Shiny : : HOJA DE REFERENCIA

Básicos

Un app **Shiny** es una pagina web (**UI**) conectado a un computador corriendo una sesión de R (**Server**)

Usuarios pueden manipular el UI, lo cual lleva al servidor a enviar una actualización del UI, corriendo código R.

PLANTILLA DE UN APP

Empieza a escribir un nuevo app con esta plantilla. Corriendo el código en la linea de comando te dará una vista preliminar del app.

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

- **ui** - funciones R anidadas que son la base para mostrar una interfaz HTML de tu app
- **server** - una función con instrucciónes de como construir y reconstruir los objetos presentado en el UI
- **shinyApp** - combina `ui` y `server` en un app. Envuélvelos con **runApp()** si estas llamando desde un script o dentro de una función

COMPARTE TU APP

La forma mas facil de compartir tu app es desplegarlo en shinyapps.io, un servicio en la nube de RStudio

1. Crea una cuenta gratis o profesional en: **http://shinyapps.io**
2. Apreta el botón **Publish** en el IDE de RStudio IDE or corre: **rsconnect::deployApp(**"<path to directory>"**)**

Construye o compra tu propio Shiny Server
at www.rstudio.com/products/shiny-server/

Construir un App

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

Añade entradas al UI con funciones *input()

Añade salidas con funciones *output()

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

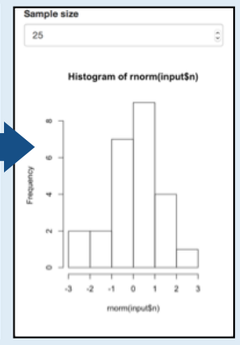
1. Refiere a salidas con **output$<id>**
2. Refiere a entradas with **input$<id>**
3. Envuelve código en una función **render*()** antes de guardar a output

```
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, divide 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 guardas a ui.

server.R termina con la función que quieres guardar a server**.**

No hace falte llamar **shinyApp()**.

Guarda tu app en una carpeta con un archivo **app.R** (o un archivo **server.R** y **ui.R**) y archivos opcionales.

El nombre de la carpeta es el nombre del 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.). Tiene que tener el nombre "**www**"

Arranca apps con `runApp(<camino a carpeta>)`

Salidas - funciones render*() y *Output() trabajan juntos para mostrar resultados R en el UI

	works with	
DT::**renderDataTable**(expr, options, callback, escape, env, quoted)		**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)
renderText(expr, env, quoted, func)		**textOutput**(outputId, container, inline)
renderUI(expr, env, quoted, func)		**uiOutput**(outputId, inline, container, …) & **htmlOutput**(outputId, inline, container, …)

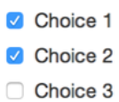
Entradas

obtén valores del usuario

Accede el valor actual de objetos de entrada con **input$<inputId>**. Valores input son **reactivos**.

actionButton(inputId, label, icon, …)

actionLink(inputId, label, icon, …)



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

checkboxInput(inputId, label, value)

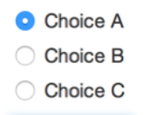
dateInput(inputId, label, value, min, max, format, startview, weekstart, language)

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

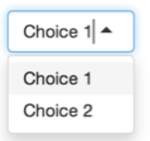
fileInput(inputId, label, multiple, accept)

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

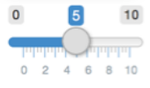
passwordInput(inputId, label, value)



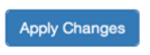
radioButtons(inputId, label, choices, selected, inline)



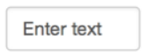
selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (tambien: selectizeInput())



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



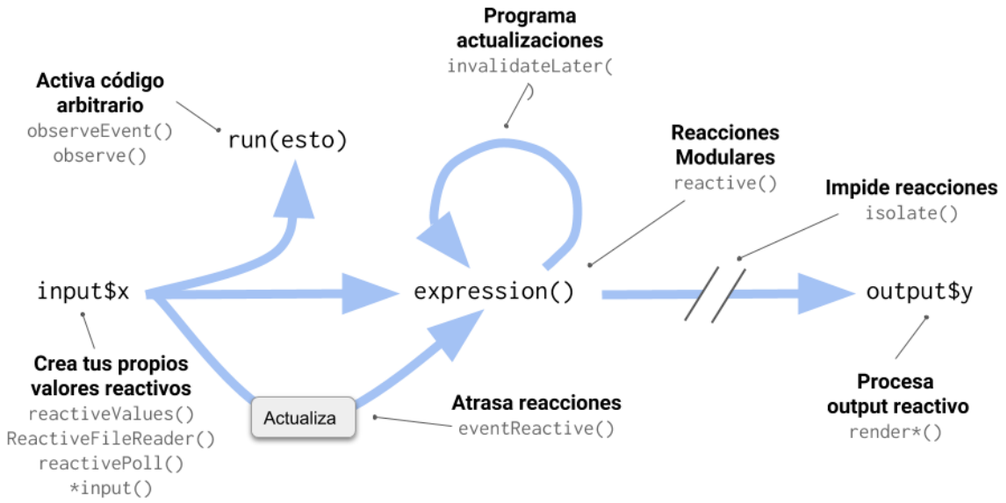
submitButton(text, icon)
(Impide reacciones en todo el app)



textInput(inputId, label, value)

Reactividad

Valores reactivos trabajan juntos con funciones reactivas. Llama un valor reactivo desde los argumentos de una de estas funciones para evitar el error `Operation not allowed without an active reactive context.`



CREA TUS PROPIOS VALORES REACTIVOS

```
# example snippets


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


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

funciones ***Input()**
(mira primera pagina)

reactiveValues(...)

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

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

IMPIDE 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)

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

MODULARISA 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 **expression reactiva** que

- hace **cache** del valor para reducir computación
- se puede llamar por otro código
- notifica sus dependencias cuando ha sido validado

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

PROCESA OUTPUT REACTIVO

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

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

shinyApp(ui, server)
```

funciones render*()
(mira primera pagina)

Construye un objeto para mostrar. Corre el código en el body cada vez que un valor reactivo en el código cambia.

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

ACTIVA CODIGO 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)

Corre código en el segundo argumento cuando valores reactivos en el primer argumento cambia. Mira **observe()** para una alternativa.

ATRASA 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 - El ui de un app es un documento HTML.

Usa funciones de Shiny's 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>
```

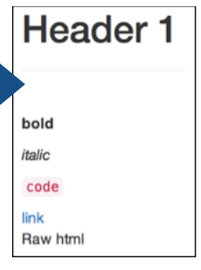
Devuelve HTML

Añade elementos HTML estáticos con **tags**, una lista de funciones paralelas a tags HTML comunes ej.. **tags$a()**. Argumentos sin nombres son pasados 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 mas 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. Pon el archivo en la sub-carpeta **www**
2. Crea un eslabón con

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

Para incluir JavaScript, usa **includeScript()** o
1. Pon el archivo en la sub-carpeta **www**
2. Crea un eslabón con

```
tags$head(tags$script(src = "<file name>"))
```

IMAGES

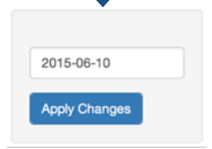
Para incluir una imagen
1. Pon el archivo en la sub-carpeta **www**
2. Crea un eslabón con

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

Diseños

Combina multiples elementos en un "solo elemento" que tiene sus propios propiedades con una función de panel, ej.

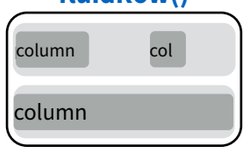
```
wellPanel(dateInput("a", ""),
          submitButton()
)
```



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

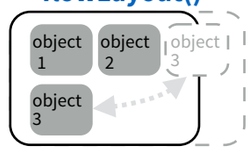
Organiza paneles y elementos en un diseño con una función de layout. Añade elementos como argumentos de las funciones *layout*.

fluidRow()



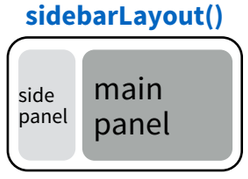

```
ui <- fluidPage(
  fluidRow(column(width = 4),
    column(width = 2, offset = 3)),
  fluidRow(column(width = 12))
)
```

flowLayout()



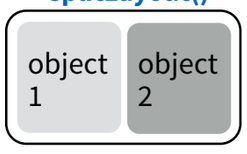

```
ui <- fluidPage(
  flowLayout( # object 1,
              # object 2,
              # object 3
  )
)
```

sidebarLayout()



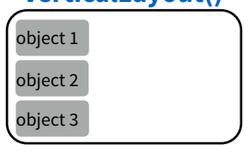

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

splitLayout()




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

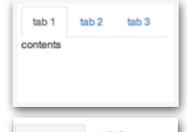
verticalLayout()




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

Sobrepone capas de **tabPanels** uno sobre otro y navega entre ellos con:

```
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"))
```