R-Shiny Workshop

COP of Biometrics, Bioinformatics and Data Management
Amsterdam - Sep. 24-29, 2018

Omar Benites Alfaro
o.benites@cgiar.org
https://github.com/omarbenites

Ivan Perez Masias
i.perez@cgiar.org
https://github.com/ivanperezm

Outline

Part I	Part II	Part III	Part IV	Part V
• **Introduction** • CIP examples • Shiny/RStudio • Structure • Setup of shiny app • Inputs & outputs • render*() • **Exercise 1**	• Render UI • Conditional panel • Reactivity expression • Plot output • **Exercise 2** • Html output • DT library • Action button and observe-Event • rhandsondtable • **Exercise 3**	**Shiny Dashboard** • Basics • Structure overview • Header • Sidebar • Body • Layouts • **Exercise 4**	**How to customize appearance** • Skins • CSS • Long titles • Sidebar width • Icons • Statuses and colors • **Exercise 5**	**Sharing apps** • Shinyapps.io • Shiny servers

Part I
Introduction

Objectives

The main objective of the Shiny-Workshop is to built interactive applications using Shiny and R code based on real case studies.

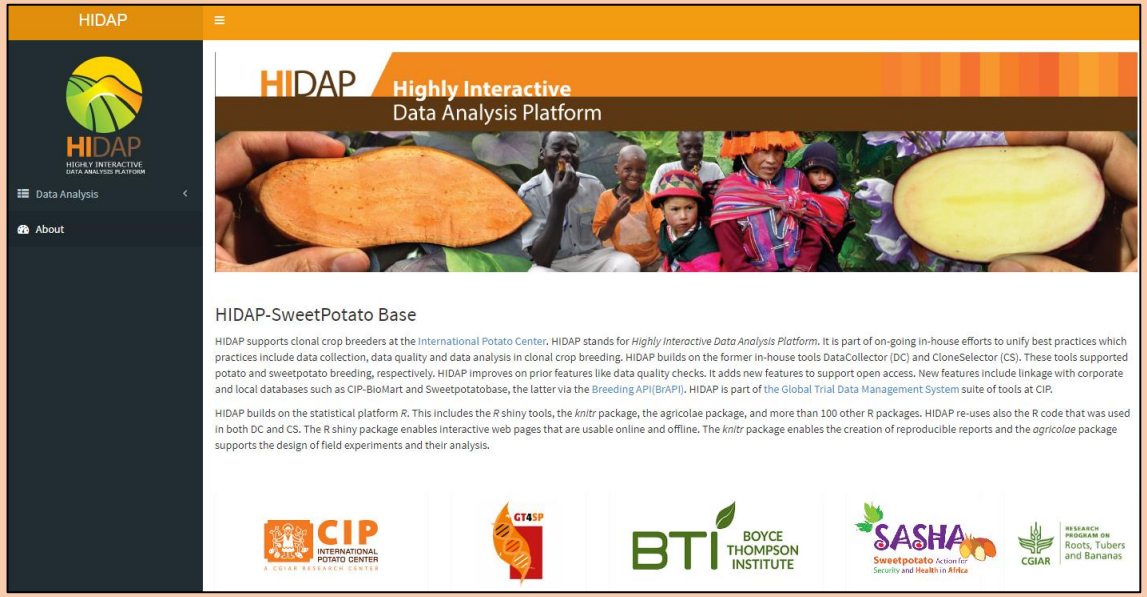
Learn how to use and manipulate different types of inputs and outputs in the Shiny environment.

Pre-requisites

- Programming in R.
- **No previous knowledge** in languages such as HTML, PHP or Javascript is requiered. It only uses R code.

CIP examples

Demos of shiny.
CIP Breeding, Shiny and BRAPI and CGIAR Big Data Platform (AGROFIMS)



https://research.cip.cgiar.org/gtdms/
https://apps.cipotato.org/hidap_sbase/

https://apps.cipotato.org/hidapagrofims/

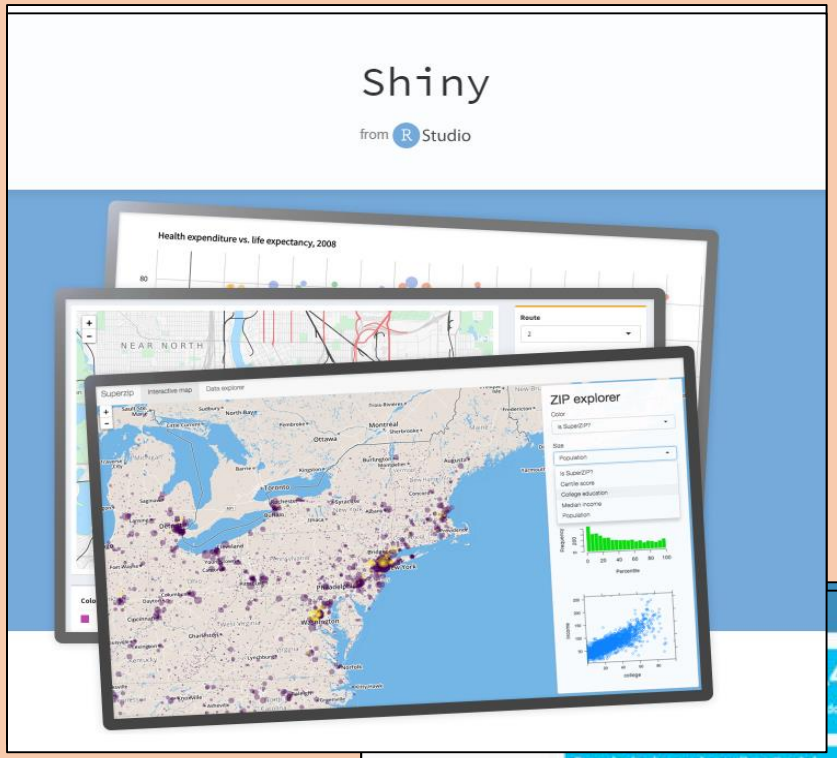
Shiny

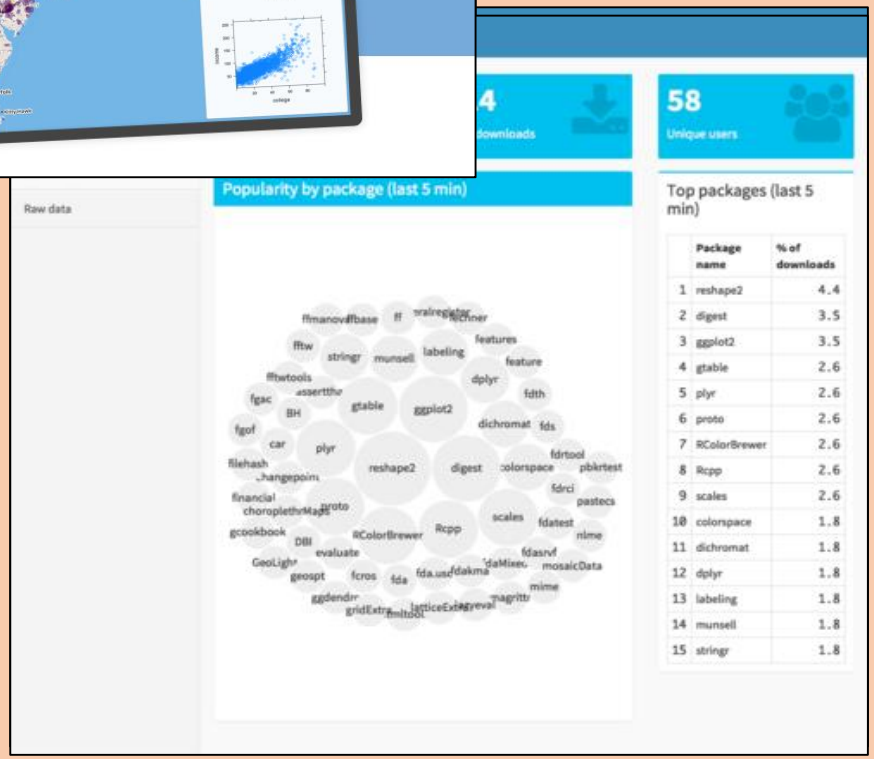
Shiny is an R package that makes it easy to **build interactive web apps** straight from R.

Shiny combines the computational power of R with the interactivity of the modern web.

You can host standalone apps on a webpage or embed them in R Markdown documents or build dashboards.

You can also extend your Shiny apps with CSS themes, htmlwidgets, and JavaScript actions.





R Shiny extensions

Additionally, we will cover and use other r-packages that improve the interface and functionalities of Shiny Apps.

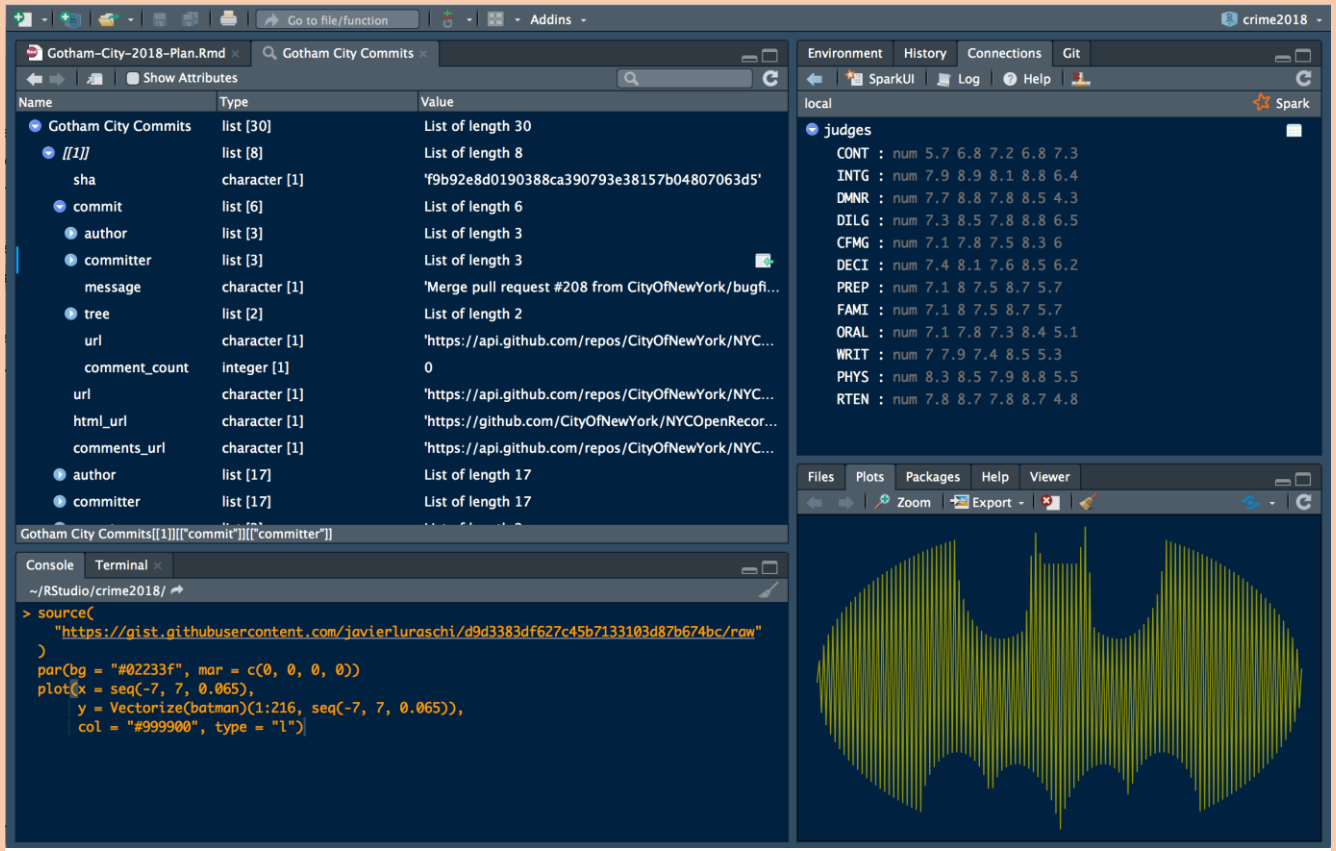
The ones that will cover as follow:

- **shinyjs**: lets you perform common useful JavaScript operations in Shiny apps. Link
- **shinysky**: lets you perform common useful JavaScript operations in Shiny apps. Link
- **shinyBS**: package that adds several additional Twitter Boostrap components to shiny . Link
- **bsplus**: provide access to some useful Bootstrap components for rmarkdown html-documents and shiny apps. Link

RStudio

RStudio is an integrated development environment (IDE) for R. It includes a **console**, syntax-highlighting **editor** that supports direct code execution, as well as **tools** for plotting, **history**, **debugging** and **workspace management**.

RStudio is available in **open source** and **commercial** editions and runs on the **desktop** (Windows, Mac, and Linux) or in a **browser** connected to RStudio Server or RStudio Server Pro (Debian/Ubuntu, RedHat/CentOS, and SUSE Linux).



Structure

Shiny apps are contained in a single script called **app.R**. The script app.R lives in a directory (for example, newdir/) and the app can be run with **runApp("newdir")**.

app.R has three components:

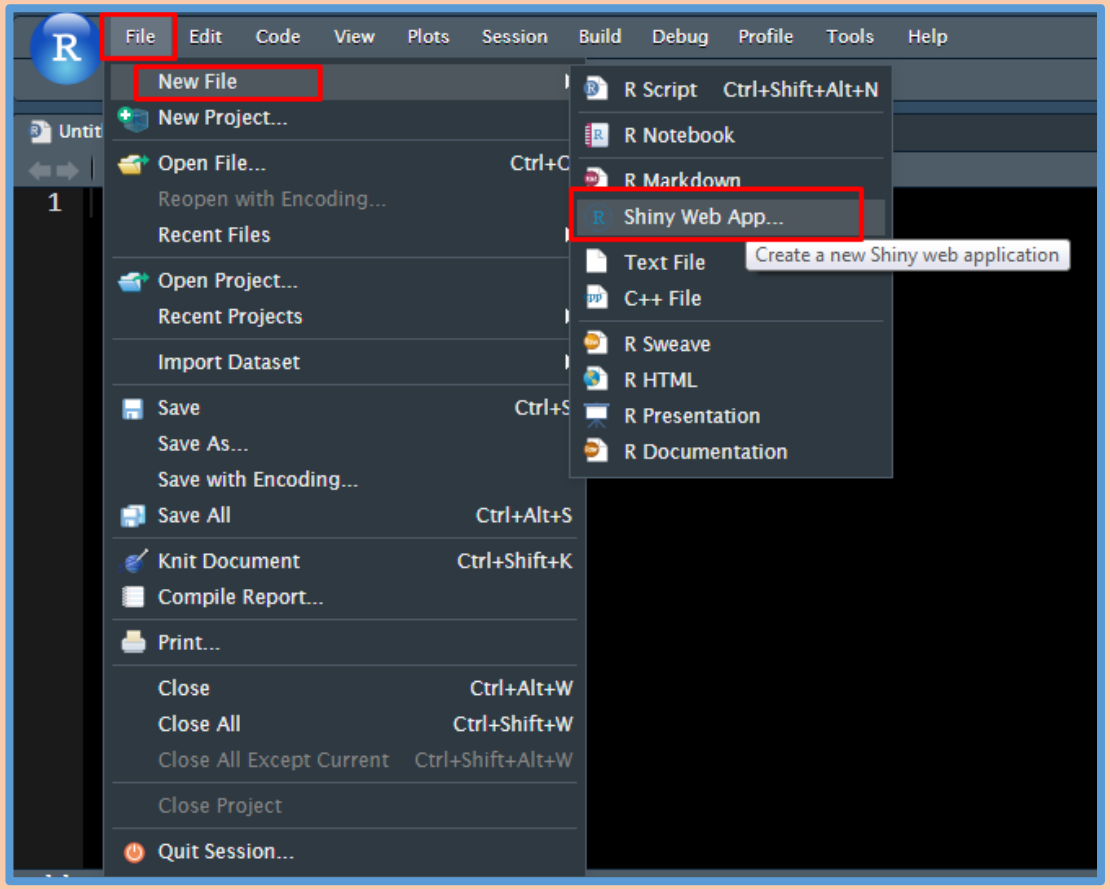
a user interface object
The user interface (ui) object controls the layout and appearance of your app.

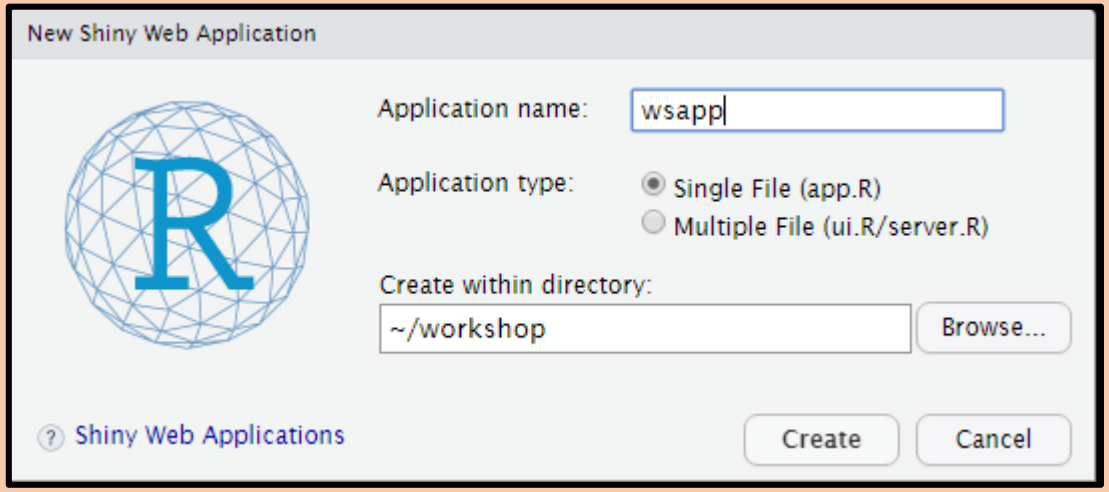
a server function
The server function contains the instructions that your computer needs to build your app

a call to the shinyApp function
Finally the shinyApp function creates Shiny app objects from an explicit UI/server pair.

```r
library(shiny)

# Define UI ----
ui <- fluidPage(

)

# Define server logic ----
server <- function(input, output) {

}

# Run the app ----
shinyApp(ui = ui, server = server)
```

Setup of shiny app





Setup of shiny app

Requirements

- R and Shiny package must installed in your computer or server.
- RStudio for the deployment of the applications.

Steps

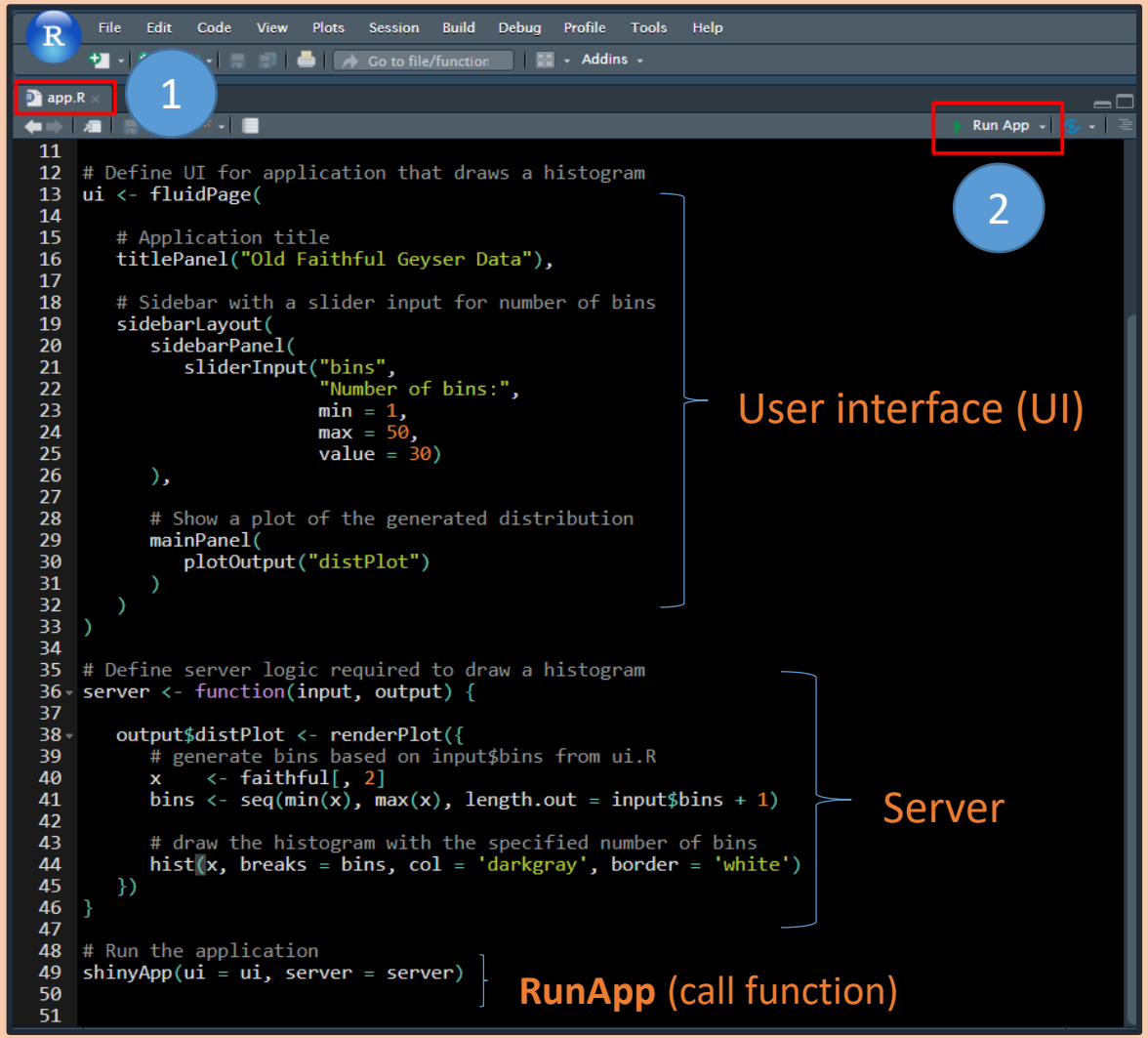
Method #1: One-file App

We can write all the code (ui and server) in a single file **app.R** file. Then, call **runApp(ui,server)** to launch the application in the browser.

Method #2 : Two-files App

Alternatively, Shiny applications have two main files: **ui.R** and **server.R**. Both are for the user interface and the server functionalities respectively. It's highly adopted when the code increase in both sites.

Method #1: One-file App



1 By default, RStudio create app.R tab containing the ui and server code.

2 In the right side, exist Run App the button to launch the shiny application.

Method #2 : Two-files App

```r
library(shiny)

# Define UI for application that draws a histogram
ui <- fluidPage(

   # Application title
   titlePanel("Old Faithful Geyser Data"),

   # Sidebar with a slider input for number of bins
   sidebarLayout(
      sidebarPanel(
         sliderInput("bins",
                     "Number of bins:",
                     min = 1,
                     max = 50,
                     value = 30)
      ),

      # Show a plot of the generated distribution
      mainPanel(
         plotOutput("distPlot")
      )
   )
)

# Define server logic required to draw a histogram
server <- 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')
   })
}

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

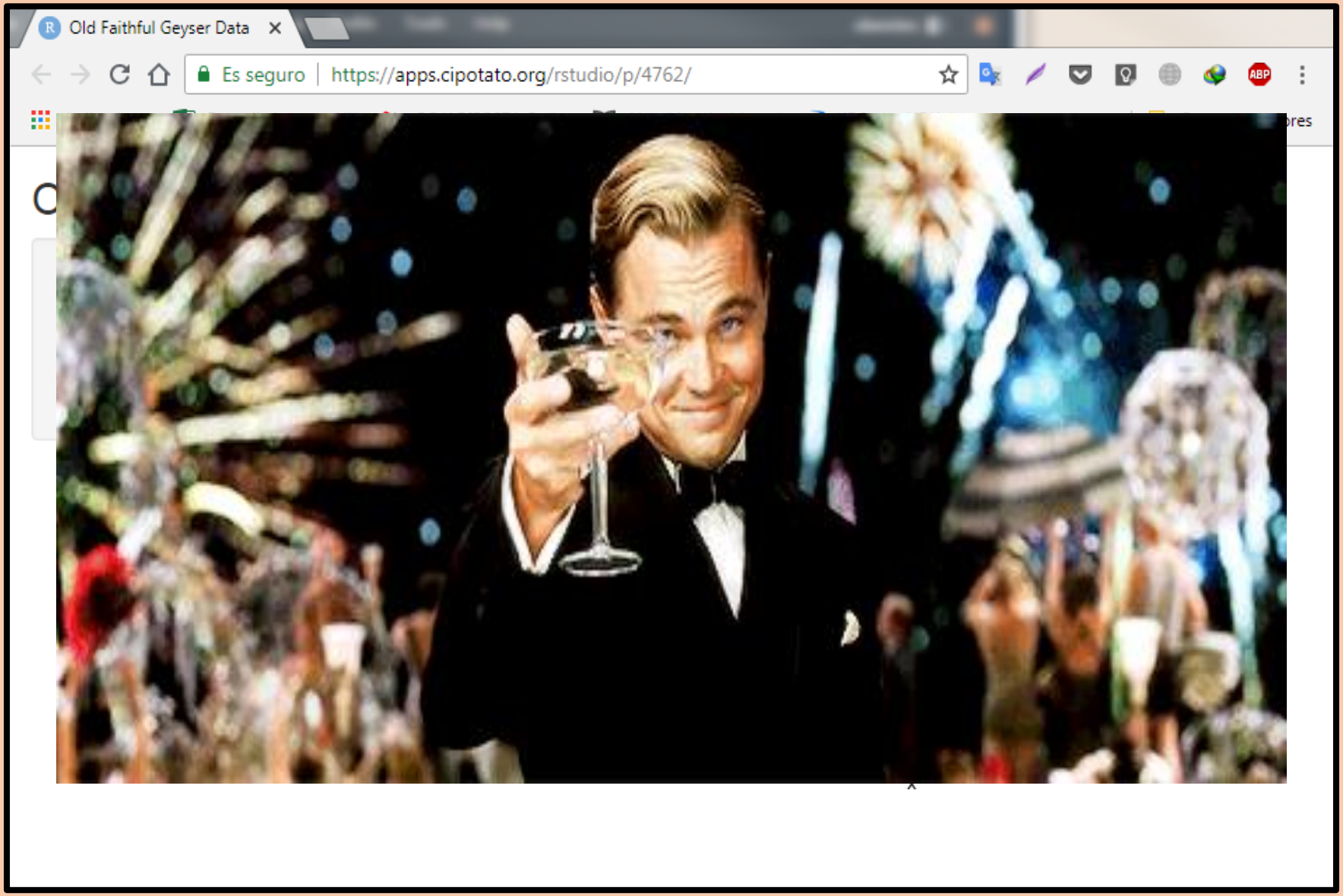
```r
# ui.R
# Define UI for application that draws a histogram
ui <- fluidPage(
  # Application title
  titlePanel("Old Faithful Geyser Data"),
  # Sidebar with a slider input for number of bins
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
       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

ui.R

```r
# server.R

# Define server logic required to draw a histogram
server <- 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')
  })
}
```

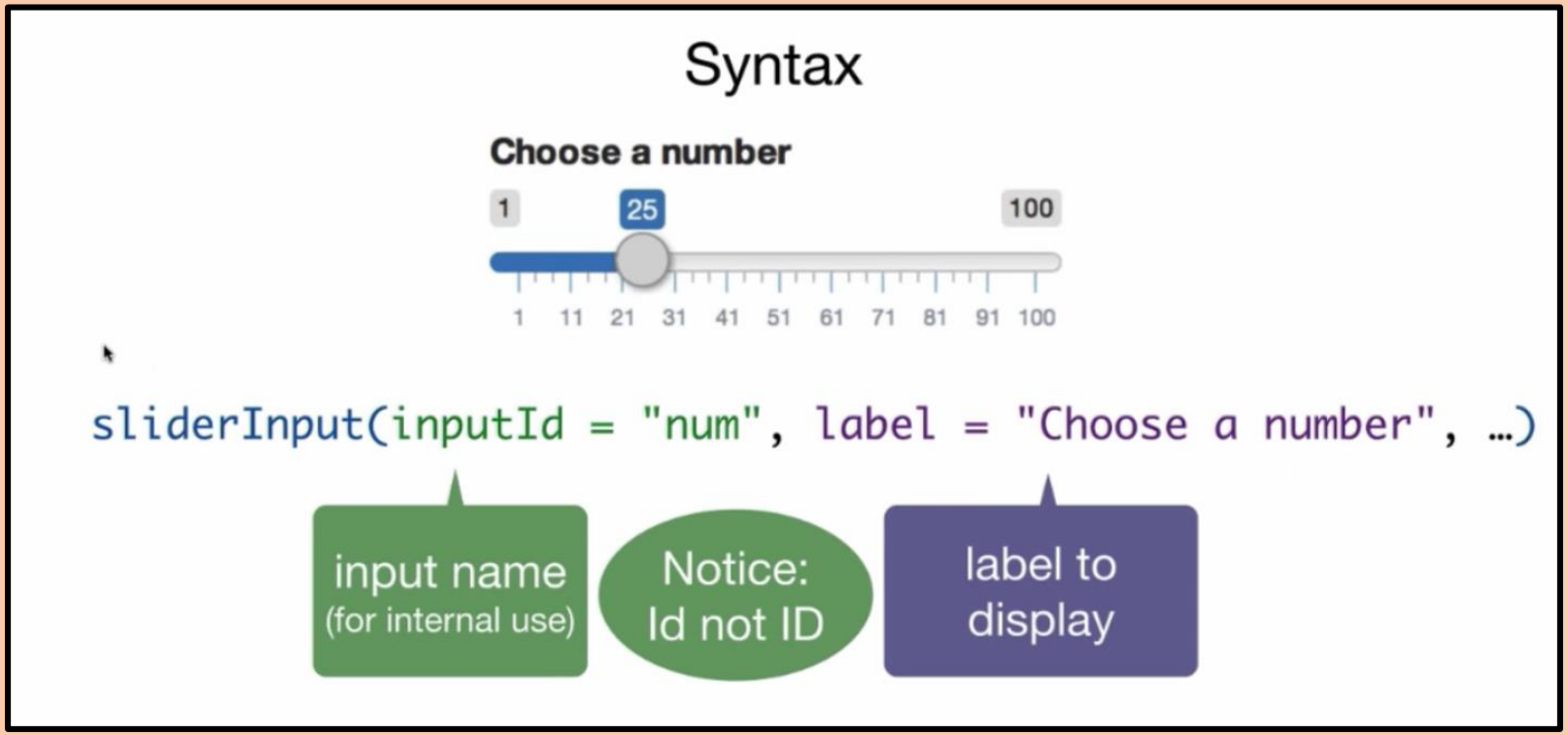
server.R

Launch an app



Input elements

Input syntax



Input elements

Shiny Inputs

Type	Function	Arguments	Example
Numeric input	`numericInput`	`inputId, label, value, min, max, step`	Enter your age 40
Text input	`textInput`	`inputId, label, value, width`	Enter your name
Options list	`radioButtons`	`inputId, label, choices, selected, inline, width`	Enter your gender Male Female
Drop-down list	`selectInput`	`inputId, label, choices, selected, multiple, selectize, width, size`	Enter your race White
Drop-down list	`selectizeInput`	... + options	
Numeric input (minimum, maximum)	`sliderInput`	`inputId, label, min, max, value, step, animate`	Score the product (0-10) 0 5 10 0 1 2 3 4 5 6 7 8 9 10
True/False	`checkboxInput`	`inputId, label, value, width`	I accept
Button	`actionButton`	`inputId, label, icon, width`	submit

Output elements

Output syntax

server.R

```
output$selected_var <- renderText({
  "You have selected this"
})
```

Output ID

Shiny Render Output

ui.R

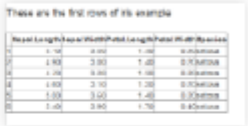
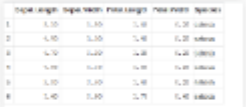
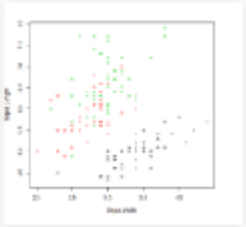
```
mainPanel(
  textOutput("selected_var")
)
```

Shiny Output

Output ID

Output elements

Shiny Outputs

Type	Function	Arguments	Example
R-console like text	`verbatimTextOutput`	`outputId`	
HTML interpreted text	`htmlOutput`	`outputId, inline`	
"Regular" table	`tableOutput`	`outputId`	
Dynamic table	`dataTableOutput`	`outputId`	
Plots	`plotOutput` `imageOutput`	`outputId, width, height, click,...`	

Exercise 1

Part II
Reactivity

renderUI

Create a shiny Input (widget) from the server side.

There are cases when the inputs need to be rendered using data that is previously processed in the server side.

```
ui <- fluidPage(
  uiOutput("moreControls")
)

server <- function(input, output) {
  output$moreControls <- renderUI({
    tagList(
      sliderInput("n", "N", 1, 1000, 500),
      textInput("label", "Label")
    )
  })
}
shinyApp(ui, server)
}
```

Conditional Panel

Creates a panel that is visible or not, depending on input values.

What do you want to show?

- (•) Table
- ( ) Summary

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.10	3.50	1.40	0.20	setosa
4.90	3.00	1.40	0.20	setosa
4.70	3.20	1.30	0.20	setosa
4.60	3.10	1.50	0.20	setosa
5.00	3.60	1.40	0.20	setosa
5.40	3.90	1.70	0.40	setosa

What do you want to show?

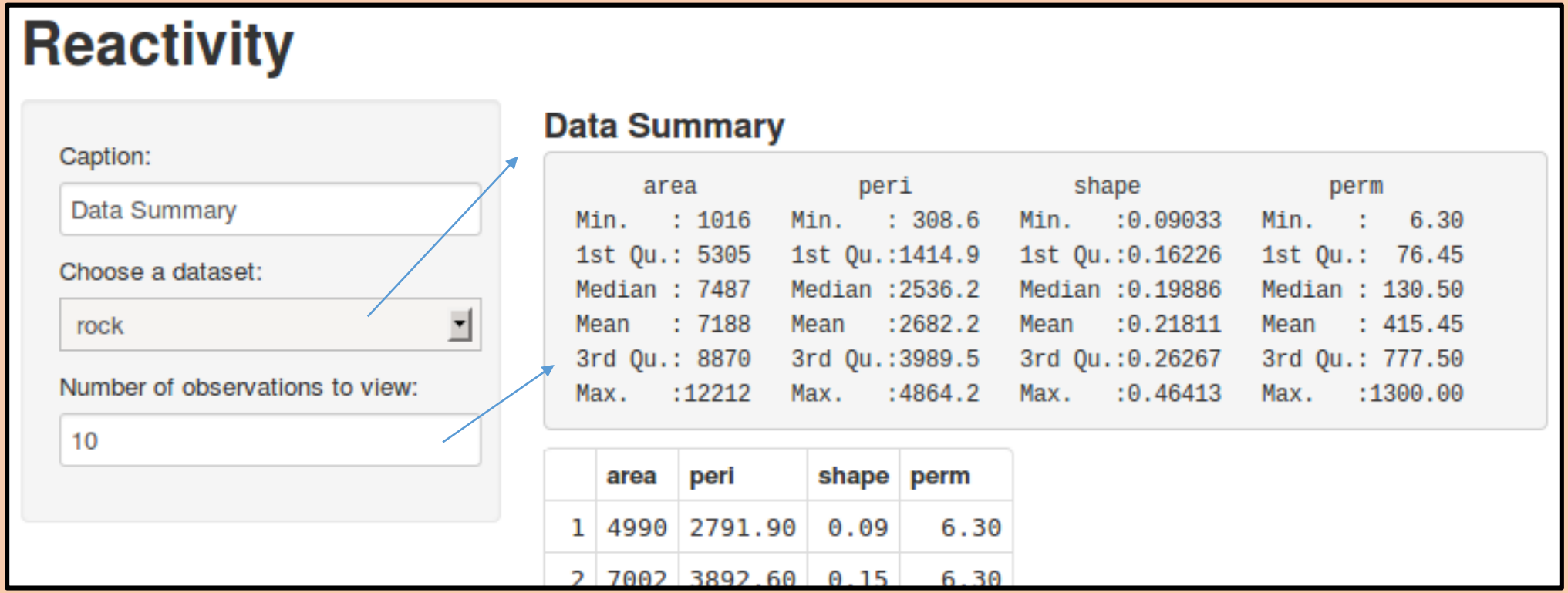
- ( ) Table
- (•) Summary

```
 Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Spe
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100   setosa
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300   versicolo
 Median :5.800   Median :3.000   Median :4.350   Median :1.300   virginica
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
```

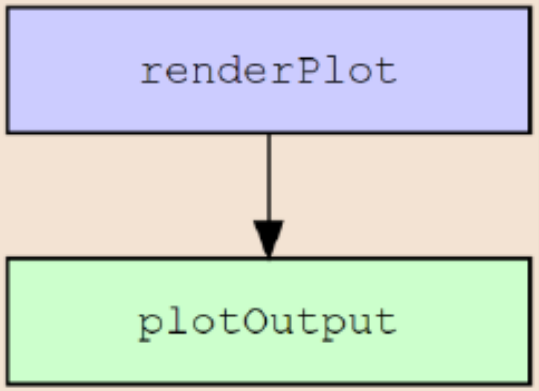
```
conditionalPanel(condition  =
"input.idCbox == 'Table'”,
...

)
conditionalPanel(condition =
"input.idCbox == ‘Summary'”,
...

)
```

Reactive Expressions

Shiny comes with a **reactive programming** library that you will use to structure your application logic. By using this library, changing input values will naturally cause the right parts of your R code to be reexecuted, which will in turn cause any changed outputs to be updated.



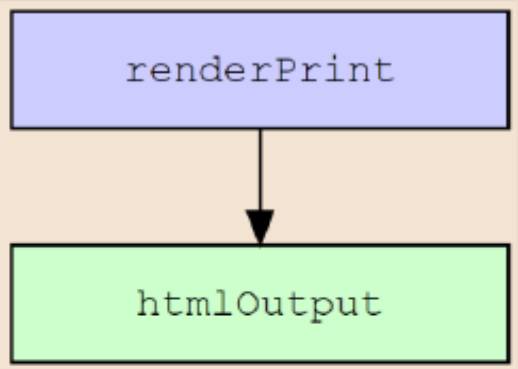
plot Output




```r
ui <- fluidPage(
  plotOutput("result")
)

server <- function(input, output) {

  output$result<-renderPlot({
    plot(Sepal.Length ~ Sepal.Width,
         col = Species, data = iris)
  }, width = 500, height = 500)

}

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

How would the figure look like if width and height are not specified?

html Output




```r
library(xtable)

ui <- fluidPage(

  htmlOutput("result")

)

server <- function(input, output) {

  output$result<-renderPrint({
    print(xtable(head(iris)), type = "html")
  })

}

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

Exercise: Replace `htmlOutput` by `verbatimTextOutput`

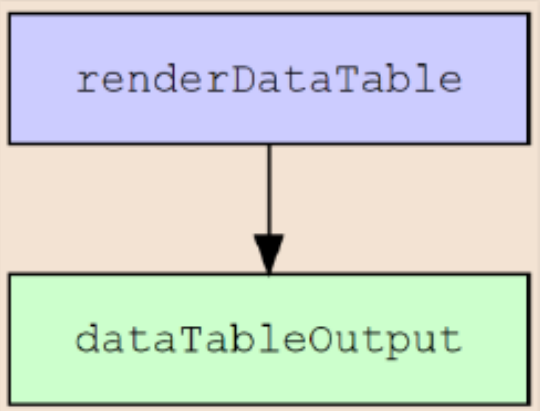
DT Library: DataTable

Dynamic table: extensions

- No need to plug into Shiny app
- Wrap the data.frame using **datatable** function from **DT** package to add more options (filter, rownames, download buttons, ...)
- Execute the following code in R-script or R-markdown.

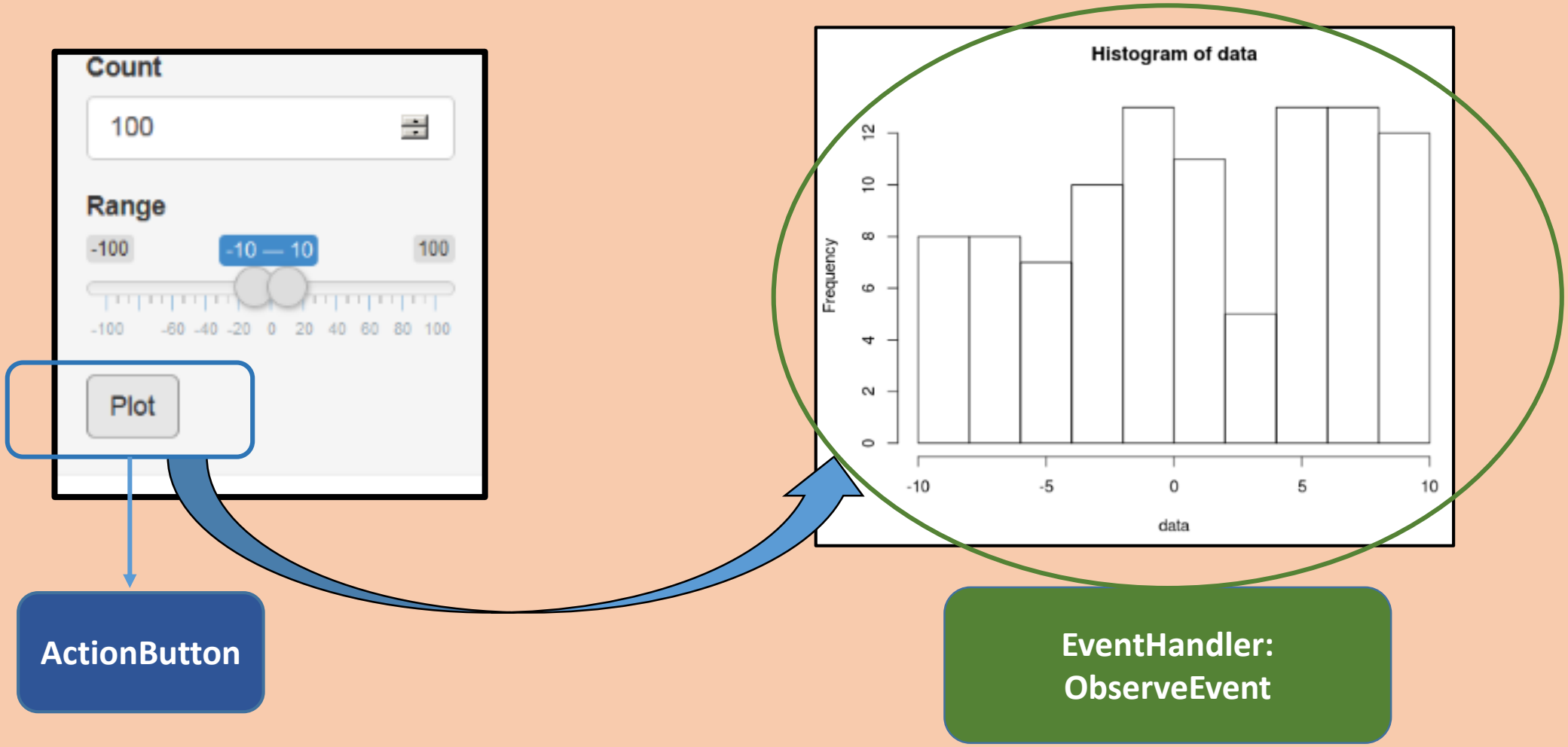
```
library(DT)
DT::datatable(iris, filter="top", rownames=FALSE,
    extensions = 'Buttons',
    options = list(dom = 'Bfrtip',
                   buttons = c('copy', 'csv', 'excel', 'pdf', 'print')
                  )
)
```

DT Library: DataTable

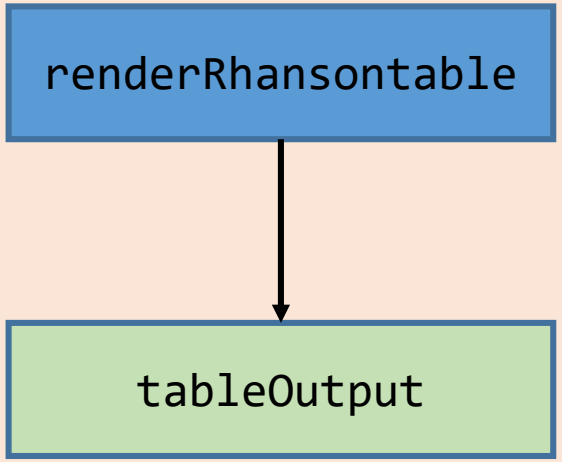



```r
ui <- fluidPage(

  dataTableOutput("result")

)

server <- function(input, output) {

  output$result<-renderDataTable({
    iris
  })

}

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

ActionButton and ObserveEvent



rhansontable Library: R-Based Spreadsheet table



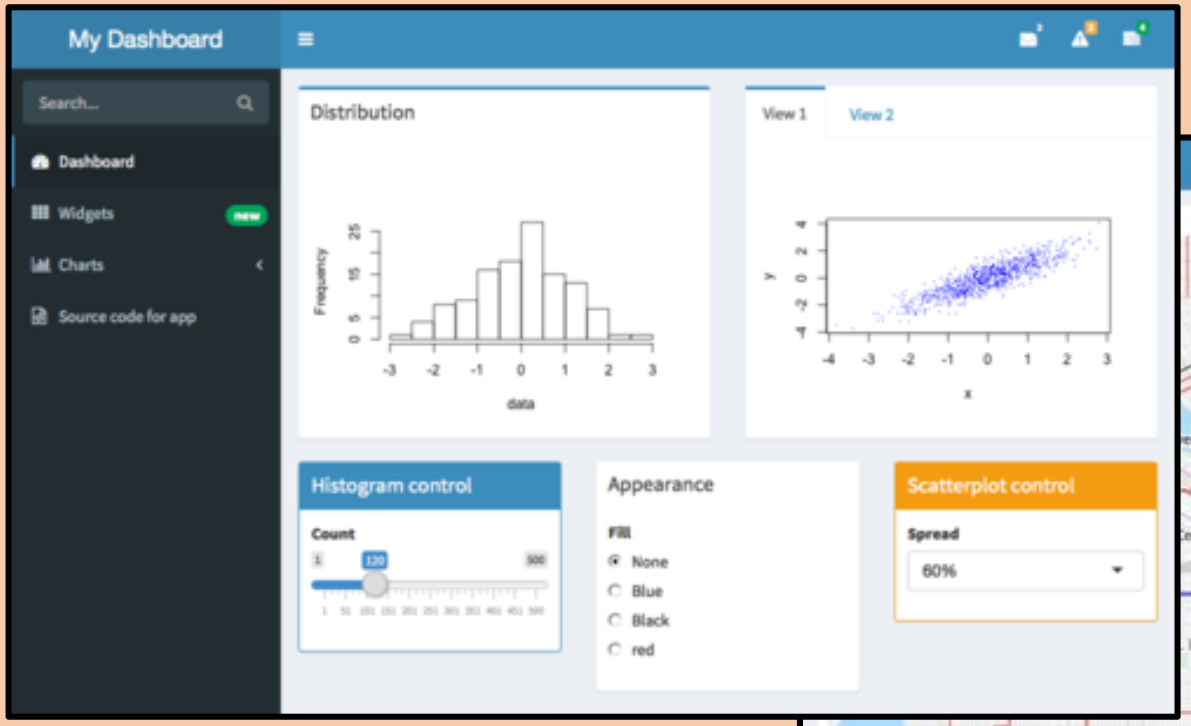

```r
library(rhandsontable)
library(shiny)

ui<- fluidPage(
  rHandsontableOutput("dtrhand")
)

server<-function(input, output){

  output$dtrhand <- renderRHandsontable({
    DF = iris
    if (!is.null(DF))
      rhandsontable(DF)
  })

}

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

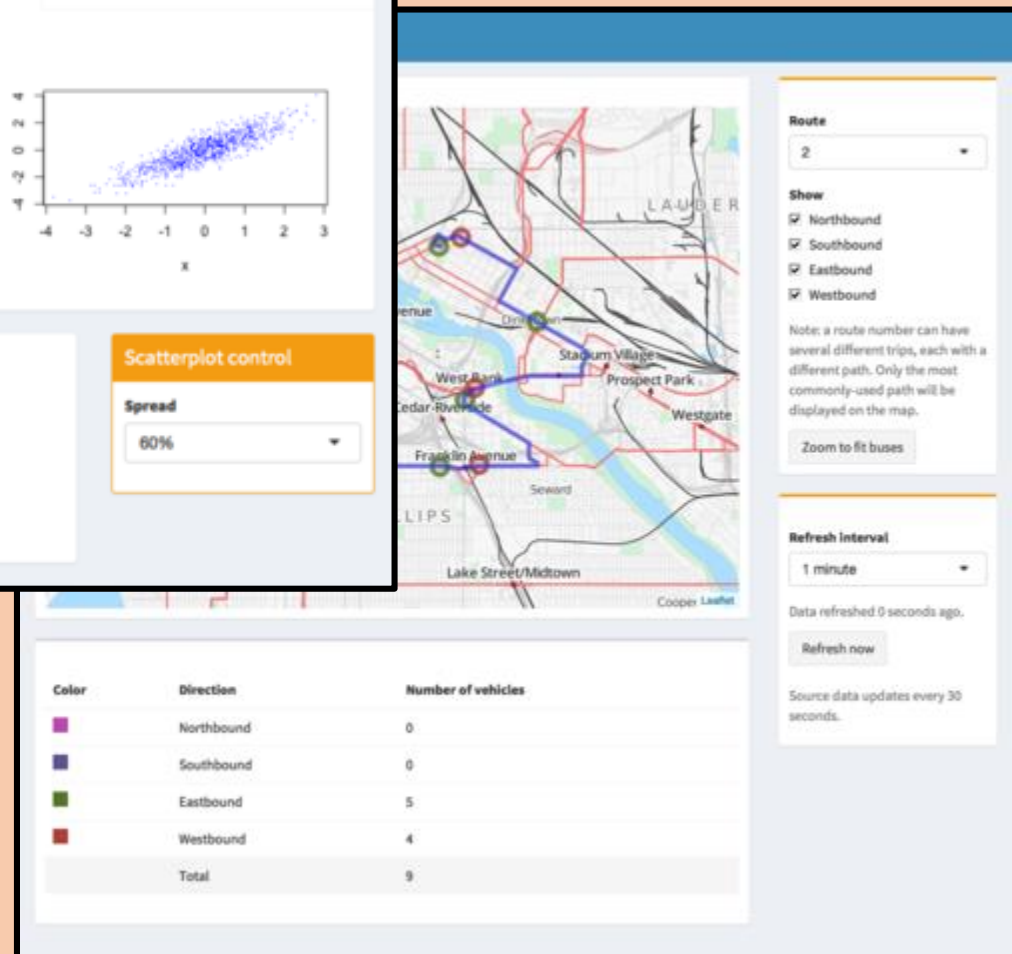
Exercise 3

Part III
Shiny Dashboard

Shiny Dashboard

shinydashboard is an R package whose job is to make it easier (as the name suggests) to build dashboards with shiny.





shinydashboard makes it easy to use Shiny to create dashboards like these:

Shiny Dashboard

A dashboard has three parts: a header, a sidebar, and a body. Here's the most minimal possible UI for a dashboard page.

```
## ui.R ##
library(shinydashboard)

dashboardPage(
  dashboardHeader(),
  dashboardSidebar(),
  dashboardBody()
)
```

Shiny Dashboard

Basics

You can quickly view it at the R console by using the shinyApp() function. (You can also use this code as a single-file app).

```r
## app.R ##
library(shiny)
library(shinydashboard)

ui <- dashboardPage(
  dashboardHeader(),
  dashboardSidebar(),
  dashboardBody()
)

server <- function(input, output) { }

shinyApp(ui, server)
```

Shiny Dashboard

UI & Server

UI

- Define Dashboard Page: **dashboardPage()**
- The dashboard page is divided in 3 elements:
  - Header: **dashboardHeader()**
  - Sidebar: **dashboardSidebar()**
  - Body: **dashboardBody()**

ui.R | server.R

```
#ShinyDashboard

library(shiny)
library(shinydashboard)


# Define UI for application
dashboardPage(
  dashboardHeader(),
  dashboardSidebar(),
  dashboardBody()
)
```

Server

- Where all computations are performed.
- Render inputs and reactive values are placed here.

ui.R | server.R*

```
library(shinydashboard)
library(shiny)

# Define server logic
server <- function(input, output) {


}
```

Shiny Dashboard

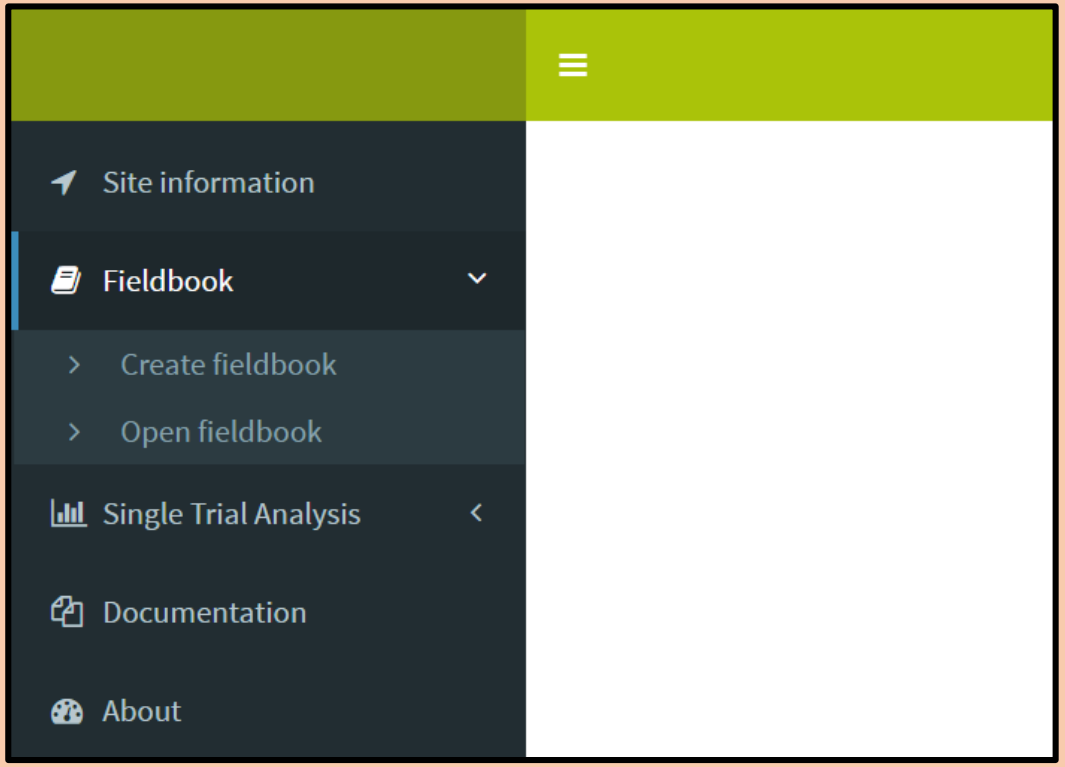
Header

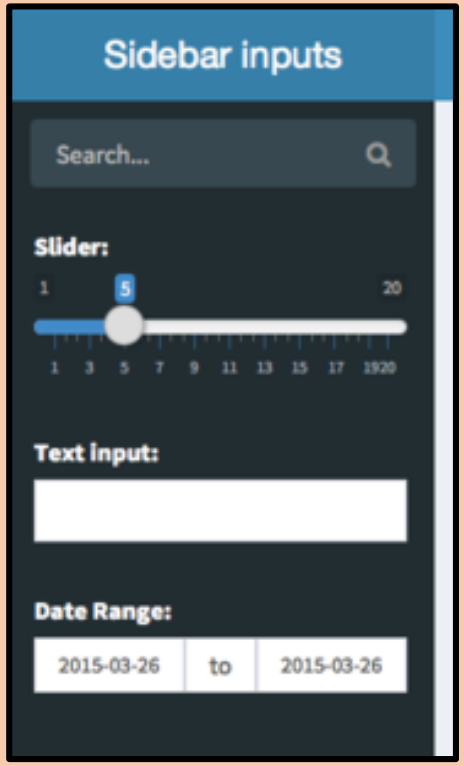
A header can have a title, dropdown menus, login, etc. Here's an example:

Shiny Dashboard

Sidebar

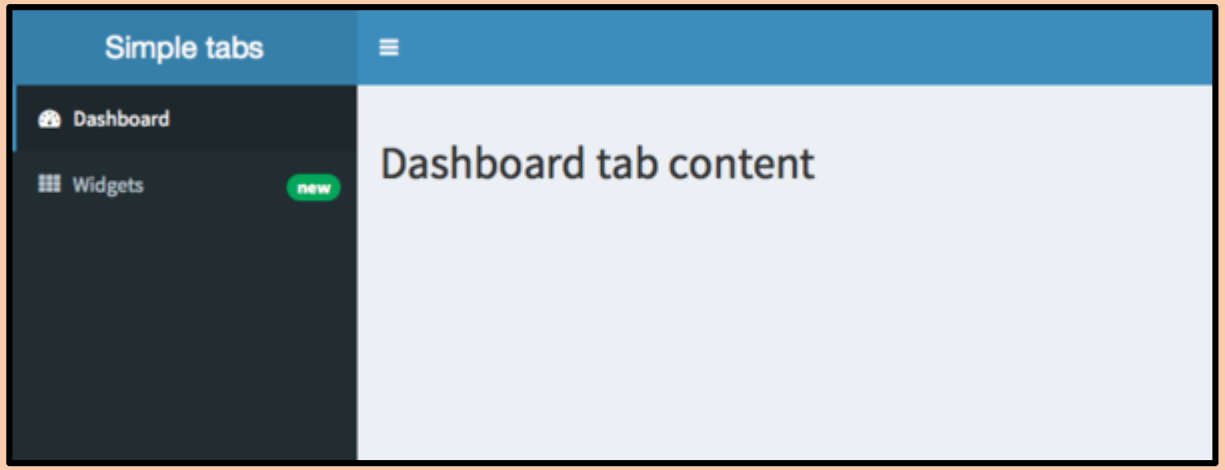
A sidebar is typically used for quick navigation. It can contain menu items that behave like tabs in a tabPanel, as well as Shiny inputs, like sliders and text inputs.





Shiny Dashboard

Sidebar menu items and tabs
Links in the sidebar can be used like tabPanels from Shiny. That is, when you click on a link, it will display different content in the body of the dashboard. Here is an example of a simple tabPanel:



- `sidebarMenu`:
- contains menu Items

- `menuItem`:
- side bar item into dashboard

- `menuSubItem`:
- sub items inside menuItem

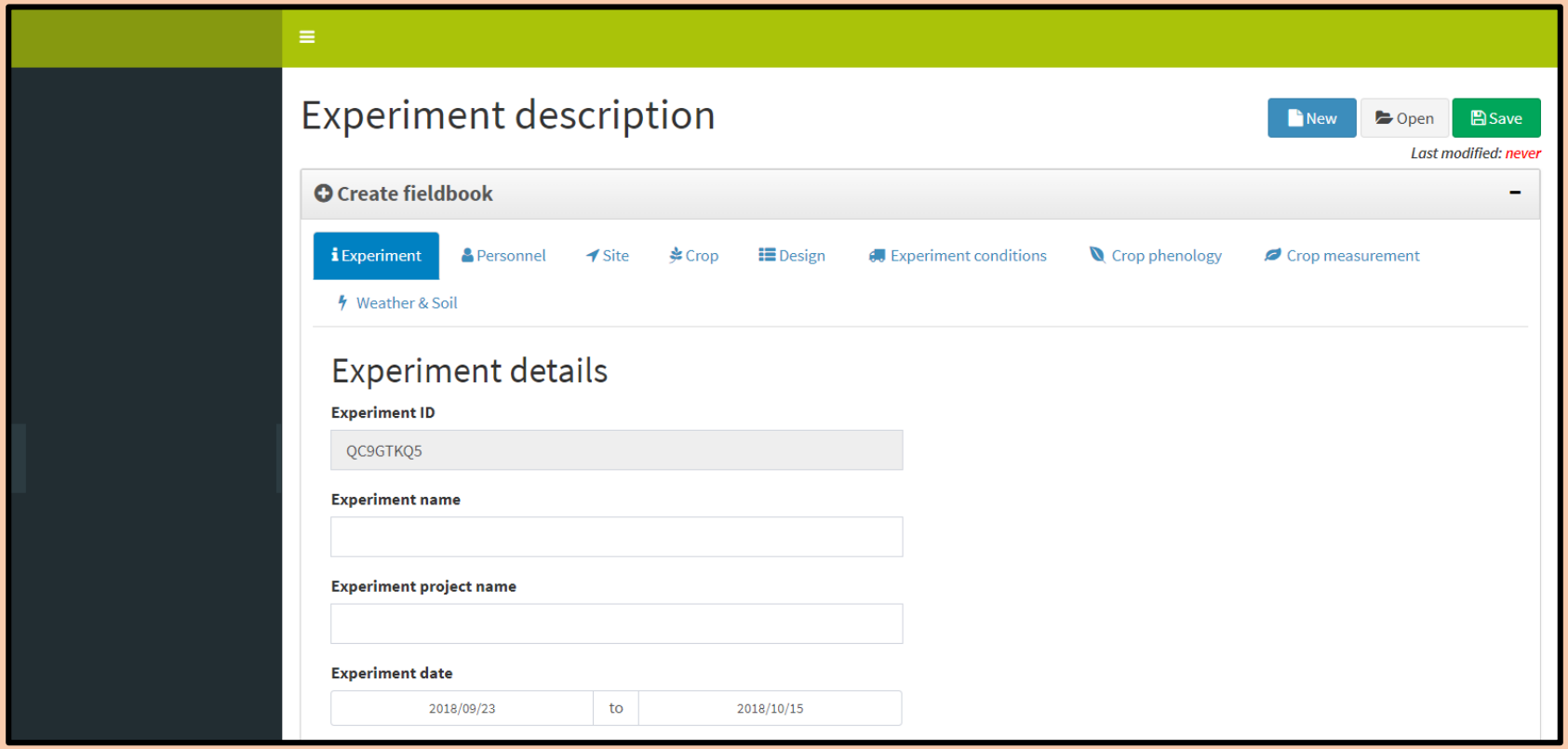
ui.R server.R

```r
#ShinyDashboard

library(shiny)
library(shinydashboard)


# Define UI for application
dashboardPage(
  dashboardHeader(title = "Title here"),
  dashboardSidebar(
    # SIDE BAR ITEMS GO HERE
    sidebarMenu(id = "tabs",
                menuItem(text= "dashboards", icon = icon("dashboard")),
                menuItem(text = "Charts", icon = icon("bar-chart-o"),
                         menuSubItem("Sub-item 1", tabName = "subitem1"),
                         menuSubItem("Sub-item 2", tabName = "subitem2")
                )
    )

  ),
  dashboardBody(
```

Shiny Dashboard

Body

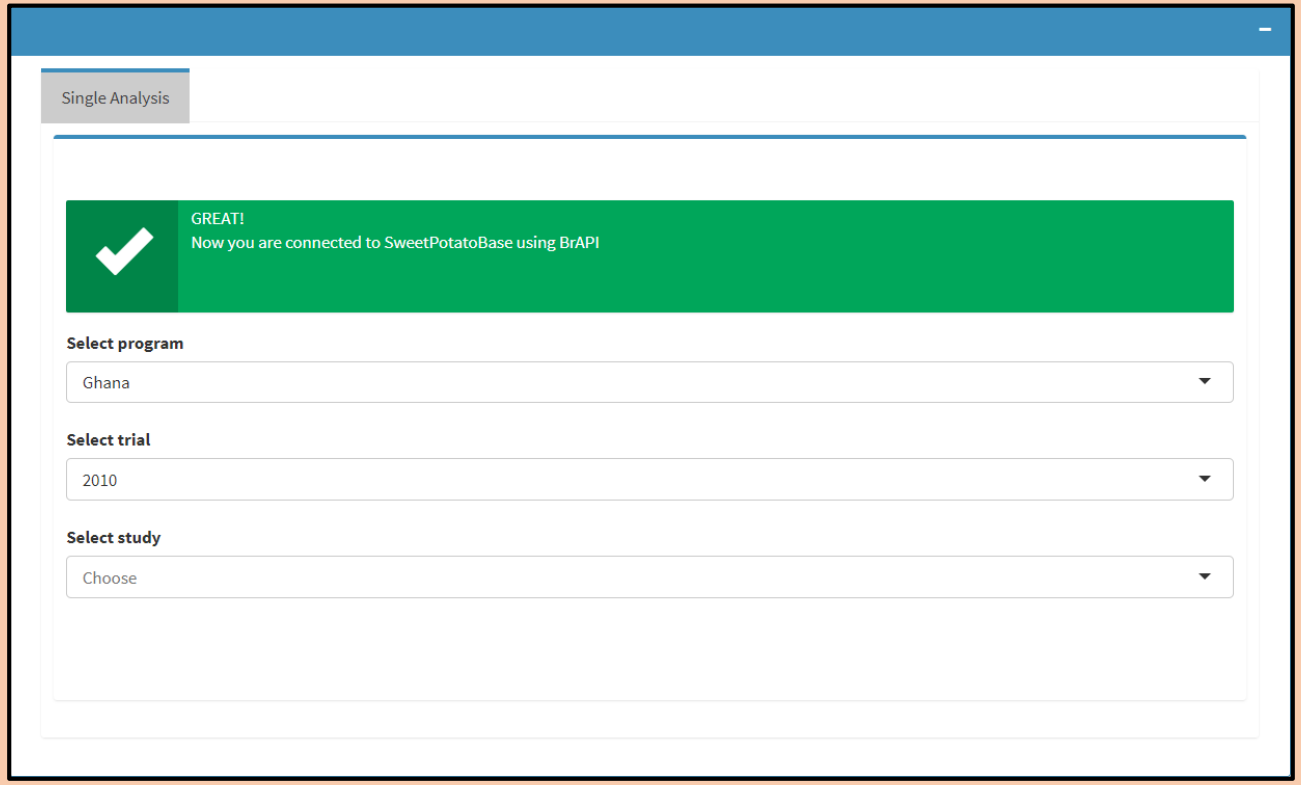
The body of a dashboard page can contain any regular Shiny content. However, if you're creating a dashboard you'll likely want to make something that's more structured. The basic building block of most dashboards is a box. Boxes in turn can contain any content.



Shiny Dashboard

Boxes

Boxes are the main building blocks of dashboard pages. A basic box can be created with the box() function, and the contents of the box can be (most) any Shiny UI content.

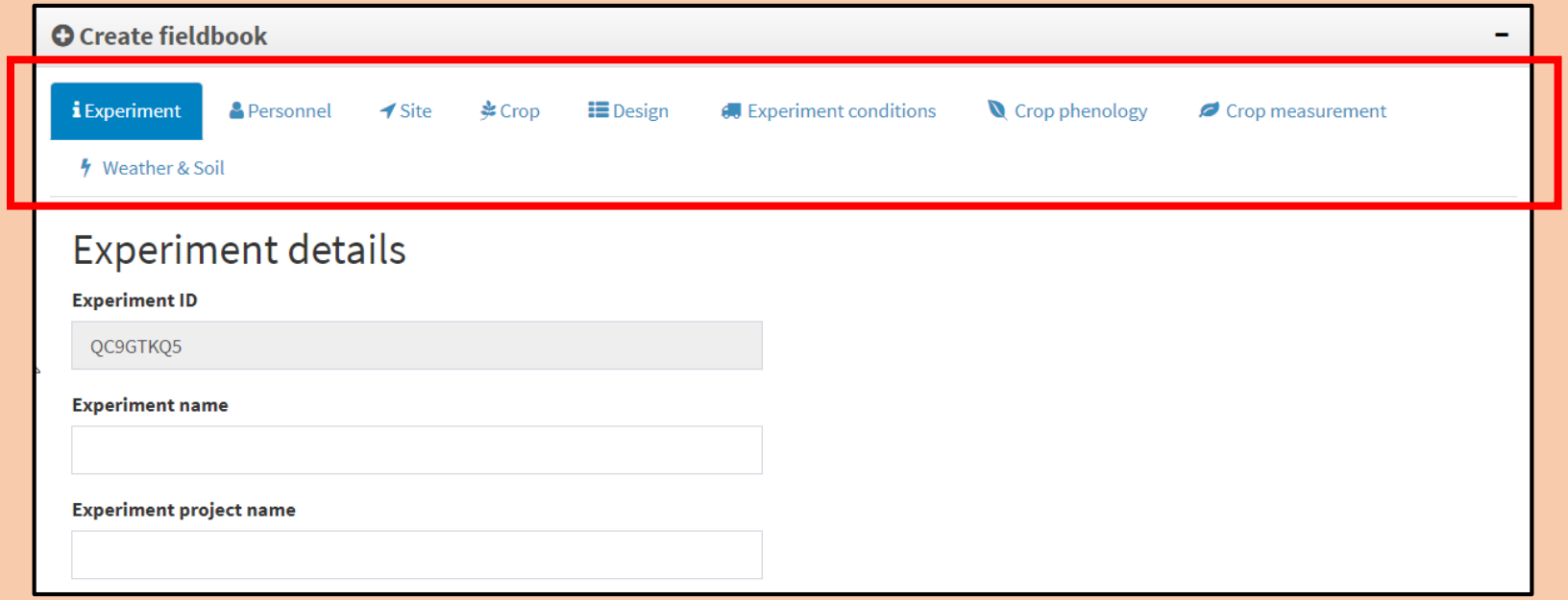


Shiny Dashboard

tabBox

If you want a box to have tabs for displaying different sets of content, you can use a tabBox.

A tabBox also has similarities to a regular box from shinydashboard, in that you can control the height, width, and title. You can also choose which side the tabs appear on, with the side argument. Note that if side="right", the tabs will be displayed in reverse order.



Shiny Dashboard

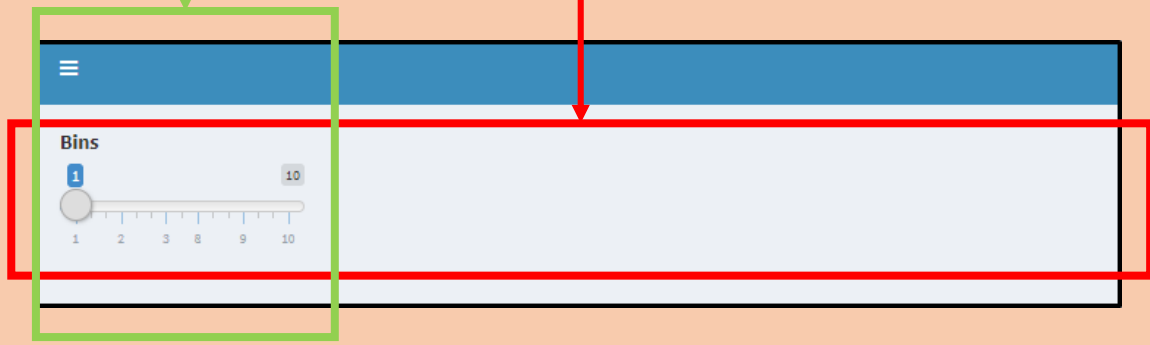
Layouts

The body can be treated as a region divided in to 12 columns of equal width, and any number of rows, of variable height. When you place a box (or other item) in the grid, you can specify how many of the 12 columns you want it to occupy.

Broadly speaking, there are two ways of laying out boxes: with a row-based layout, or with a column-based layout.

- `fluidRow`: row generator. It is the container for **columns.**
- `column:` column generator**.** It is the container for **widgets**.

```
dashboardBody(
  # OUPUT GOES HERE
  fluidRow( #GENERATE ROWS

      column(4, #GENERATE COLUMNS
             shiny::sliderInput(inputId = "bins",label = "Bins",
                                min = 1,max = 10,value=0.05)
      )#end column
  )#end fluidRow
```



Exercise 4

Part IV
How to customize appearance

Shiny Dashboard

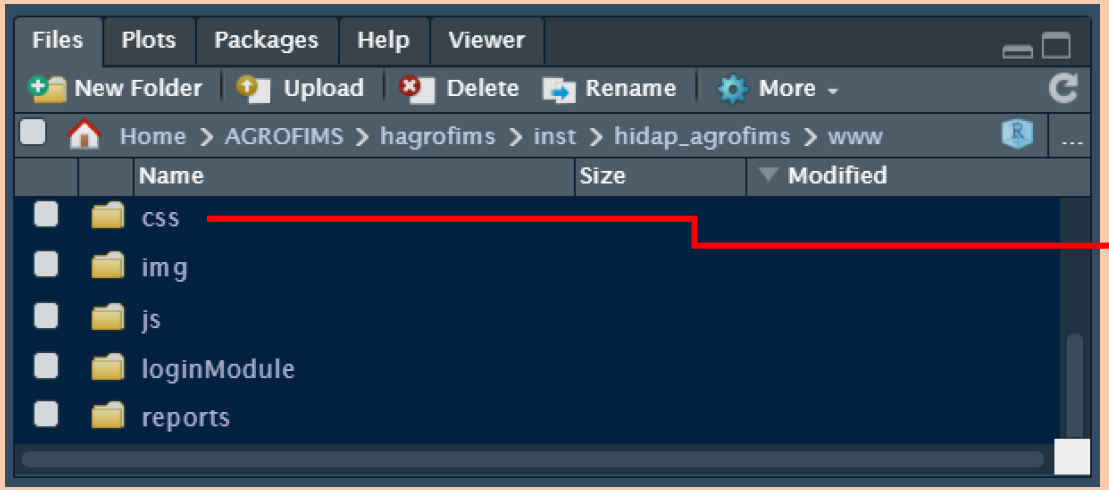
Skins

There are a number of color themes, or skins. The default is blue, but there are also black, purple, green, red, and yellow. You can choose which theme to use with dashboardPage(skin = "**blue**"), dashboardPage(skin = "black"), and so on.

Shiny Dashboard

CSS

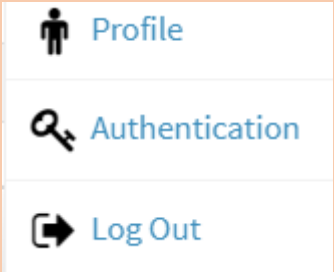
You can add custom CSS to your app by creating a **www/** subdirectory to your app and adding a CSS file there. Suppose, for example, you want to change the title font of your dashboard to the same font as the rest of the dashboard, so that it looks like this:

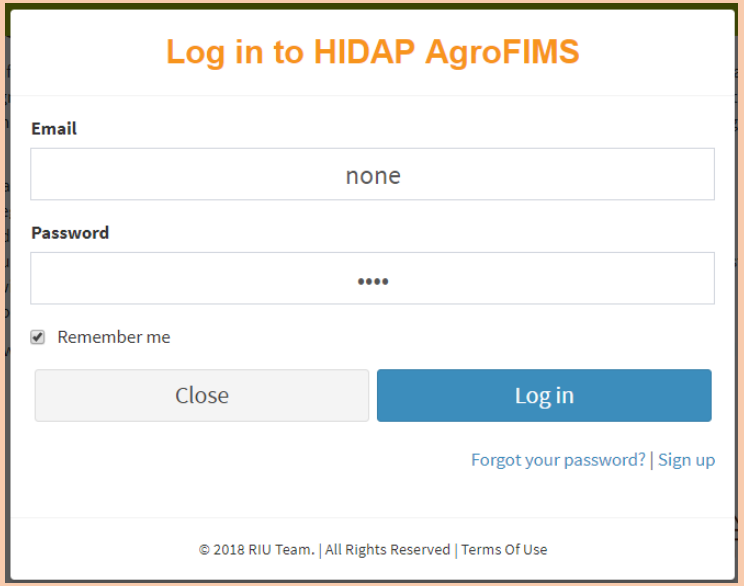


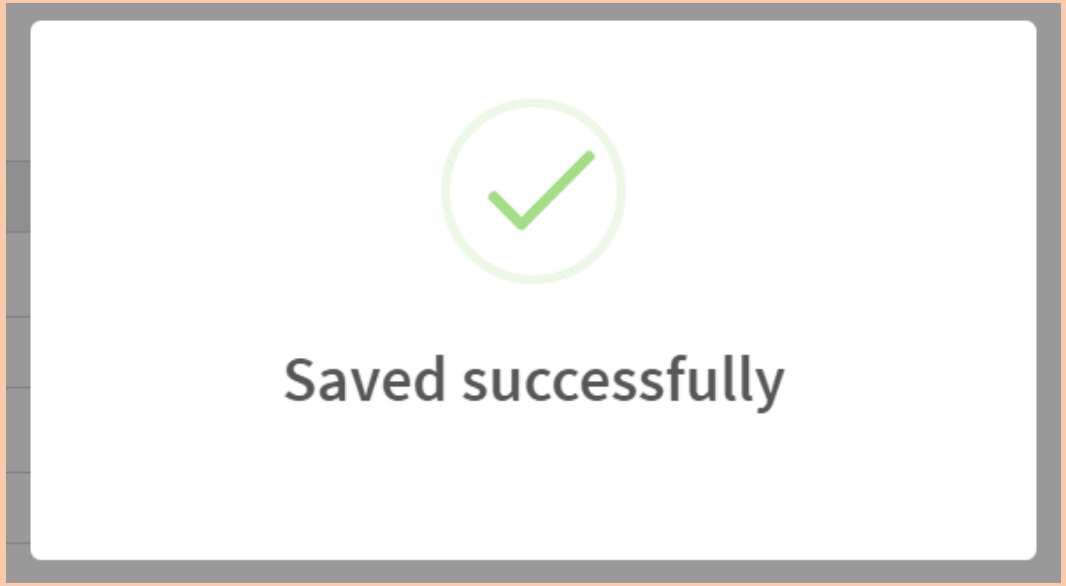

```
/* Skin */
.skin-blue .main-header .navbar .dropdown-menu li a {
    color: #3c8dbc;
}
```

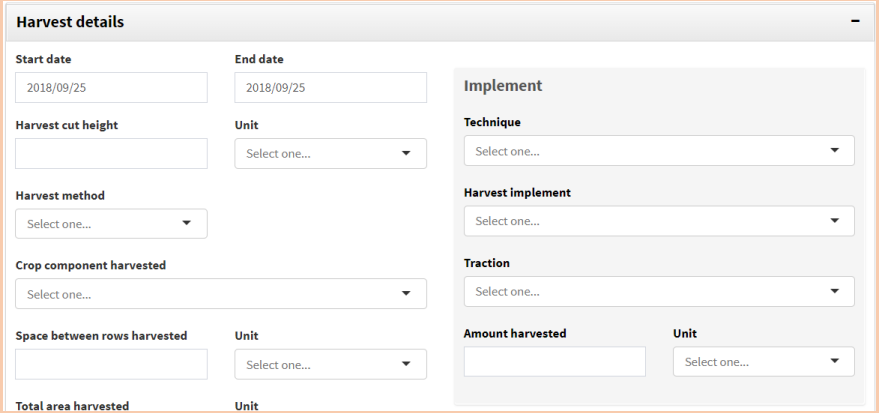
Shiny Dashboard

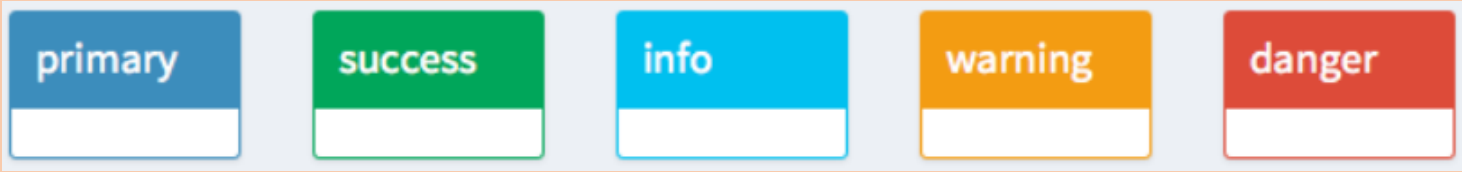
More examples…

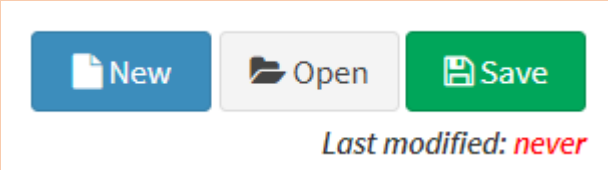


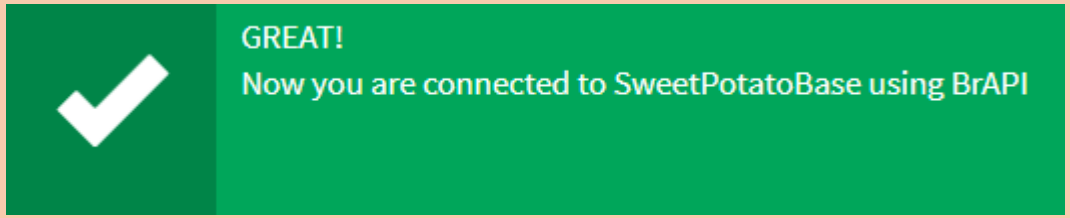












Part V
Sharing apps

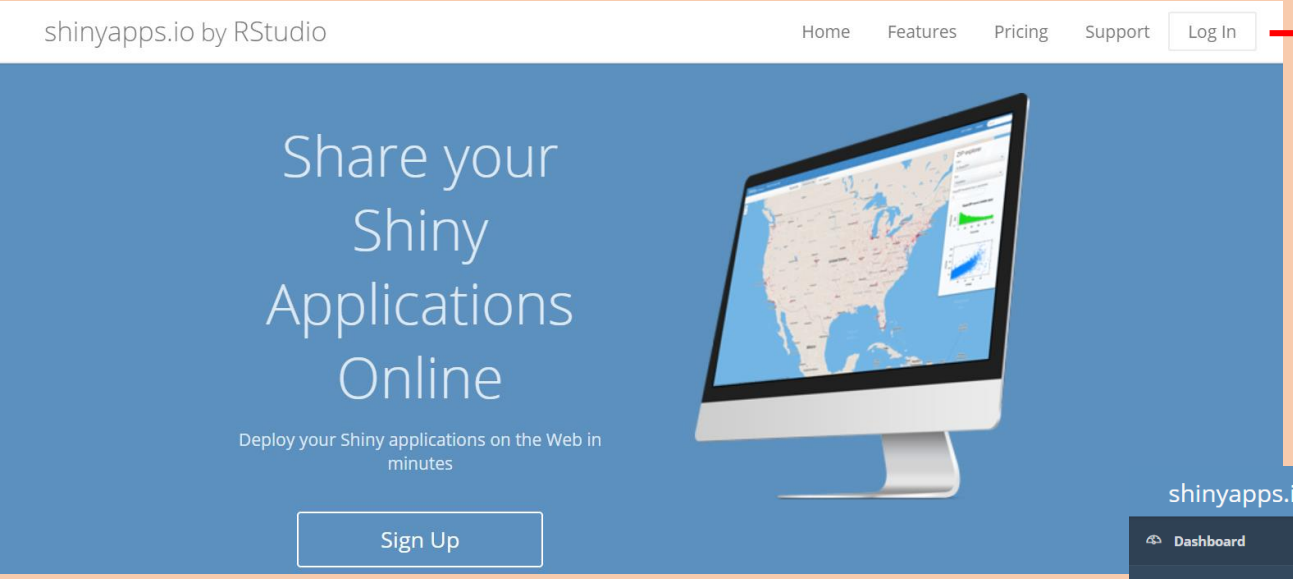
Put Shiny Web Apps Online

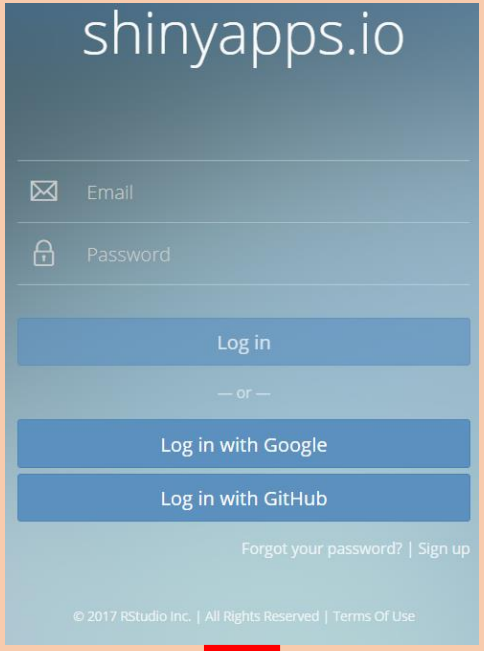
RStudio lets you put shiny web applications and interactive documents online in the way that works best for you.

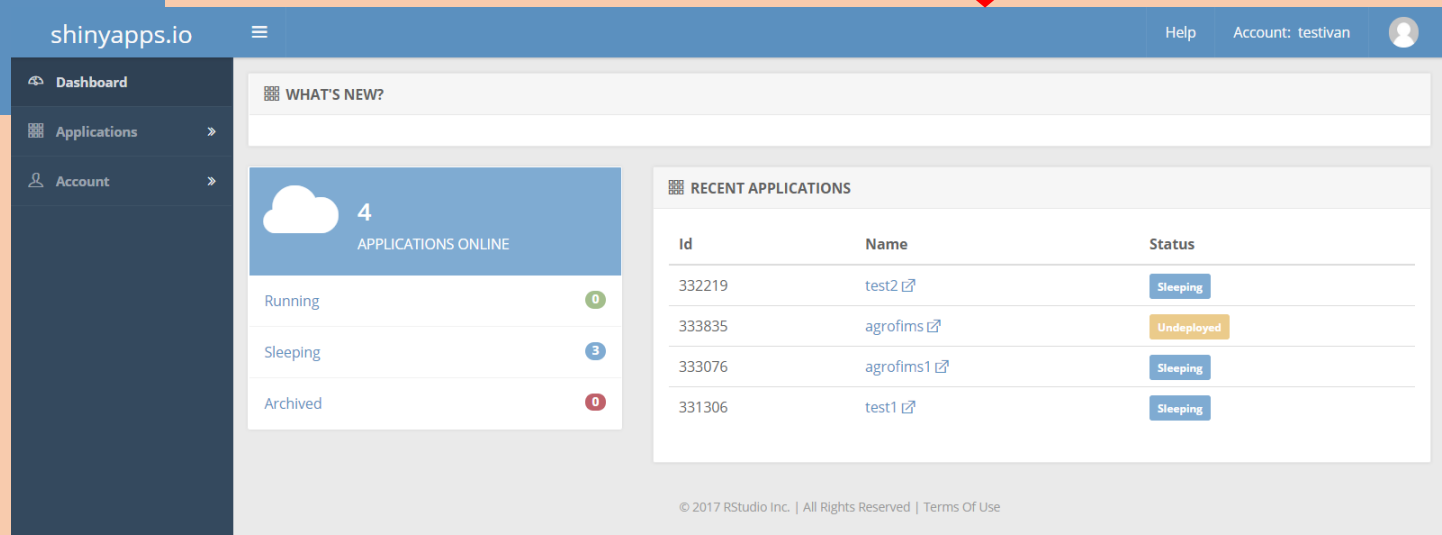
Category	Description	RStudio Connect	Shiny Server Pro	Shiny Server Open Source	Shinyapps.io
Overview	Commercial License (not AGPL)	●	●		●
	RStudio Support	●	●		●
	Deploy Shiny applications to the Web	●	●	●	●
	Push-button publishing from RStudio IDE	●			●
	Deploy and access shiny apps, dashboards, R Markdown reports, static plots, and APIs in one place	●			

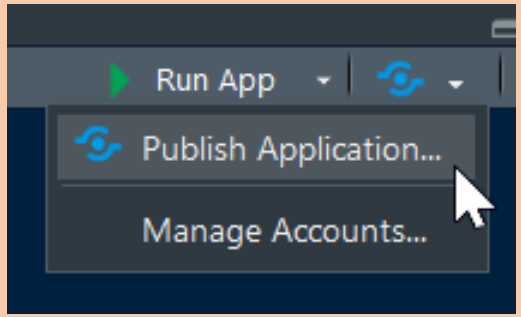
shinyapps.io

You don't need to own a server or know how to configure a firewall to deploy and manage your applications in the cloud. No hardware, installation.



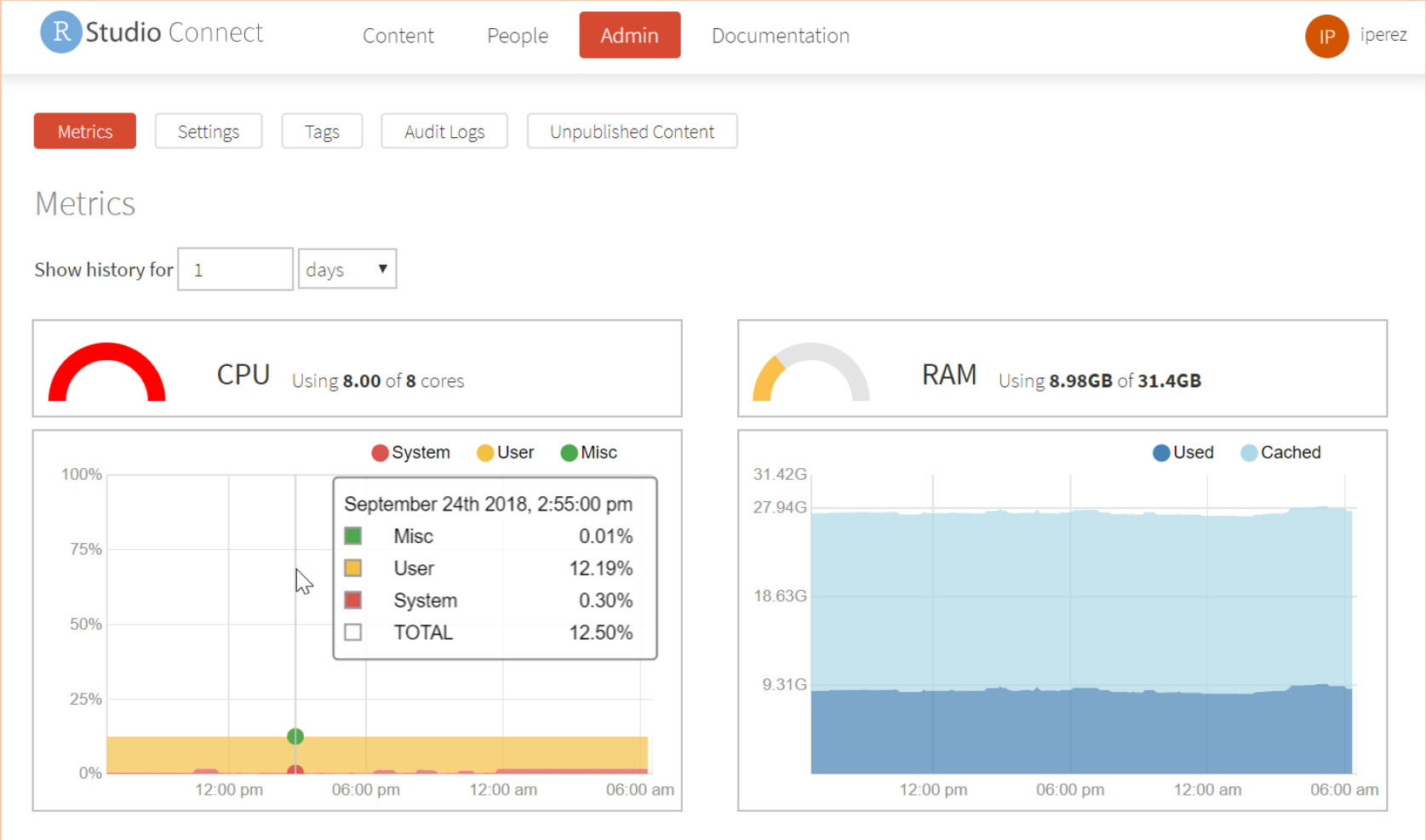






RStudio Connect

RStudio Connect is a new publishing platform for the work your teams create in R. Share Shiny applications, R Markdown reports, Plumber APIs, dashboards, plots, and more in one convenient place.



Thank you