Shiny : : GUÍA RÁPIDA

Básicos

Una **Shiny App** es una página web (**UI**) conectada a una computadora que ejecuta una sesión de R en vivo (**Server**)

Los usuarios pueden manipular la UI, lo cual lleva al server (*servidor*) a enviar una actualización de la UI exhibida (mediante ejecución de código en R).

PLANTILLA DE LA APP

Empieza a escribir una nueva Shiny App con esta plantilla. Para tener una vista previa de la app ejecuta el siguiente código.

```
library(shiny)
ui <- fluidPage()
server <- function(input, output){}
shinyApp(ui = ui, server = server)
```

- ui - funciones R anidadas que ensamblan la interfaz de usuario HTML de tu app
- server - una función con instrucciones de como construir y reconstruir los objetos mostrados en la UI
- shinyApp - combina ui y server en una app. Envuelvelos con runApp() si estás llamando desde un script o dentro de una función

COMPARTE TU APP de tres formas

1. Aloja tu app en http://shinyapps.io, un servicio en la nube de RStudio. Para eso:
   - Crea una cuenta gratuita o profesional en http://shinyapps.io
   - Cliquea en botón de publicar en la RStudio IDE o ejecuta **rsconnect::deployApp**(«<ruta a la carpeta>»)
2. **Adquiere RStudio Connect,** una plataforma para publicar en R y Python
   www.rstudio.com/products/connect/
3. **Construye tu propio servidor Shiny**
   www.rstudio.com/products/shiny-server/

RStudio

Construir una App

Añade entradas a la UI con funciones *input()

Añade salidas con funciones *output()

Instruye al servidor cómo procesar las salidas con R en la función *server* de la siguiente forma:

1. Refiere a salidas con **output$<id>**
2. Refiere a entradas with **input$<id>**
3. Incluye el código dentro de una función **render*()** antes de guardar a output

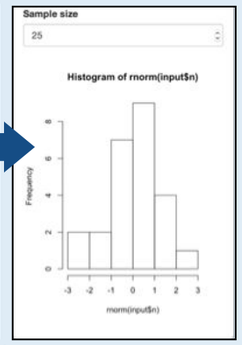
Completa la plantilla añadiendo argumentos a fluidPage() y un cuerpo a la función *server*.

```
library(shiny)

ui <- fluidPage(
  numericInput(inputId = "n",
    "Tamaño muestra", value = 25),
  plotOutput(outputId = "hist")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}

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



Guarda tu plantilla como **app.R**. O si prefieres, puedes dividir tu plantilla en dos archivos **ui.R** y **server.R**.

```
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Tamaño muestra", value = 25),
  plotOutput(outputId = "hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui = ui, server = server)
```

```
# ui.R
fluidPage(
  numericInput(inputId = "n",
    "Tamaño muestra", value =
25),
  plotOutput(outputId = "hist")
)
```

```
# server.R
function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
```

ui.R contiene todo lo que guardarías en la UI.

server.R incluye lo que quieres guardar en la función *server*.

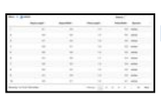
No hay necesidad de llamar a **shinyApp()**.

Guarda tu app en una carpeta como un archivo **app.R** (o un archivo **server.R** y **ui.R**) más los archivos extra

El nombre de la carpeta es el nombre de la app
(opcional) define objetos disponibles para ambos ui.R y server.R
(opcional) usado en el modo *showcase*
(opcional) datos, *scripts*, etc.
(opcional) carpeta de archivos para compartir con navegadores web (imágenes, CSS, .js, etc.). Debe llamarse "**www**".

Inicia la app con **runApp(<ruta a carpeta>)**

Salidas - Las funciones render*() y *Output() trabajan juntas para agregar los valores de salida (*outputs*) de R en la UI



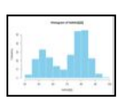
DT::**renderDataTable(**expr, options, callback, escape, env, quoted**)**

trabaja con

dataTableOutput(outputId, icon, …)

renderImage(expr, env, quoted, deleteFile)

imageOutput(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)

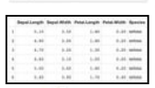


renderPlot(expr, width, height, res, …, env, quoted, func)

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)

foo

renderText(expr, env, quoted, func)

textOutput(outputId, container, inline)

renderUI(expr, env, quoted, func)

& **uiOutput**(outputId, inline, container, …)
htmlOutput(outputId, inline, container, …)

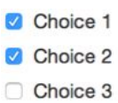
Entradas

Obtén los valores de entrada (*inputs*) del usuario

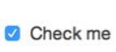
Accede el valor actual de un *input* con **input$<inputId>**. Los *inputs* son **reactivos**.

actionButton(inputId, label, icon, …)

actionLink(inputId, label, icon, …)



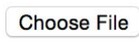
checkboxGroupInput(inputId, label, choices, selected, inline)



checkboxInput(inputId, label, value)

dateInput(inputId, label, value, min, max, format, startview, weekstart, language)

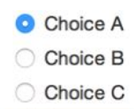
dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)



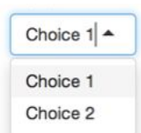
fileInput(inputId, label, multiple, accept)

numericInput(inputId, label, value, min, max, step)

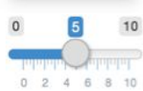
passwordInput(inputId, label, value)



radioButtons(inputId, label, choices, selected, inline)



selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (también: selectizeInput())



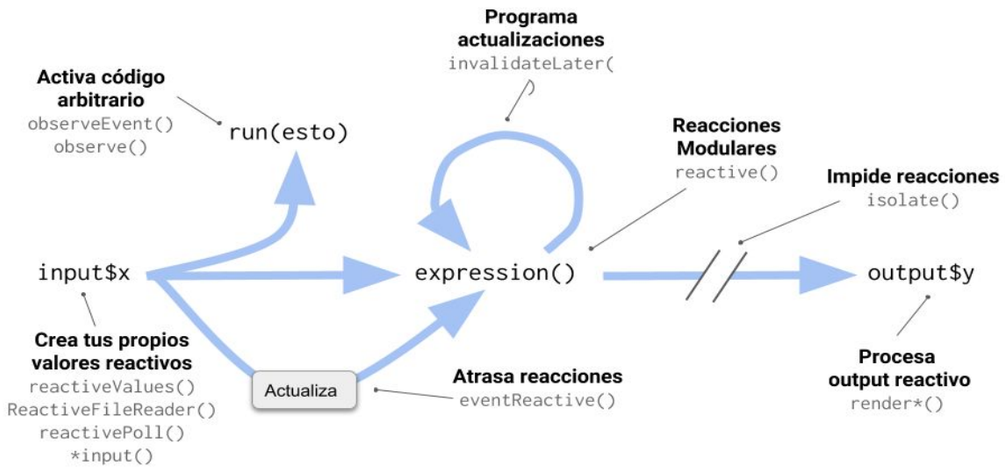
sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)

submitButton(text, icon)
(Impide reacciones en toda la app)

textInput(inputId, label, value)

Reactividad

Los valores reactivos trabajan conjuntamente con funciones reactivas. Llama a los valores reactivos desde los argumentos de una de estas funciones para evitar el error Operation not allowed without an active reactive context.



CREAR TUS PROPIOS VALORES REACTIVOS

```
# example snippets

ui <- fluidPage(
 textInput("a","","A")
)

server <-
function(input,output){
 rv <- reactiveValues()
 rv$number <- 5
}
```

funciones *Input()
(mirar primera hoja)
reactiveValues(...)

Cada función de *input* crea un valor reactivo guardado como **input$<inputId>**

reactiveValues() crea una lista de valores reactivos cuyos valores puedes configurar.

PREVENIR REACCIONES

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 textOutput("b")
)

server <-
function(input,output){
 output$b <-
  renderText({
   isolate({input$a})
  })
}

shinyApp(ui, server)
```

isolate(expr)

Ejecuta un bloque de código.
Devuelve una copia **no-reactiva** de los resultados.

MODULARIZAR REACCIONES

```
ui <- fluidPage(
 textInput("a","","A"),
 textInput("z","","Z"),
 textOutput("b"))

server <-
function(input,output){
 re <- reactive({
  paste(input$a,input$z)})
 output$b <- renderText({
   re()
  })
}
shinyApp(ui, server)
```

reactive(x, env, quoted, label, domain)

Crea una **expresión reactiva** que

- Se almacena en **caché** para reducir costos computacionales
- Permite ser llamada por otro código
- Notifica sus dependencias cuando ha sido validada

Llama la expresión empleando la sintaxis de funciones, ej. re()

REPRODUCIR *OUTPUTS* REACTIVOS

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 textOutput("b")
)

server <-
function(input,output){
 output$b <-
  renderText({
   input$a
  })
}

shinyApp(ui, server)
```

funciones render*()
(mirar primera hoja)

Construye un objeto para mostrar. Volverá a ejecutar el código para reconstruir el objeto, siempre que el valor reactivo cambie el código.

Guarda el resultado a **output$<outputId>**

DISPARAR EJECUCIÓN DE CÓDIGO ARBITRARIO

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 actionButton("go","Go")
)

server <-
function(input,output){
 observeEvent(input$go,{
   print(input$a)
  })
}

shinyApp(ui, server)
```

observeEvent(eventExpr, handlerExpr, event.env, event.quoted, handler.env, handler.quoted, labe, suspended, priority, domain, autoDestroy, ignoreNULL)

Ejecuta código en el 2do argumento cuando los valores reactivos en el 1er argumento cambian. **observe()** es otra alternativa a esta función.

ATRASAR REACCIONES

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 actionButton("go","Go"),
 textOutput("b")
)

server <-
function(input,output){
 re <- eventReactive(
  input$go,{input$a})
 output$b <- renderText({
   re()
  })
}

shinyApp(ui, server)
```

eventReactive(eventExpr, valueExpr, event.env, event.quoted, value.env, value.quoted, label, domain, ignoreNULL)

Crea expresiones reactivas con código en el segundo argumento que solo se invalida cuando valores reactivos en el primer argumento cambian.

UI – La UI de una app es un documento HTML.

Usa funciones de Shiny para reunir este HTML con R.

```
fluidPage(
 textInput("a","")
)
## <div class="container-fluid">
##  <div class="form-group shiny-input-container">
##     <label for="a"></label>
##     <input id="a" type="text"
##        class="form-control" value=""/>
##   </div>
## </div>
```

Devuelve HTML

Añade elementos HTML estáticos con **tags**, una lista de funciones paralelas a tags HTML comunes ej.. **tags$a()**. Argumentos sin nombres son transpasados dentro del tag; argumentos con nombre se convierten en atributos del tag.

tags$a	tags$data	tags$h6	tags$nav	tags$span
tags$abbr	tags$datalist	tags$head	tags$noscript	tags$strong
tags$address	tags$dd	tags$header	tags$object	tags$style
tags$area	tags$del	tags$hgroup	tags$ol	tags$sub
tags$article	tags$details	tags$hr	tags$optgroup	tags$summary
tags$aside	tags$dfn	tags$HTML	tags$option	tags$sup
tags$audio	tags$div	tags$i	tags$output	tags$table
tags$b	tags$dl	tags$iframe	tags$p	tags$tbody
tags$base	tags$dt	tags$img	tags$param	tags$td
tags$bdi	tags$em	tags$input	tags$pre	tags$textarea
tags$bdo	tags$embed	tags$ins	tags$progress	tags$tfoot
tags$blockquote	tags$eventsource	tags$kbd	tags$q	tags$th
tags$body	tags$fieldset	tags$keygen	tags$ruby	tags$thead
tags$br	tags$figcaption	tags$label	tags$rp	tags$time
tags$button	tags$figure	tags$legend	tags$rt	tags$title
tags$canvas	tags$footer	tags$li	tags$s	tags$tr
tags$caption	tags$form	tags$link	tags$samp	tags$track
tags$cite	tags$h1	tags$mark	tags$script	tags$u
tags$code	tags$h2	tags$map	tags$section	tags$ul
tags$col	tags$h3	tags$menu	tags$select	tags$var
tags$colgroup	tags$h4	tags$meta	tags$small	tags$video
tags$command	tags$h5	tags$meter	tags$source	tags$wbr

Los tags más comunes tienen *wrapper functions* y no necesitas prefijar sus nombres con **tags$**

```
ui <- fluidPage(
 h1("Header 1"),
 hr(),
 br(),
 p(strong("bold")),
 p(em("italic")),
 p(code("code")),
 a(href="", "link"),
 HTML("<p>Raw html</p>")
)
```

Para incluir un archivo CSS usa **includeCSS()**, o

1. Agrega el archivo en la sub-carpeta **www**
2. Crea un link con

```
tags$head(tags$link(rel = "stylesheet",
type = "text/css", href = "<nombre archivo>"))
```

Para incluir JavaScript, usa **includeScript()** o

1. Agrega el archivo en la sub-carpeta **www**
2. Crea un link con

```
tags$head(tags$script(src = "<nombre
                archivo>"))
```

IMAGES Para incluir una imagen

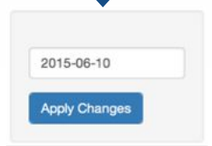
1. Agrega el archivo en la sub-carpeta **www**
2. Crea un link con

```
img(src="<nombre archivo>")
```

Diseños

Combina múltiples elementos en un "elemento único" que tiene sus propias características con una función de panel, ej.

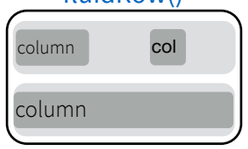
```
wellPanel(dateInput("a", ""),
    submitButton()
)
```



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

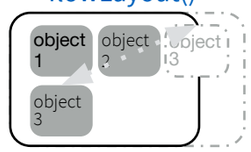
Puedes organizar paneles y elementos en un diseño con una función de diseño (*layout*). Los elementos se añaden como argumentos de las funciones de *diseño*.

fluidRow()



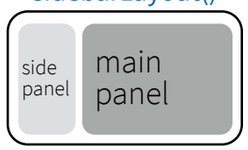

```
ui <- fluidPage(
 fluidRow(column(width = 4),
   column(width = 2, offset = 3)),
 fluidRow(column(width = 12))
)
```

flowLayout()



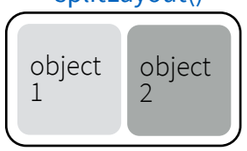

```
ui <- fluidPage(
 flowLayout( # object 1,
             # object 2,
             # object 3
 )
)
```

sidebarLayout()



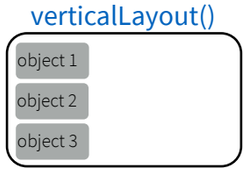

```
ui <- fluidPage(
 sidebarLayout(
  sidebarPanel(),
  mainPanel()
 )
)
```

splitLayout()



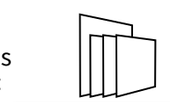

```
ui <- fluidPage(
 splitLayout( # object 1,
              # object 2
 )
)
```

verticalLayout()




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

Ordena en capas **tabPanels** unos sobre otros y navega entre ellos:




```
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"))
```