1),            #<- increase in step of

  # Ouput function to display.
  tableOutput("cmp_table")
)
```

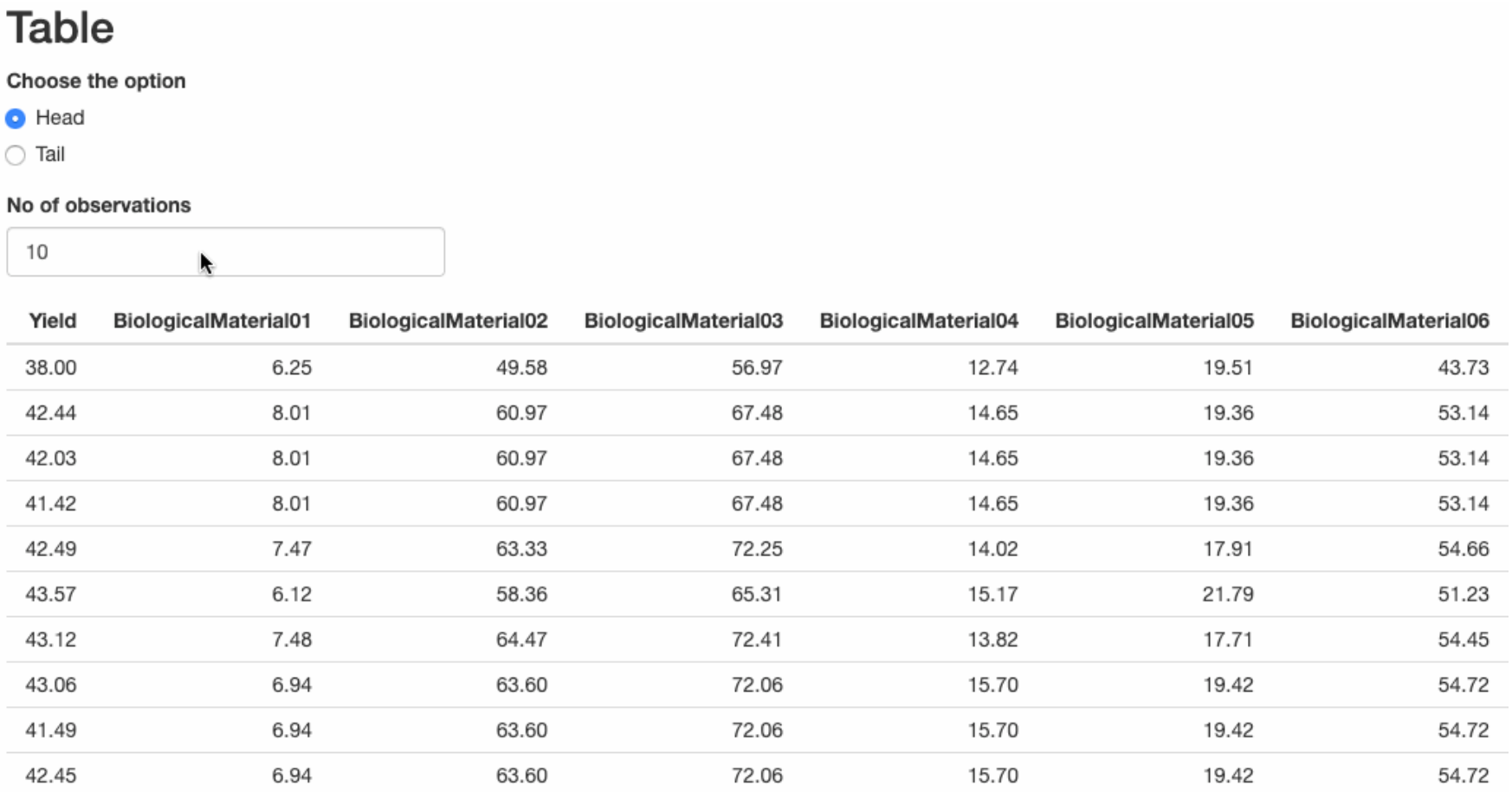
Build a reactive table of the given variable

- Adding the server part of the app in the `server.R` script

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

# Render output table.
output$cmp_table = renderTable(

  # Check input option to display head or tail.
  if(input$option == 'Head'){
    head(CMP[1:10], input$num) #<- display head of the data
  }

  else{
    tail(CMP[1:10], input$num) #<- display tail of the data
  }

)
}
```

Build a reactive table of the given variable

- Click on `Run App` from the app script



Yield	BiologicalMaterial01	BiologicalMaterial02	BiologicalMaterial03	BiologicalMaterial04	BiologicalMaterial05	BiologicalMaterial06
38.00	6.25	49.58	56.97	12.74	19.51	43.73
42.44	8.01	60.97	67.48	14.65	19.36	53.14
42.03	8.01	60.97	67.48	14.65	19.36	53.14
41.42	8.01	60.97	67.48	14.65	19.36	53.14
42.49	7.47	63.33	72.25	14.02	17.91	54.66
43.57	6.12	58.36	65.31	15.17	21.79	51.23
43.12	7.48	64.47	72.41	13.82	17.71	54.45
43.06	6.94	63.60	72.06	15.70	19.42	54.72
41.49	6.94	63.60	72.06	15.70	19.42	54.72
42.45	6.94	63.60	72.06	15.70	19.42	54.72

Knowledge check 1

Exercise 1

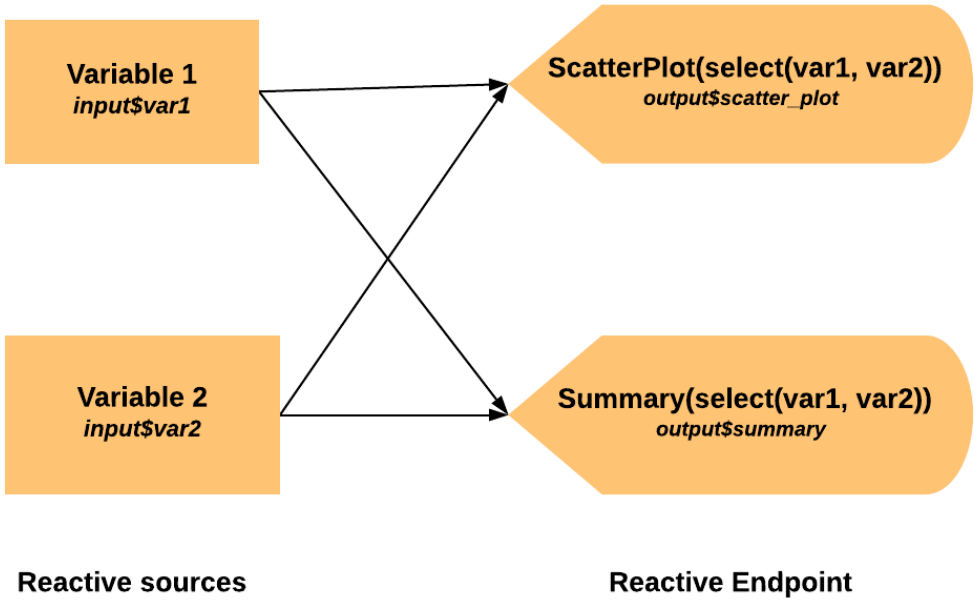
Module completion checklist

Objective	Complete
Introduce the concept of reactivity for interactive dashboards	✔
Summarize different objects in reactive programming with their flow and implementation	✔
Implement a simple reactive plot and a table in Shiny	✔
Build a Shiny app with reactive expressions	
Configure the app to isolate part of the app from regenerating	
Build a Shiny app to respond to actions using observeEvent()	
Configure the app to delay the response to events	
Outline different layouts to organize the Shiny widgets	
Host the app on the cloud-based service from RStudio	

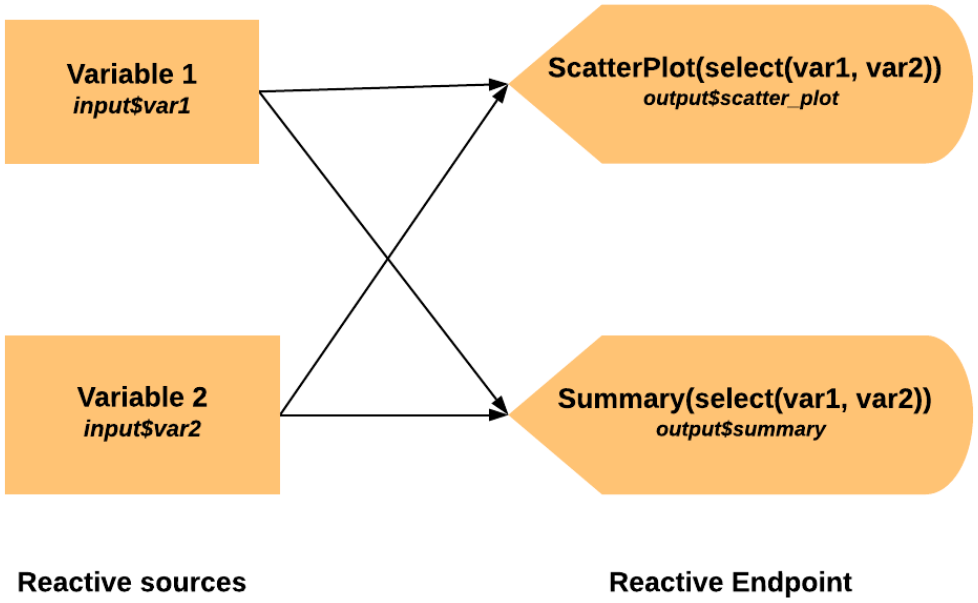
Reactive expressions

- Until now, we saw how to create apps with reactive sources and reactive endpoints
- Now, we will learn the usage of reactive expressions
- Reactive expressions exist to avoid **too much computation**
- Let's say we want to get two variables as the input from the user, save it to a new dataframe and produce a scatterplot and a summary of these two variables

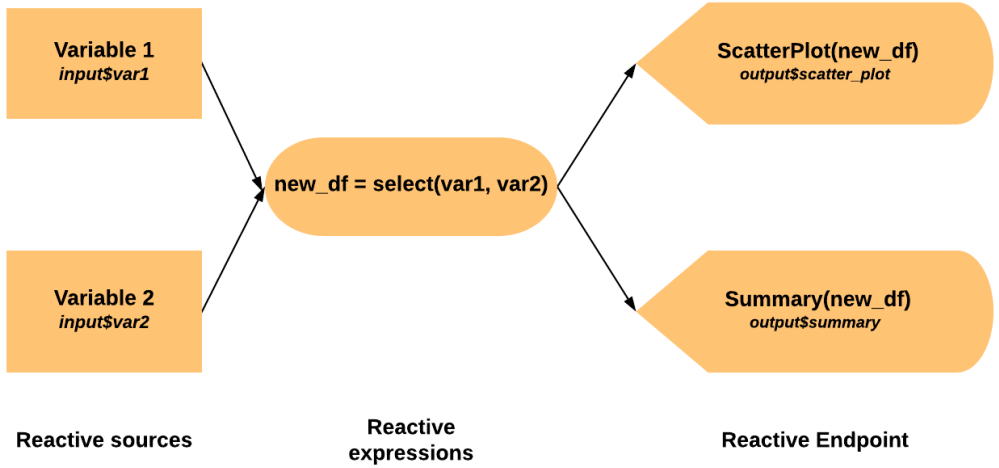
- The flow in general will be:



Reactive expression flow



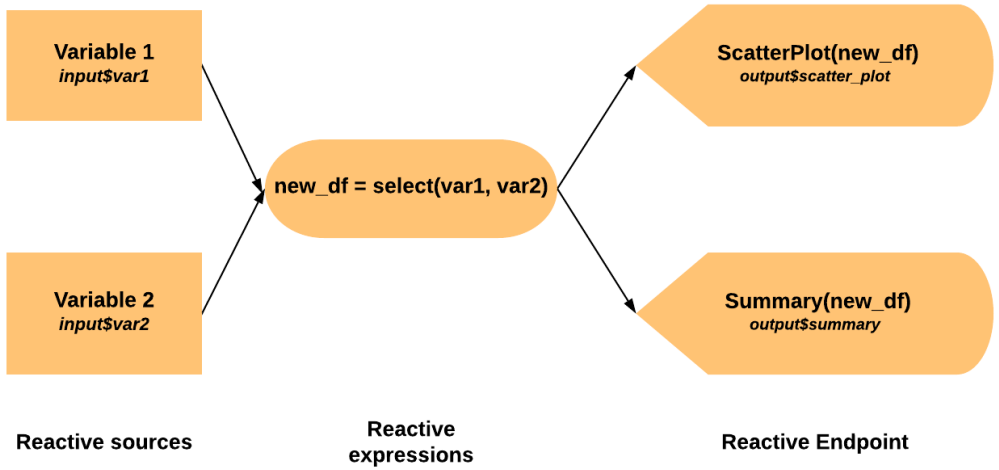
- In the general flow, we select both the input variables twice - once for the scatterplot and another for the summary
- This repetitive computation can be avoided if we use reactive expressions



- We can select both the input variables to a new dataframe in a reactive expression and then pass to the endpoints for plot and summary output
- By using this flow, computation in the app decreases by using reactive expressions

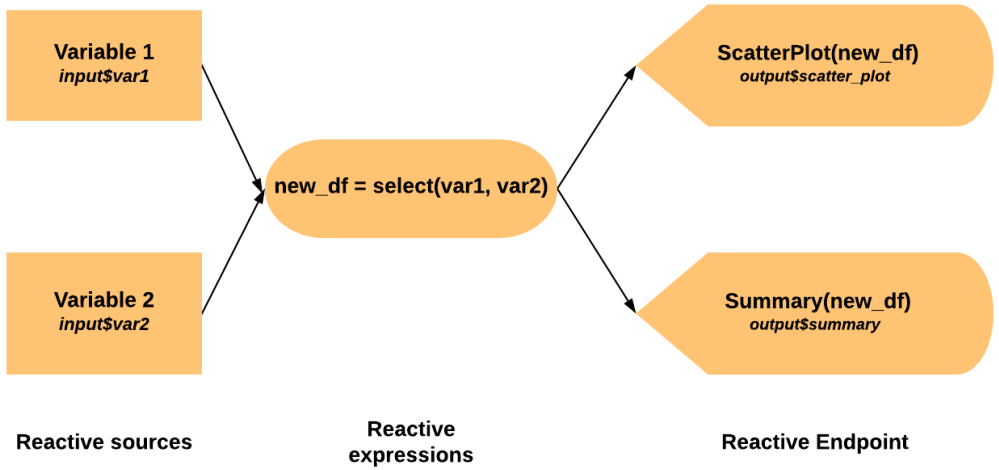
Build an app using reactive expression

- We want to display the scatterplot and summary of two variables chosen by the user
- In the `day-2` folder, find the subfolder with name `3-scatter_and_summary`
- There are two R scripts - `server.R` and `ui.R` with base structure
- We will be using `CMP` dataset today
- Within the UI, we will create two inputs which are `selectInput` type to choose two variables from a list of columns of CMP dataset



Build an app using reactive expression

- Let's create a title panel to give the title to the app
- We will create a reactive expression in the `server.R` to select two variables chosen by the user
- There are two outputs:
  - a plot output which is rendered by `plotOutput` function in `ui` and `renderPlot` function in `server`
  - a summary of two variables rendered by `verbatimTextOutput` function in `ui` and `renderPrint` function in `server`



Build an app using reactive expression

- Adding the ui part of the app in the `ui.R` script

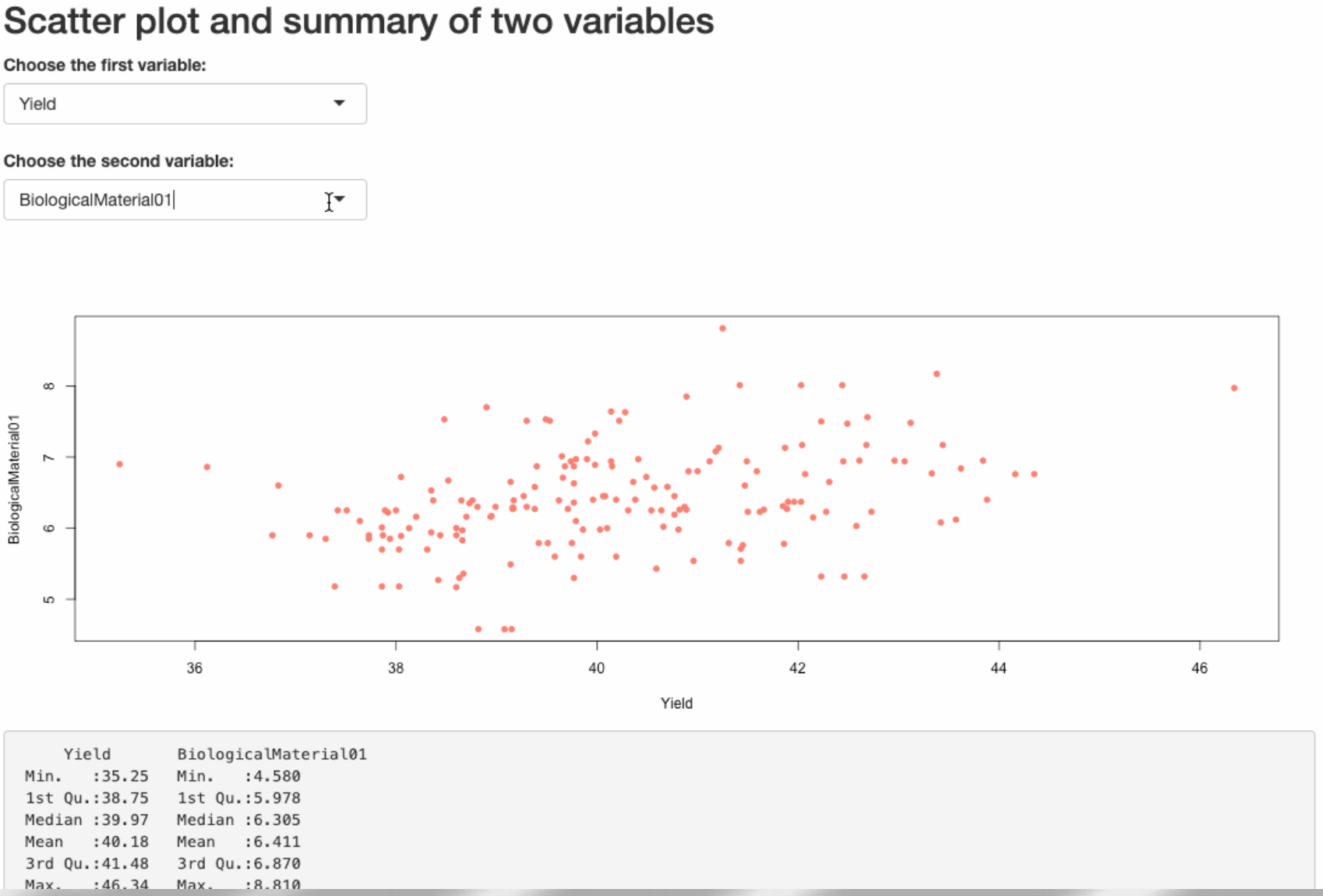
```
ui <- fluidPage(

  # Title of the app.
  titlePanel("Scatterplot and summary of two variables"),

  # Select input from a list of inputs.
  selectInput(inputId = "variable1",    #<- input1 id
              label = "Choose the first variable:", #<- input1 label
              choices = as.list(names(CMP)), #<- input1 list
              selected = names(CMP)[1]),  #<- default value of input1

  # Select input from a list of inputs.
  selectInput(inputId = "variable2",  #<- input2 id
              label = "Choose the second variable:", #<- input2 label
              choices = as.list(names(CMP)), #<- input2 list
              selected = names(CMP)[2]),  #<- default value of input2

  # Render the output as plot.
  plotOutput(outputId = "scatterPlot"),

  # Print the summary as text output.
  verbatimTextOutput("summary")
)
```

Build an app using reactive expression

- Adding the server part of the app in the `server.R` script

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

  # Create a reactive expression to select two variables chosen by user.
  data <- reactive({
    CMP %>% select(input$variable1, input$variable2)
  })

  # Print output scatterplot.
  output$scatterPlot <- renderPlot({

    plot(data(),           #<- the new data selected
         col = "salmon",   #<- color of the plot
         pch = 16)         #<- plot with filled circles

  })

  # Print the summary output.
  output$summary <- renderPrint({

    summary(data())      #<- print the summary of data

  })

}
```

Build an app using reactive expression



Module completion checklist

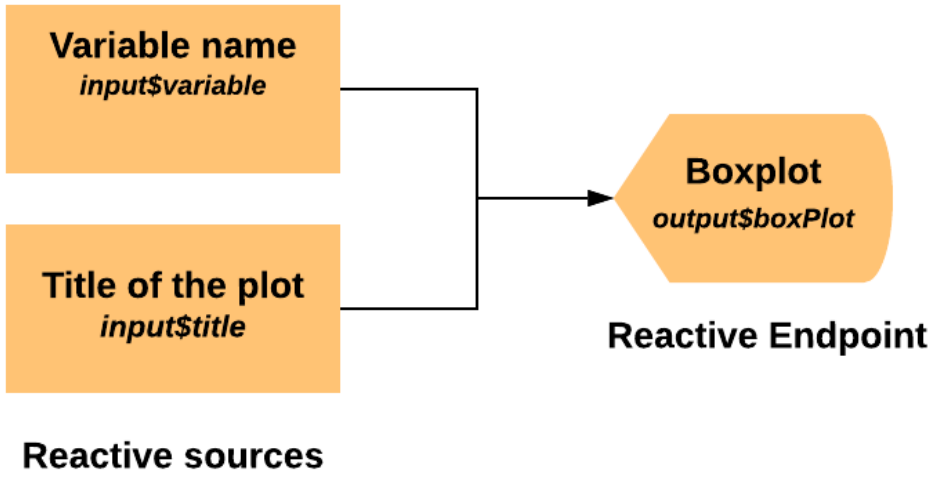
Objective	Complete
Introduce the concept of reactivity for interactive dashboards	✔
Summarize different objects in reactive programming with their flow and implementation	✔
Implement a simple reactive plot and a table in Shiny	✔
Build a Shiny app with reactive expressions	✔
Configure the app to isolate part of the app from regenerating	
Build a Shiny app to respond to actions using observeEvent()	
Configure the app to delay the response to events	
Outline different layouts to organize the Shiny widgets	
Host the app on the cloud-based service from RStudio	

Configure the app to isolate part of the app

- Reactive apps regenerate and run the entire code every time the user input is changed
- **What if we don't want the entire app to regenerate every time?**
- We'd want to **isolate** the part of the app from regenerating
- We can use the `isolate()` function to avoid re-rendering the app every time
- Let's create an app to view a boxplot of chosen variable and also get the title from the user
- We will get the variable name from the user and we want the portion of it to re-render every time the user changes the input
- But we do not want the title to re-render every time user changes the variable name
- Our end result will have the plot change every time a new input is given (both variable and title), but title will only change when both inputs change

rBuild a boxplot of the given variable

- We want to create a boxplot of the any column of the dataframe chosen by the user
- In the folder `day-2`, find the subfolder `4-boxplot` subfolder containing our app
- Within the UI, we will create two inputs:
  - `selectInput` which has the column names of the `CMP` dataframe
  - `textInput` which gets the title of the plot from the user
- We will also create a title panel to give the title to the app
- The output is a plot output which is rendered by `plotOutput` function in `ui` and `renderPlot` function in `server`



Build a boxplot of the given variable

- Adding the ui part of the app in the `ui.R` script

```
ui <- fluidPage(

  # Title of the app.
  titlePanel("Boxplot of the chosen variable"),

  # Select input from a list of inputs.
  selectInput(inputId = "variable",    #<- input id
              label = "Choose the first variable:", #<- input label
              choices = as.list(names(CMP)), #<- input list
              selected = names(CMP)[1]),  #<- default value of input

  # Text input to get the title of the plot.
  textInput(inputId = "title",       #<- title input id
            label = "write a title", #<- input label
            value = "boxplot"),      #<- default value

  # Render the output as plot.
  plotOutput(outputId = "boxPlot")

)
```

Build a boxplot of the given variable

- Adding the server part of the app in the `server.R` script

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

  # Print output scatterplot.
  output$boxPlot <- renderPlot({

    boxplot(CMP[input$variable],         #<- variable name
            main = isolate(input$title),#<- title name
            col = "aquamarine3")         #<- color of the plot

  })


}
```

Build a boxplot of the given variable

- Click on `Run App` from the app script
- See that the title does not change until the new variable input is given, hence the title is isolated from regeneration

Knowledge check 2

Exercise 2

Module completion checklist

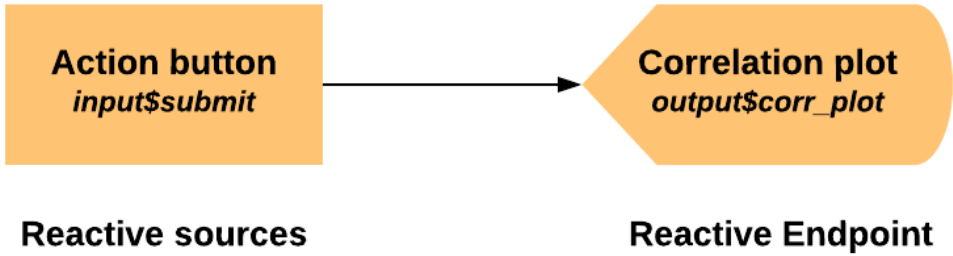
Objective	Complete
Introduce the concept of reactivity for interactive dashboards	✔
Summarize different objects in reactive programming with their flow and implementation	✔
Implement a simple reactive plot and a table in Shiny	✔
Build a Shiny app with reactive expressions	✔
Configure the app to isolate part of the app from regenerating	✔
Build a Shiny app to respond to actions using observeEvent()	
Configure the app to delay the response to events	
Outline different layouts to organize the Shiny widgets	
Host the app on the cloud-based service from RStudio	

Observing an event

- We have learned so far how to create reactive values, reactive expressions and to isolate a portion of app from re-rendering every time
- What if we want some action to trigger a new app?
- We want to observe an action from the user side and then trigger a response from server side
- We can use `observeEvent()` function for such cases

Creating a simple app with an action button

- Let's understand the functionality of `observeEvent()` with a simple app
- Let's create an app which displays a correlation plot of the first 10 variables of the `CMP` dataset when a button is clicked
- Within the UI, we will create an `actionButton` to trigger the rendering of plot event
- We will also create a title panel to give the title to the app
- The output is a plot output which is rendered by `plotOutput` function in `ui` and `renderPlot` function in `server` after clicking of the action button



Creating a simple app with an action button

- Find a subfolder `5-corrplot` in the `day-2` folder that contains the code for our new app
- Adding the ui part of the app in the `ui.R` script

```
ui <- fluidPage(

  # Title of the app.
  titlePanel("Correlation plot"),

  # Action button to trigger the event.
  actionButton(inputId = "submit",  #<- input ID
               label = "click me"), #<- input label

  # Render the output as plot.
  plotOutput(outputId = "corr_plot")

)
```

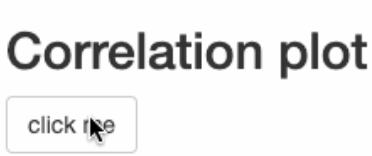
Creating a simple app with an action button

- Adding the server part of the app in the `server.R` script

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

observeEvent(input$submit,    #<- observe the click of action button

                # Print output scatterplot.
                output$corr_plot <- renderPlot({

                  corrplot(cor(CMP[1:10]))

                })
)
}
```

Creating a simple app with an action button

- Click on `Run App` from the app script
- The plot is generated only after the button is clicked
- Clicking the button is the action that triggered the display of the plot which is the event

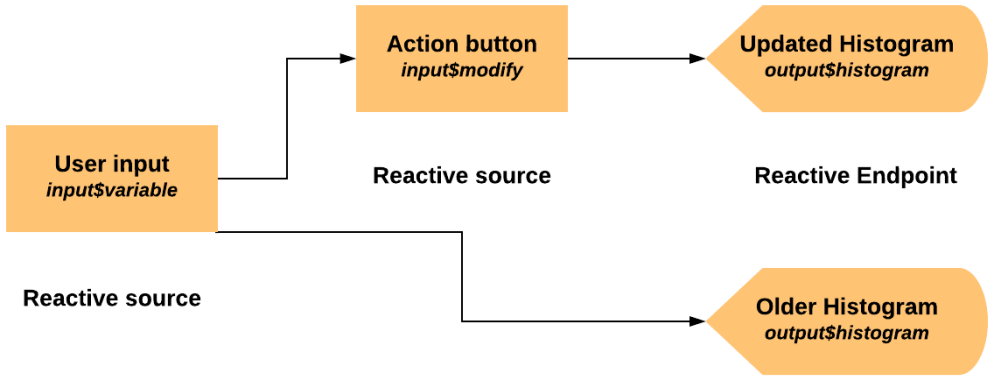


Module completion checklist

Objective	Complete
Introduce the concept of reactivity for interactive dashboards	✔
Summarize different objects in reactive programming with their flow and implementation	✔
Implement a simple reactive plot and a table in Shiny	✔
Build a Shiny app with reactive expressions	✔
Configure the app to isolate part of the app from regenerating	✔
Build a Shiny app to respond to actions using observeEvent()	✔
Configure the app to delay the response to events	
Outline different layouts to organize the Shiny widgets	
Host the app on the cloud-based service from RStudio	

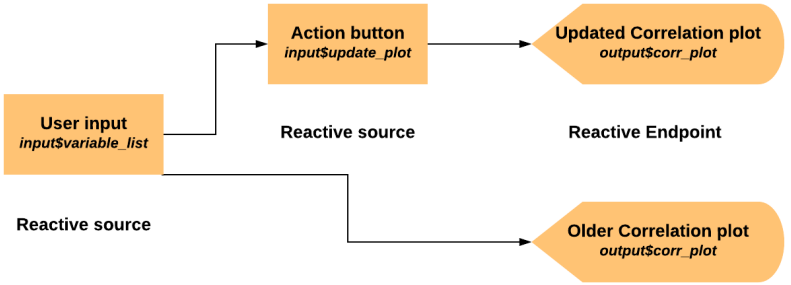
Creating apps with `eventReactive()`

- `eventReactive()` functions are used to delay the rendering of the app
- The user might constantly change the input, but we want the output to get updated based on the user's input only after explicitly stated by the user
- In this case, we use an `eventReactive()` function
- Say for example we want to plot a histogram of the variable chosen by the user
- The histogram plot does not get updated until user clicks on the `update plot` button



Creating an event reactive Shiny app

- Let's create an app which displays a correlation plot of all variables chosen by the user
- Within the UI, we will create an input `actionButton` to trigger the rendering of plot event
  - We will also create a title panel to give the title to the app
- Here we have two outputs:
  - a list of input columns for the user to choose multiple columns which is rendered by `uiOutput` function in `ui` and `renderUI` function in server
  - a plot output which is rendered by `plotOutput` function in `ui` and `renderPlot` function in `server` after clicking of the action button



- Navigate to `6-event-reactive-corrplot` folder within `day-2` to access the app

Creating an event reactive Shiny app

- Adding the ui part of the app in the `ui.R` script

```
ui <- fluidPage(

  # Title of the app.
  titlePanel("Correlation plot of chosen variables"),

  uiOutput("cor_var"), #<- input list of variables to select multiple variables

  # Action button to trigger the event.
  actionButton(inputId = "submit",  #<- input id
               label = "Update plot"), #<- input label

  # Render the output as plot.
  plotOutput(outputId = "corr_plot")

)
```

Creating an event reactive Shiny app

- Adding the server part of the app in the `server.R` script

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

  output$cor_var = renderUI({
    selectInput("cor_var",                          #<- input variable id
                "Choose Variables",                 #<- label to the select input
                choices = as.list(names(cmp_sub)), #<- choice list
                multiple = T,                       #<- allow multiple selection
                selected = names(CMP)[1:2])         #<- default values
  })

  data <- eventReactive(input$submit, {
    cmp_sub[, input$cor_var]     #<- selected input list
  })

  output$corr_plot = renderPlot({
              corrplot(cor(data()))  #,- render plot
            })
}
```

Creating a simple app with an action button

- Click on `Run App` from the app script

Knowledge check 3

Exercise 3

Module completion checklist

Objective	Complete
Introduce the concept of reactivity for interactive dashboards	✔
Summarize different objects in reactive programming with their flow and implementation	✔
Implement a simple reactive plot and a table in Shiny	✔
Build a Shiny app with reactive expressions	✔
Configure the app to isolate part of the app from regenerating	✔
Build a Shiny app to respond to actions using observeEvent()	✔
Configure the app to delay the response to events	✔
Outline different layouts to organize the Shiny widgets	
Host the app on the cloud-based service from RStudio	

Panels and layouts in Shiny

- **Panels and layouts** are used for arranging our input and output widgets in the UI
- We have `5` layouts along with `12` panel functions
- Panels include:

```
absolutePanel()
conditionalPanel()
fixedPanel()
headerPanel()
inputPanel()
mainPanel()
navlistPanel()
sidebarPanel()
tabPanel()
tabsetPanel()
titlePanel()
wellPanel()
```

- We have used `titlePanel()` to display the title of the app
- We will create an app with `tabPanel()` and `tabsetPanel()` today
- Feel free to understand the functionality of other panels with the helper function

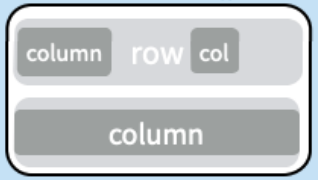
```
?panel_name
```

Layouts in Shiny

- The layouts combine multiple elements into a single element that contains its own panel functions
- The 5 layouts include:
  - `fluidRow()` - arrange the widgets in a row
  - `flowLayout()` - lay out elements in a left-to-right, top-to-bottom arrangement
  - `sidebarLayout()` - arrange the widget with a sidebar and a main bar
  - `splitLayout()` - lay out elements horizontally, dividing the available horizontal space into equal parts (by default)
  - `verticalLayout()` - create a container that includes one or more rows of content

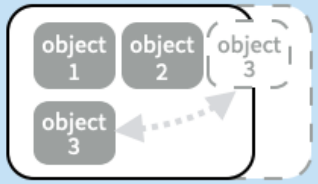
Layouts in Shiny

fluidRow()




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

flowLayout()




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

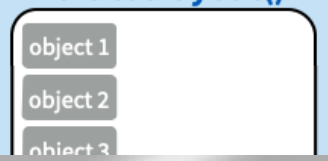
sidebarLayout()


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

splitLayout()


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

verticalLayout()




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

Layer of panels in Shiny

- Sometimes, we can create a layer of panels as well and navigate between them
- Three important functions allow us create a number of panels together as a single app in layers
- Let us understand how they are layered with an example with `tabsetPanel()`
  - `tabsetPanel` creates multiple tabs inside a single app horizontally
  - It looks like `Google Chrome` having multiple tabs
  - Consider one `tabsetPanel` to be a new window of `Google Chrome`
  - Each `tab` within the window is created using `tabPanel`

Layer of panels in Shiny

```
ui <- fluidPage( tabsetPanel(
    tabPanel("tab 1", "contents"),
    tabPanel("tab 2", "contents"),
    tabPanel("tab 3", "contents")))

ui <- fluidPage( navlistPanel(
    tabPanel("tab 1", "contents"),
    tabPanel("tab 2", "contents"),
    tabPanel("tab 3", "contents")))

ui <- navbarPage(title = "Page",
    tabPanel("tab 1", "contents"),
    tabPanel("tab 2", "contents"),
    tabPanel("tab 3", "contents"))
```

- There are also:
  - `navlistPanel` which arranges the tabs vertically
  - `navbarPage` which creates a top level navigation bar

`tabsetPanel` app today!

- We will create 3 tabs using `tabPanel` inside a `tabsetPanel`
  - tab1 contains the histogram of variable 1
  - tab2 contains the histogram of variable 2
  - tab3 contains the scatterplot of both the variable
- The code structure will look like:

```
tabsetPanel(
  tabPanel(),
  tabPanel(),
  tabPanel()
)
```

- Anything we want to display in a single tab should be written within the tab
- This includes:
  - input widget
  - output widget
  - HTML tags
  - Title

UI of the app

- Navigate to `7-layout-app` folder in `day-2` where we have our app

```
ui <- fluidPage(

  # Give the title to the app.
  titlePanel("Understanding layouts"),

  # Start the tab window.
  tabsetPanel(

    # Tab 1
    tabPanel("Histogram 1",                          #<- tab 1 name
             tags$br(), tags$br(),                   #<- html breaks
             tags$h2("Histogram of variable 1"),     #<- title of plot
             tags$br(), tags$br(),                   #<- html breaks

             # Select input from a list of inputs.
             selectInput(inputId = "variable1",           #<- input id
                         label = "Choose the variable:",  #<- input label
                         choices = as.list(names(CMP)),   #<- input list
                         selected = names(CMP)[1]),       #<- default value of input

             plotOutput("histogram1")            #<- Plot output
    ),
```

UI of the app

```
    # Tab 2
    tabPanel("Histogram 2",                          #<- tab 2 name
             tags$br(), tags$br(),                   #<- html breaks
             tags$h2("Histogram of variable 2"),     #<- title of the plot
             tags$br(), tags$br(),                   #<- html breaks

             # Select input from a list of inputs.
             selectInput(inputId = "variable2",           #<- input id
                         label = "Choose the variable:", #<- input label
                         choices = as.list(names(CMP)),  #<- input list
                         selected = names(CMP)[1]),      #<- default value of input

             plotOutput("histogram2")             #<- plot output
    ),

    # Tab 3
    tabPanel("Relationship between two variables",     #<- tab 3 name
             tags$br(), tags$br(),                     #<- html breaks
             tags$h2("Scatter plot of two variables"), #<- title of the plot
             tags$br(), tags$br(),                     #<- html breaks
             plotOutput("scatter_plot")                #<- plot output

             )
    )
  )
```

Server of the app

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

  output$histogram1 = renderPlot({

    # Plot the histogram.
    hist(CMP[[input$variable1]],       #<- variable name chosen by user
         col = "salmon",               #<- color of the histogram
         main = input$variable1,       #<- title of the histogram
         xlab = input$variable1)       #<- x axis label
  })

  output$histogram2 = renderPlot({

    # Plot the histogram.
    hist(CMP[[input$variable2]],       #<- variable name chosen by user
         col = "lightblue",            #<- color of the histogram
         main = input$variable2,       #<- title of the histogram
         xlab = input$variable2)       #<- x axis label
  })
```

Server of the app

```
  # Print output scatterplot.
  output$scatter_plot <- renderPlot({

    plot(CMP[[input$variable1]],       #<- variable 1
         CMP[[input$variable2]],       #<- variable 2
         col = "salmon",               #<- color of the plot
         pch = 16,                     #<- plot with filled circles
         xlab = input$variable1,       #<- x axis label
         ylab = input$variable2        #<- y axis label
         )

  })

}
```

Run the app with different tabs

- Click on `Run App` to run the app

Module completion checklist

Objective	Complete
Introduce the concept of reactivity for interactive dashboards	✔
Summarize different objects in reactive programming with their flow and implementation	✔
Implement a simple reactive plot and a table in Shiny	✔
Build a Shiny app with reactive expressions	✔
Configure the app to isolate part of the app from regenerating	✔
Build a Shiny app to respond to actions using observeEvent()	✔
Configure the app to delay the response to events	✔
Outline different layouts to organize the Shiny widgets	✔
Host the app on the cloud-based service from RStudio	

Deploy the app

- We can deploy our app on the web server for sharing
- There are both free and paid services available
- We will host our app for free today on ***https://www.shinyapps.io/***
- We need an account to deploy our app

Steps in deploying the app

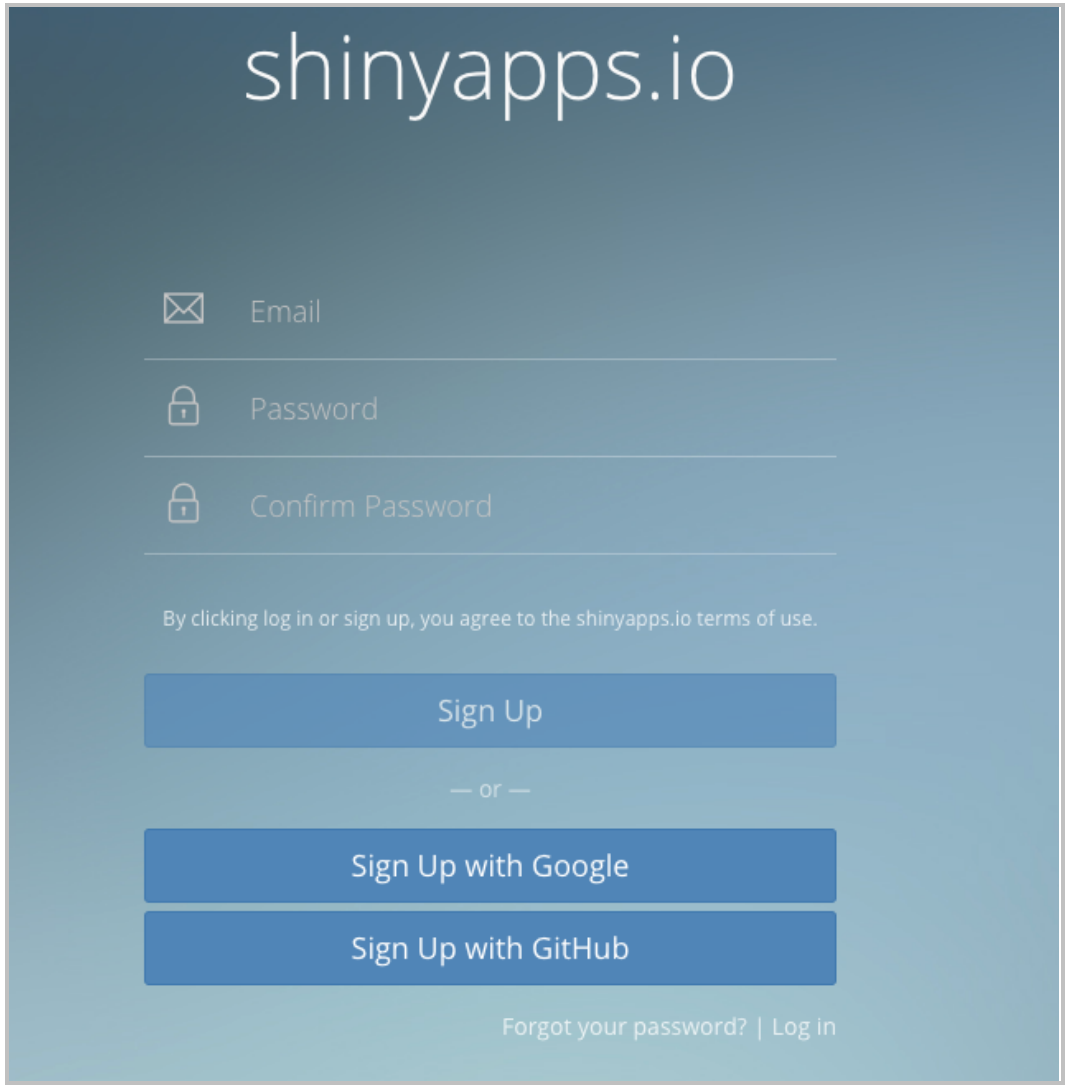
- Create your app and run locally and make sure it is working
- Create an account in ***https://www.shinyapps.io/*** and login to your account
- Install and load `rsconnect` library
- Navigate and set the working directory to the app you want to deploy
- Authorize your web account to access your local app
  - This can be done by getting a token and secret key from the account
  - Paste the token and secret key to the R console
- Use `deployApp()` to deploy the app
- **Remember keep all the data files needed for the app in the app folder**

Create app and run locally

- We will deploy a simple histogram we created first today
- Navigate to `8-deploying-app` folder in `day-2`
- Run the app and check that it runs locally
- Note we kept our CMP dataset in the app folder
- Also we have not set the working directory to `data_dir`
- This is because while deploying the app to web, any path specified should not be relative but absolute
- So in order to avoid the issue, we do not specify any path in our code

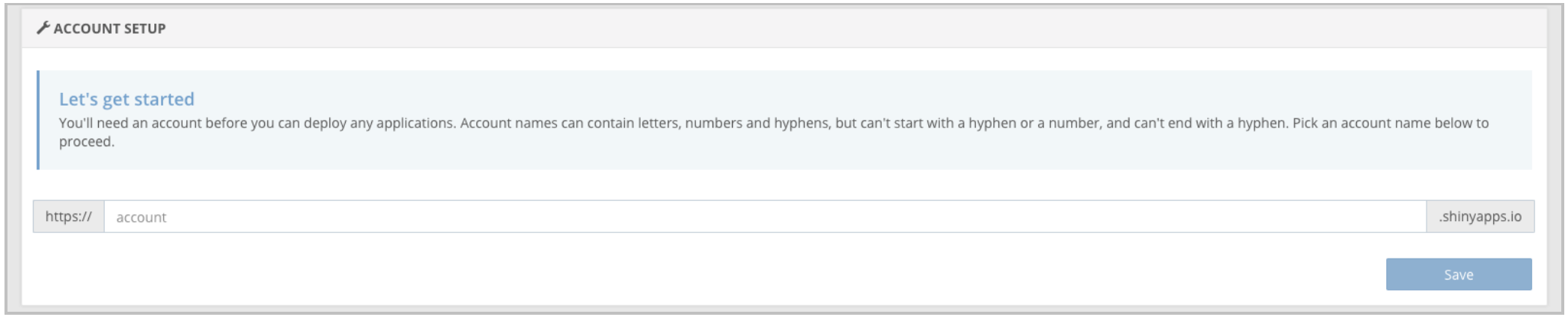
Create an account

- Go to ***https://www.shinyapps.io/admin/#/signup*** and create a new account
- You can use your Google or Github account if you have one
- Log in to the account after signing up



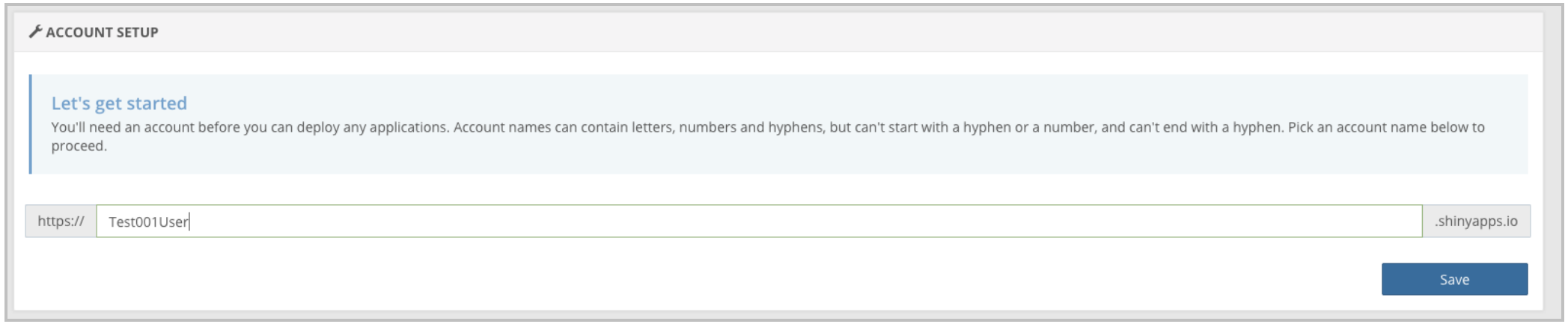
Give a unique user name

- When you sign up for the first time, it asks for an account setup to give a unique name to your account



Give a unique user name

- Give any name accepted by the website and save it



Install and load package

```r
# Install package.
install.packages("rsconnect")

# Load the package into the environment.
library(rsconnect)

# View package documentation.
library(help = "rsconnect")
```

rsconnect-package {rsconnect} R Documentation

Deployment Interface for R Markdown Documents and Shiny Applications

Description

The 'rsconnect" package provides a programmatic deployment interface for RPubs, shinyapps.io, and RStudio Connect. Supported contents types include R Markdown documents, Shiny applications, plots, and static web content.

Managing Applications

Deploy and manage applications with the following functions:

- deployApp(): Deploy a Shiny application to a server.
- configureApp(): Configure an application currently running on a server.
- restartApp(): Restart an application currently running on a server.
- terminateApp(): Terminate an application currently running on a server.
- deployments(): List deployment records for a given application directory.

More information on application management is available in the applications() help page.

Managing Accounts and Users

Manage accounts on the local system.

- setAccountInfo(): Register an account.
- removeAccount(): Remove an account.
- accountInfo(): View information for a given account.

More information on account management is available in the accounts() help page.

Set working directory

- Set the working directory to the app directory in the console

```
setwd("~/Desktop/af-werx/shiny/day-2/9-deploying-app")
```

Authorize the account

- Go to tokens on your dashboard
- Click on `Show` and again click on `show secret`
- Copy the entire command and paste to your console

Deploy the app

- Type `deployApp()` in the console
- It takes sometime to deploy and it directly opens in the browser once it deploys
- Share the URL to your friends or team to show your work!

Knowledge check 4

Exercise 4

Module completion checklist

Objective	Complete
Introduce the concept of reactivity for interactive dashboards	✔
Summarize different objects in reactive programming with their flow and implementation	✔
Implement a simple reactive plot and a table in Shiny	✔
Build a Shiny app with reactive expressions	✔
Configure the app to isolate part of the app from regenerating	✔
Build a Shiny app to respond to actions using observeEvent()	✔
Configure the app to delay the response to events	✔
Outline different layouts to organize the Shiny widgets	✔
Host the app on the cloud-based service from RStudio	✔

Workshop!

- Workshops are to be completed in the afternoon either with a dataset for a capstone project or with another dataset of your choosing
- Make sure to annotate your code so that it is easy for others to understand what you are doing
- This is an exploratory exercise to get you comfortable with the content we discussed today
- Today, you will:
  - Create interactive dashboards by taking inputs from the users
  - Display plots, tables, text outputs
  - Incorporate `reactive()`, `observeEvent()`, `eventReactive()` functions
  - Use different input and output widgets
  - Use the dataset which you have been using to build your visualizations in the previous classes
  - Use different layouts and organize your app
  - Host the app on the cloud platform

This completes our module

Congratulations!