



Modularizando una Shiny app



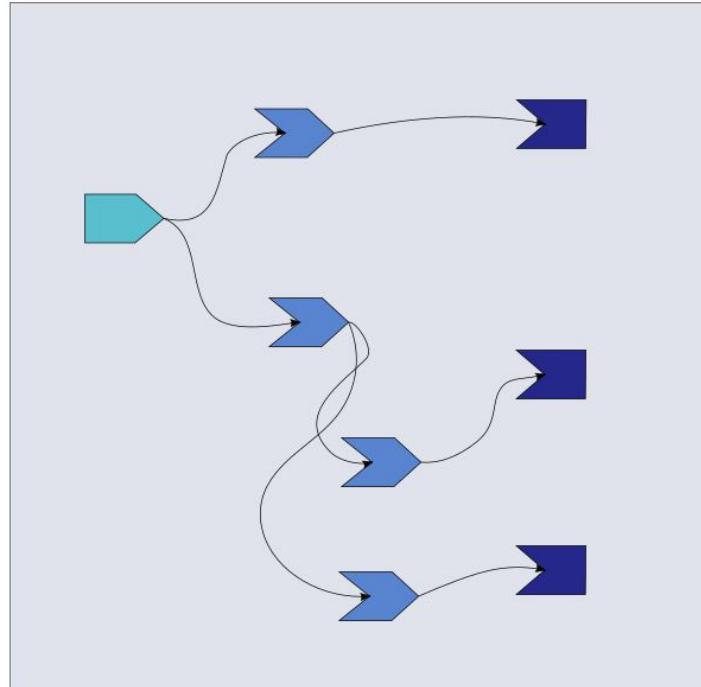
Federico Rivadeneira
federico@apppsilon.com

Contenidos

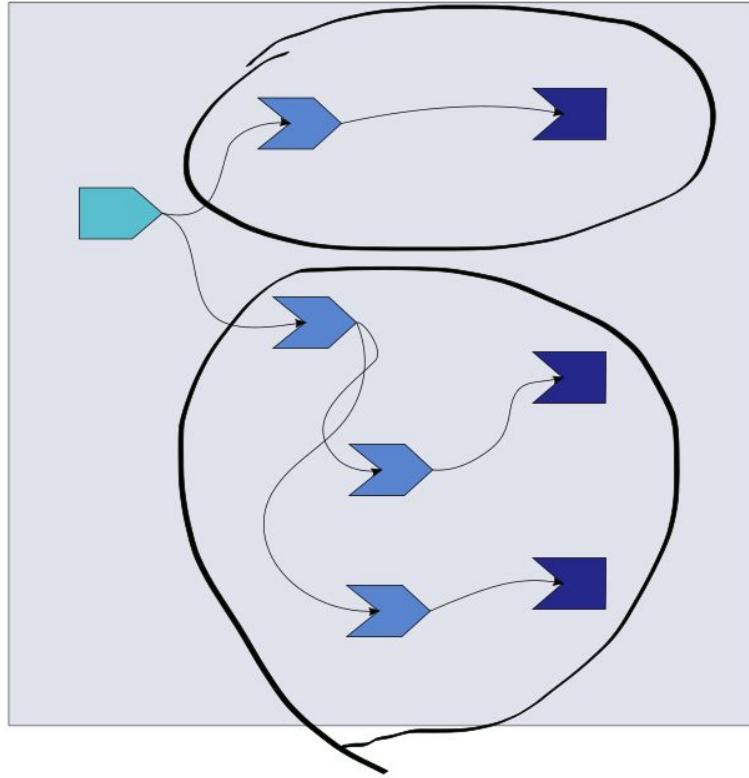
1. ¿Qué son los módulos y por qué usarlos?
2. Diferentes tipos de módulos
3. Módulos en Shiny
 - a. Ejemplo
 - b. Comunicación entre módulos de shiny
4. Modularizando código con box
5. Más info

¿Qué son los módulos y por qué usarlos?

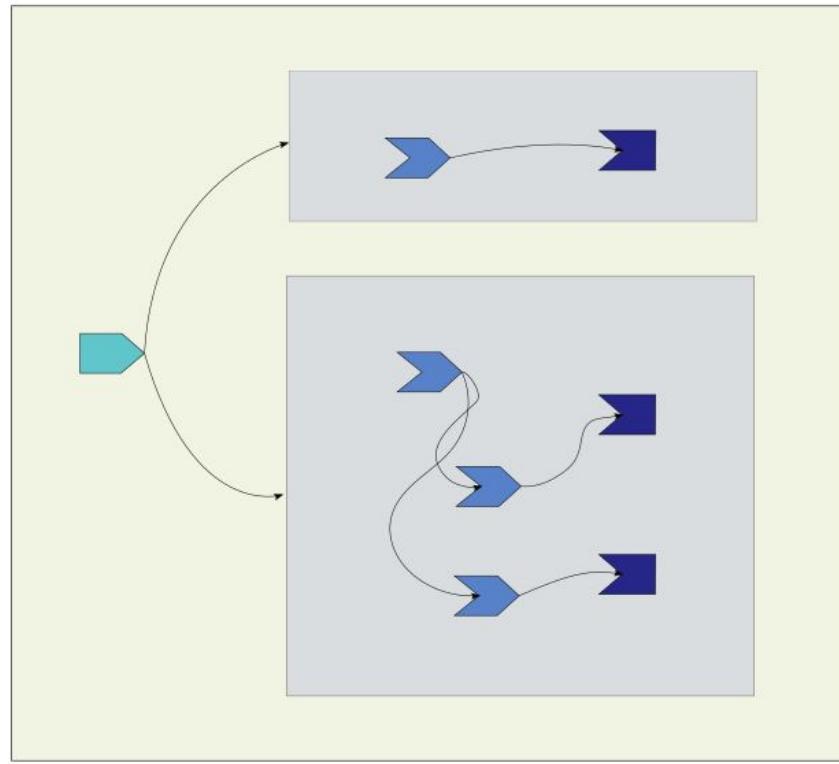
Los módulos, a grandes rasgos, son una **colección** de funciones/métodos y constantes.



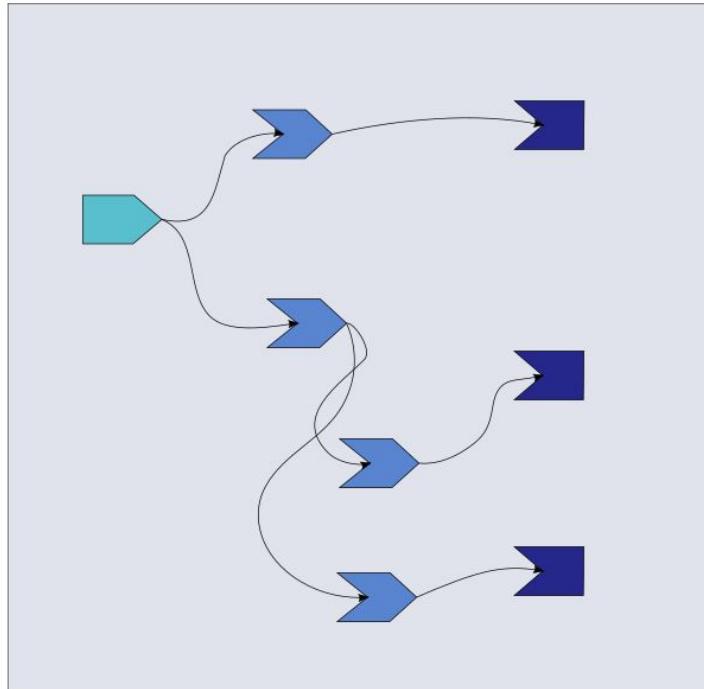
Identificar y aislar/encapsular los diferentes componentes
y subcomponentes lógicos de una aplicación.

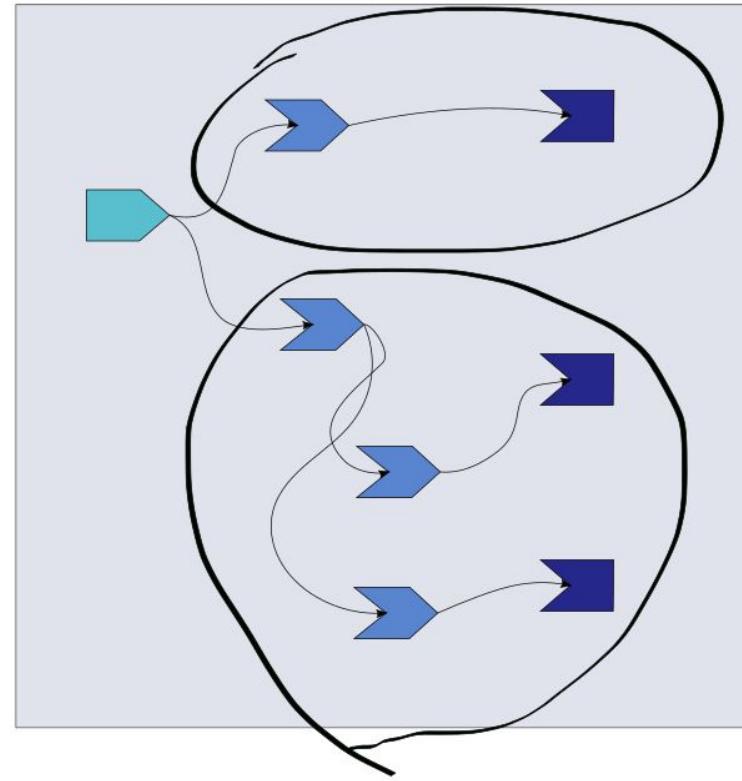


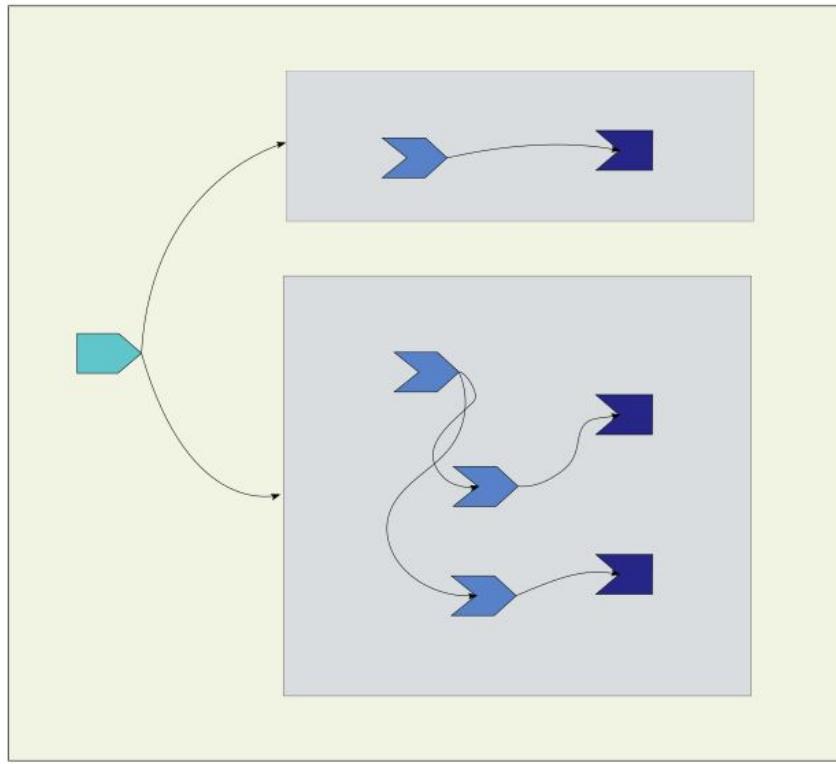
Nos permiten reutilizar partes de nuestro
código/aplicación de una forma consistente



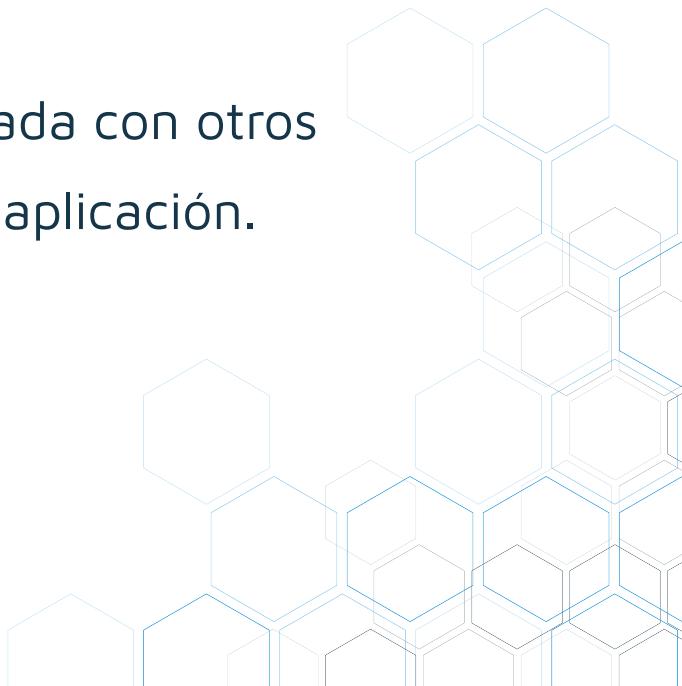
Repasemos

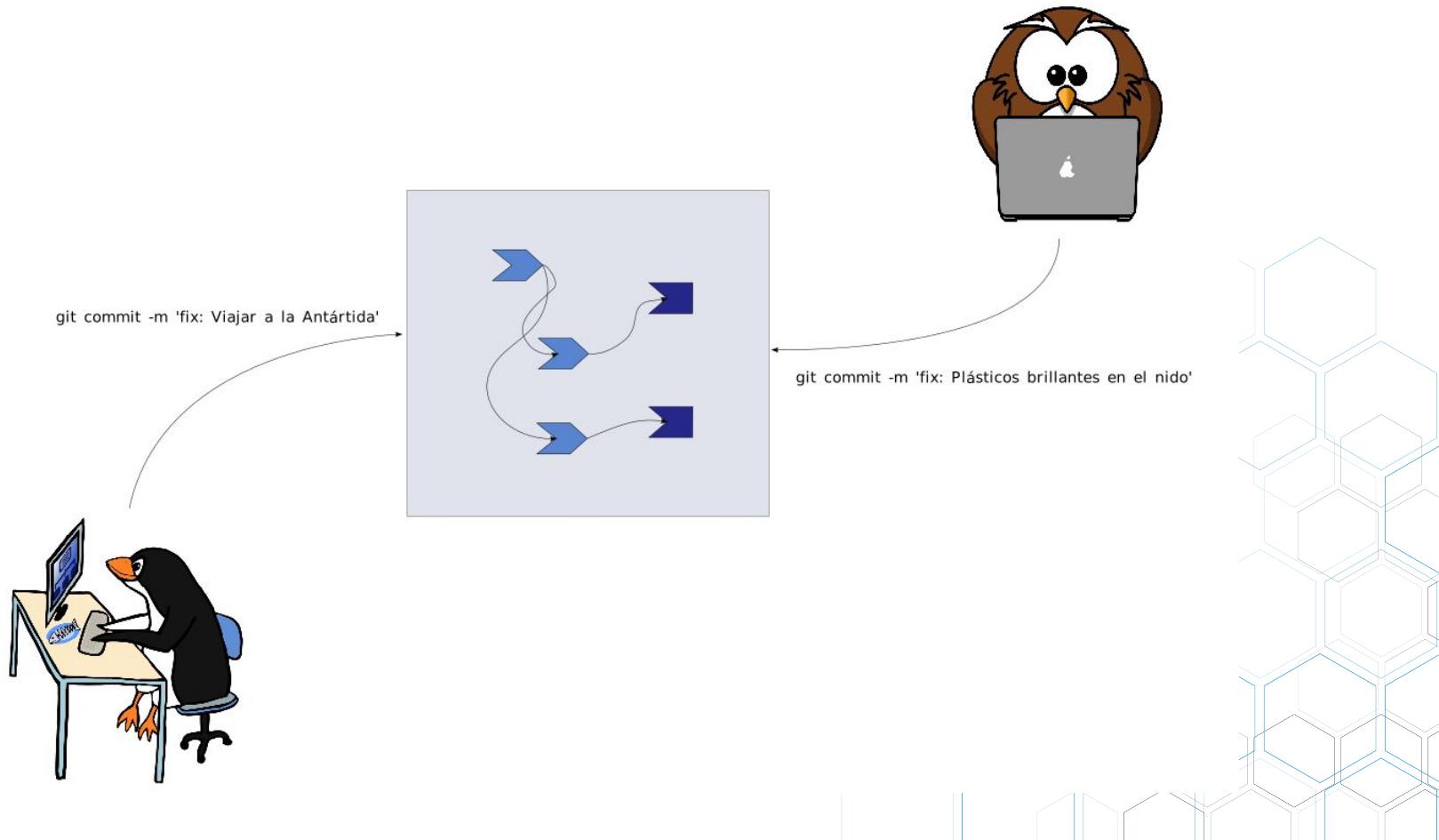






Trabajar de forma aislada y concertada con otros
desarrolladores sobre una misma aplicación.

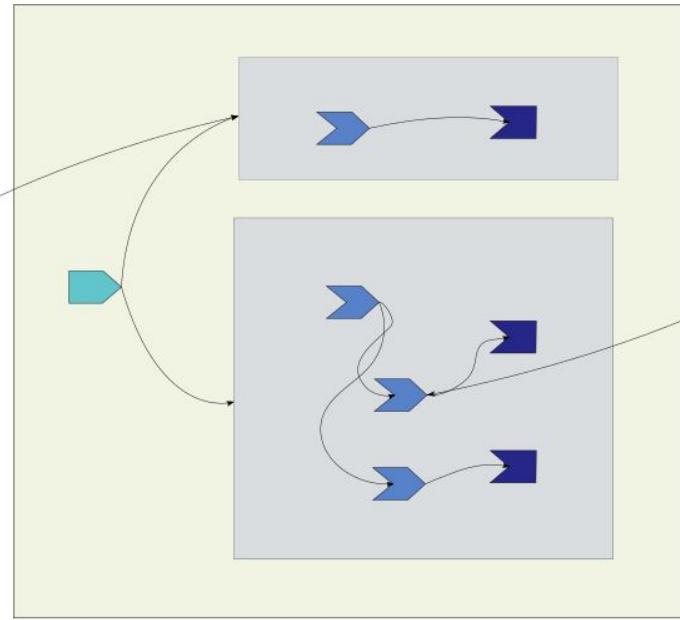




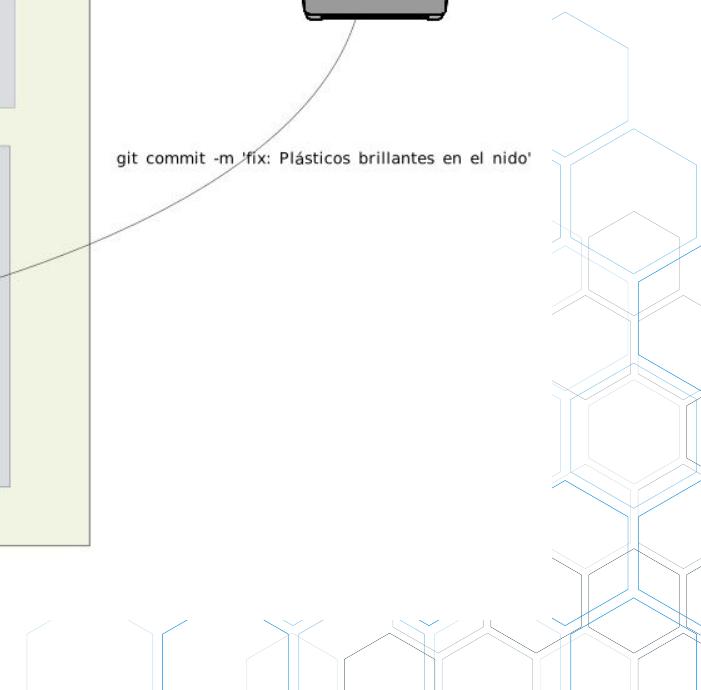
Al otro día...
git pull....



git commit -m 'fix: Viajar a la Antártida'



git commit -m 'fix: Plásticos brillantes en el nido'



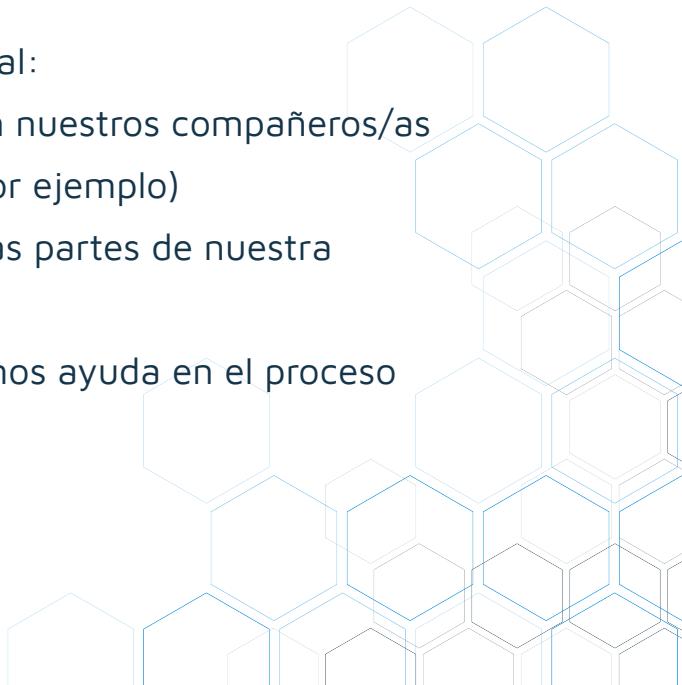


Facilitan el testeo y el debugging de la aplicación.



Módulos - Resumen

- Los módulos son una colección de distintos componentes lógicos en nuestra aplicación.
- Nos permiten compartmentalizar la aplicación lo cual:
 1. Nos ayuda a trabajar de manera ordenada con nuestros compañeros/as y con los sistemas de control de versión (git por ejemplo)
 2. Nos permite reutilizar componentes en distintas partes de nuestra aplicación
 3. Nos ayuda a escribir mejores tests y también nos ayuda en el proceso de debugging.



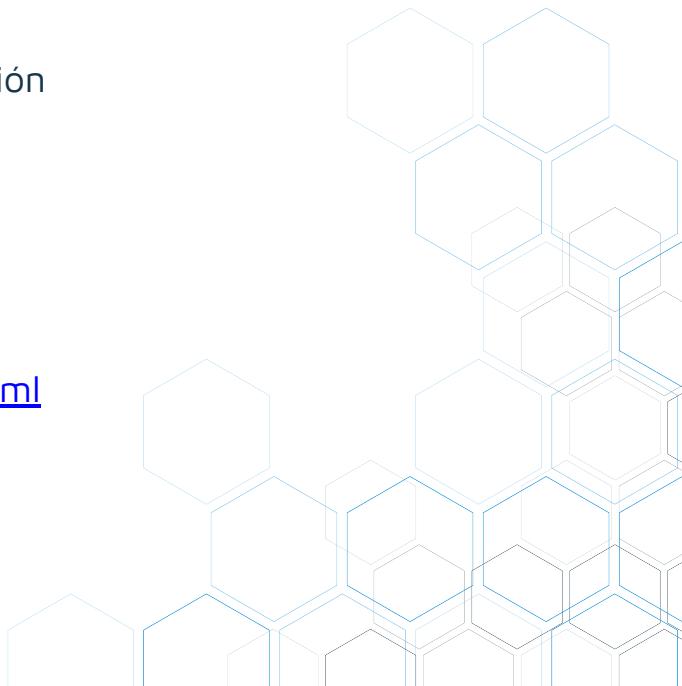
Módulos en Shiny

Shiny modules

De qué nos sirve modularizar nuestro código de Shiny?

- Encapsular la lógica de un componente en la aplicación
- Reutilizar el componente
- Facilitar el trabajo sobre la aplicación
- Facilitar el testeo y debugging

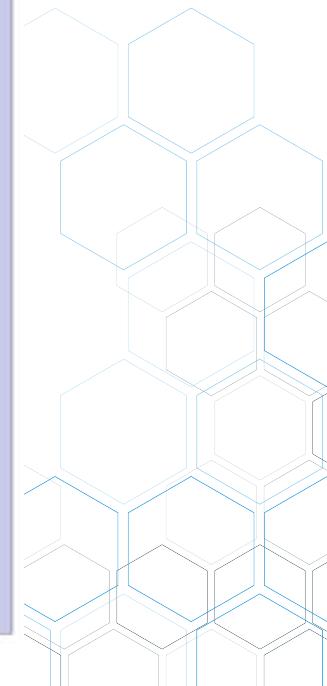
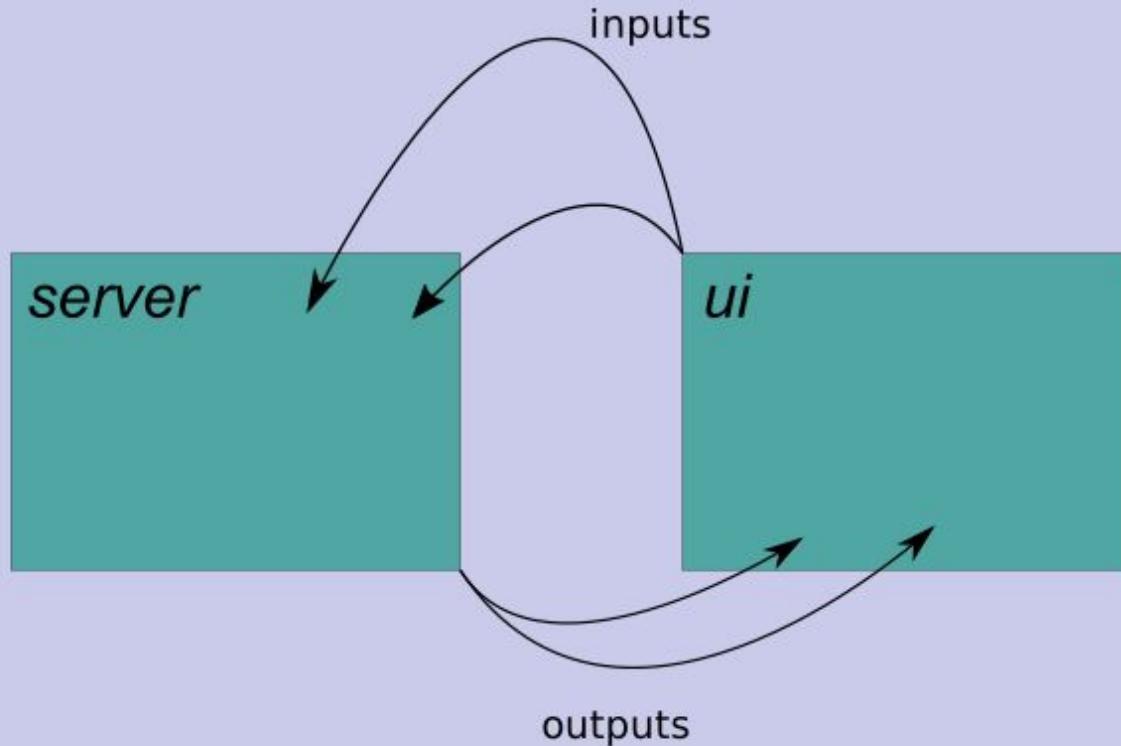
Ver más en: <https://shiny.rstudio.com/articles/modules.html>



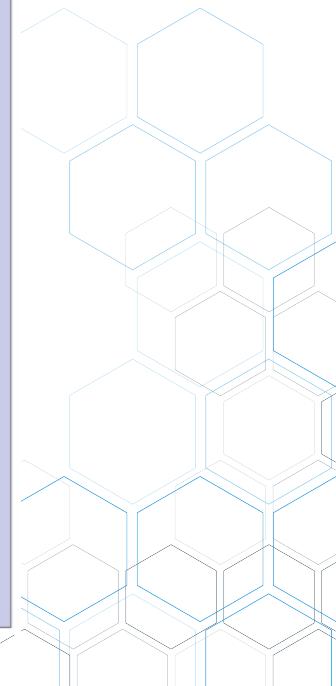
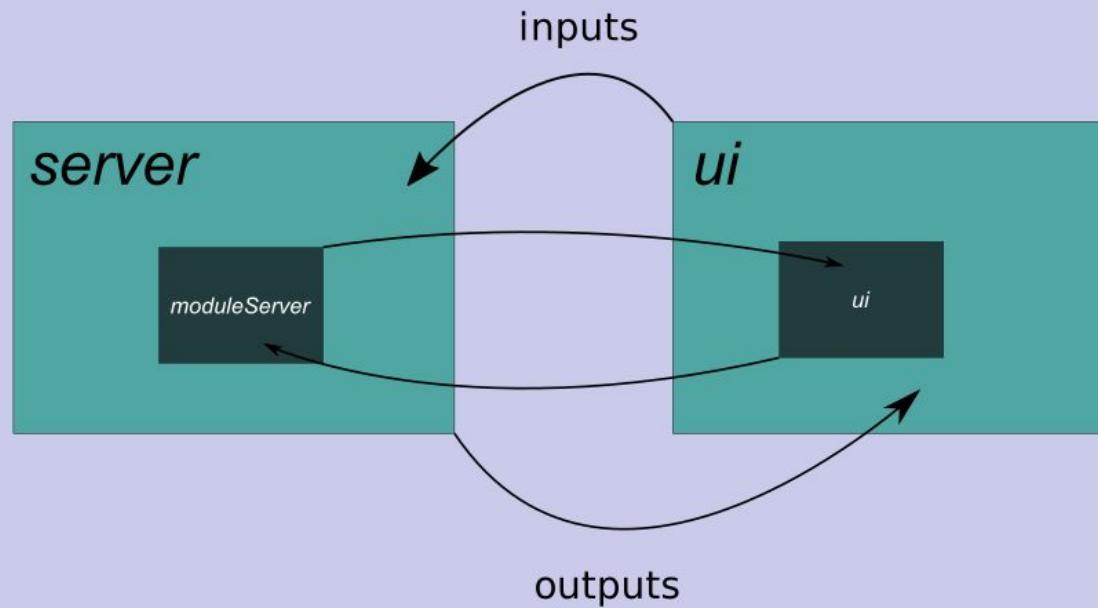
¿Cómo se usan los módulos de Shiny?



GLOBAL



GLOBAL



Conceptos clave

- [Environments](#) en R!
 - Una aplicación de shiny “vive” bajo el environment global, pero a su vez cada función **server** y **ui** constituyen sus propios environments que viven bajo el mismo “techo” (globalEnv)
 - Los módulos como cualquier otra función poseen su propio environment, de esta forma, se “aíslan” del exterior.



Pero... surge un problema



Google Trend Index

Trend index

Travel

Date range

2007-01-01 to 2017-07-31

Overlay smooth trend line

Trend index

1.1

1.0

0.9

0.8

0.7

1

2

Description app.R

Shiny comes with a variety of built in input widgets. With minimal syntax it is possible to include widgets like the ones shown on the left in your apps:

```
# Select type of trend to plot
selectInput(inputId = "type", label = strong("Trend index"),
  choices = unique(trend_data$type),
  selected = "Travel")

# Select date range to be plotted
dateRangeInput("date", strong("Date range"),
  start = "2007-01-01", end = "2017-07-31",
  min = "2007-01-01", max = "2017-07-31")
```

Displaying outputs is equally hassle-free:

```
mainPanel(
  plotOutput(outputId = "lineplot", height = "300px"),
  textOutput(outputId = "desc"),
  tags$a(href = "https://www.google.com/finance/domestic_trends",
    "Source: Google Domestic Trends", target = "_blank")
)
```

Build your plots or tables as you normally would in R, and make them reactive with a call to the appropriate render function:

```
output$lineplot <- renderPlot({
  plot(x = selected_trends$date, y = selected_trends$close, type = "l")
```

trigger Network {} Style Editor Console Performance Memory Storage Accessibility Application

Rules Layout Computed Changes Fonts Animation

Filter Styles

Pseudo-elements

::before, ::after {
 -webkit-box-sizing: border-box;
 -moz-box-sizing: border-box;
 box-sizing: border-box;
}

::after, ::before {
 -webkit-box-sizing: border-box;
 -moz-box-sizing: border-box;
 box-sizing: border-box;
}

Situación: Necesitamos agregar dos inputs que tienen una lógica muy similar.

Específicamente: dos selectizeInputs que listen letras, para categoría y para tipo y mostrar su output.

modulos_ejemplo_1_incorrecto.R

Ejemplo 1 - Sin usar módulos

Seleccione una Categoría

Seleccione un tipo

Categoría seleccionada: A
Tipo seleccionado: F

```
library(shiny)

ui <- function() {
  fluidPage(
    title = "Ejemplo 1",
    fluidRow(
      h2("Ejemplo 1 - Sin usar módulos", style = "margin-left: 50px"),
      column(
        width = 6,
        class = "col-sm-offset-4",
        selectizeInput(
          inputId = "categoria",
          label = "Seleccione una Categoría",
          choices = LETTERS[1:5]
        ),
        selectizeInput(
          inputId = "tipo",
          label = "Seleccione un tipo",
          choices = LETTERS[6:10]
        ),
        div(
          class = "well",
          textOutput("letra_seleccionada_categoria"),
          textOutput("letra_seleccionada_tipo")
        )
      )
    )
}

server <- function(input, output, session) {
  output$letra_seleccionada_categoria <- renderText({
    sprintf("Categoría seleccionada: %s", input$categoria)
  })
  output$letra_seleccionada_tipo <- renderText({
    sprintf("Tipo seleccionado: %s", input$tipo)
  })
}

shinyApp(ui, server)
```

1. Identificamos que estamos repitiendo código
2. El comportamiento del server es “lógicamente” el mismo

¿Porqué modularizar? Por todo lo que ya vimos antes, pero además:

- Al replicar podemos cometer errores
- Alarga el código innecesariamente
- Es aburrido leer código repetitivo, se ocultan los errores para el revisor.
- Nadie quiere leer un script de 5000000 líneas por lo que impacta en la reproducibilidad, mantenimiento, paciencia, etc.
- Es aburrido leer código repetitivo, se ocultan los errores para el revisor.
- Alarga el código innecesariamente
- Al replicar podemos cometer errores

The screenshot shows a Shiny application interface. On the left, there's a sidebar with tabs for 'Trigger', 'Network', 'Style Editor', 'Console', 'Performance', 'Memory', 'Storage', 'Accessibility', and 'Application'. The main area has a title 'Google Trend Index' and contains two sections: 'Trend index' and 'Date range'. In the 'Trend index' section, there's a dropdown menu labeled 'Travel' with a red number '1' above it. Below it is a line plot titled 'Trend index' with a red number '2' above it. The plot shows a jagged line representing the trend index over time, with a smooth blue line overlaid. The 'Date range' section includes date pickers for '2007-01-01' and '2017-07-31', and a checkbox for 'Overlay smooth trend line'. At the bottom of the page, there's a code editor showing R code for the 'app.R' file. The code defines a 'selectInput' for 'type' with 'Travel' as the selected value, and a 'dateRangeInput' for the date range. It also defines a 'mainPanel' containing a 'plotOutput' for 'lineplot' with a height of 300px, a 'textOutput' for 'desc', and a link to Google Trends. The code editor highlights the 'type' input field with a blue box. On the right side of the page, there's a 'Description' section with text about built-in input widgets and their minimal syntax, followed by a code snippet for the 'app.R' file. A 'Rules' panel on the right shows CSS rules for pseudo-elements like ::before and ::after.

```
# Select type of trend to plot
selectInput(inputId = "type", label = strong("Trend index"),
            choices = unique(trend_data$type),
            selected = "Travel")

# Select date range to be plotted
dateRangeInput("date", strong("Date range"),
               start = "2007-01-01", end = "2017-07-31",
               min = "2007-01-01", max = "2017-07-31")

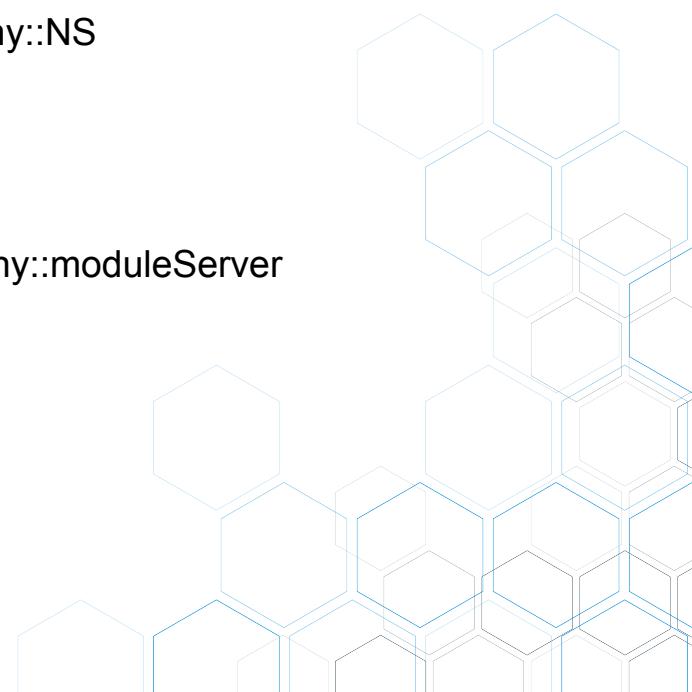
mainPanel(
  plotOutput(outputId = "lineplot", height = "300px"),
  textOutput(outputId = "desc"),
  tags$a(href = "https://www.google.com/finance/domestic_trends",
         "Source: Google Domestic Trends", target = "_blank")
)
```

modulos_ejemplo_1_correcto.R

```
3 * letras_ui <- function(id, desde, hasta, label) {  
4   ns <- NS(id) ← shiny::NS  
5   tagList(  
6     selectizeInput(  
7       inputId = ns("letras"),  
8       label = label,  
9       choices = LETTERS[desde:hasta]  
10    ),  
11    div(  
12      class = "well",  
13      textOutput(ns("letra_seleccionada"))  
14    )  
15  )  
16} ← shiny::moduleServer  
17  
18 * letras_server <- function(id, label) {  
19   moduleServer(← id, function(input, output, session) {  
20     output$letra_seleccionada <- renderText({  
21       sprintf("%s: %s", label, input$letras)  
22     })  
23   })  
24 } ← shiny::moduleServer  
25
```

shiny::NS

shiny::moduleServer



Seleccione una Categoría

D

Categoría seleccionada: D

Seleccione un Tipo

H

Tipo seleccionado: H

Inspector Debugger Network {} Style Editor Console Performance Memory Storage Accessibility Application

Search HTML

```
<div>
  <select id="categoria-letras" class="form-control selectized shiny-bound-input" tabindex="-1" style="display: none;"></select> event
  <div class="selectize-control form-control single plugin-selectize-plugin-ally"></div>
  <script type="application/json" data-for="categoria-letras">{"plugins": ["selectize-plugin-ally"]}</script>
</div>
</div>
<div class="well"></div>
<div class="form-group shiny-input-container">
  <label id="tipo-letras-label" class="control-label" for="tipo-letras-selectized">Seleccione un Tipo</label>
  <div>
    <select id="tipo-letras" class="form-control selectized shiny-bound-input" tabindex="-1" style="display: none;"></select> event
    <div class="selectize-control form-control single plugin-selectize-plugin-ally"></div>
    <script type="application/json" data-for="tipo-letras" data-for-type="letras">{"enLínea": "true", "selectize.plugin.ally.all": true}</script>
  </div>
</div>
```

```
html > body > div.container-fluid > div.row > div.col-sm-6.col-sm-offset-4 > div.form-group.shiny-input-container > div > div.selectize-control.form-control.single > div.selectize-input.items.has-options.fu...
```

Rules Layout Compute

Filter Styles

Pseudo-elements

```
.selectize-control.single .sel
  content: '';
  display: block;
  position: absolute;
  top: 50%;
  right: 17px;
  margin-top: -3px;
  width: 0;
  height: 0;
  border-style: solid;
  border-width: 5px 5px 0 5px;
```

```

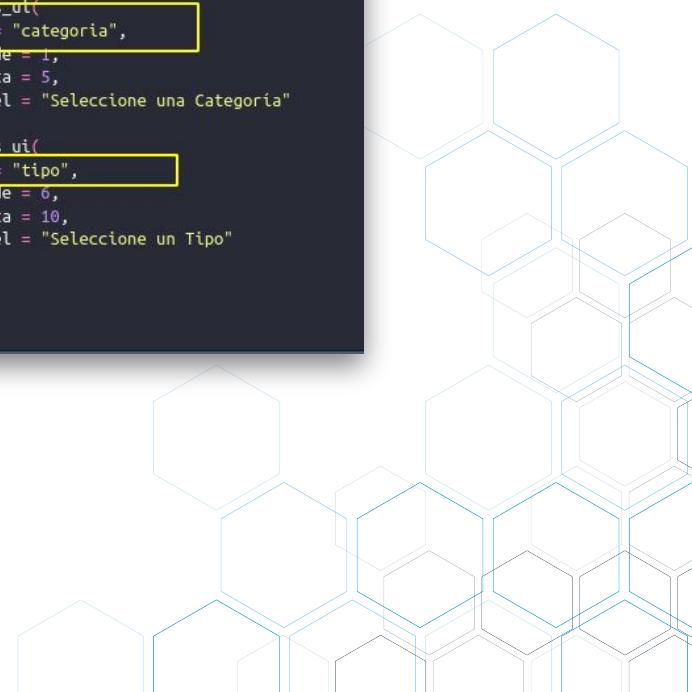
3 * letras_ui <- function(id, desde, hasta, label) {
4   ns <- NS(id)
5   tagList(
6     selectizeInput(
7       inputId = ns("letras"),
8       label = label,
9       choices = LETTERS[desde:hasta]
10    ),
11    div(
12      class = "well",
13      textOutput(ns("letra_seleccionada"))
14    )
15  )
16 ^ }
17
18 * letras_server <- function(id, label) {
19   moduleServer(id, function(input, output, session) {
20     output$letra_seleccionada <- renderText({
21       sprintf("%s: %s", label, input$letras)
22     })
23   })
24 ^ }
25

```

```

26 * ui <- function() {
27   fluidPage(
28     title = "Ejemplo 1",
29     fluidRow(
30       h2("Ejemplo 1", style = "margin-left: 50px"),
31       column(
32         width = 6,
33         class = "col-sm-offset-4",
34         letras_ui(
35           id = "categoria",
36           desde = 1,
37           hasta = 5,
38           label = "Seleccione una Categoria"
39         ),
40         letras ui(
41           id = "tipo",
42           desde = 6,
43           hasta = 10,
44           label = "Seleccione un Tipo"
45         )
46       )
47     )
48   )

```



Seleccione una Categoría

D

Categoría seleccionada: D

Seleccione un Tipo

H

Tipo seleccionado: H

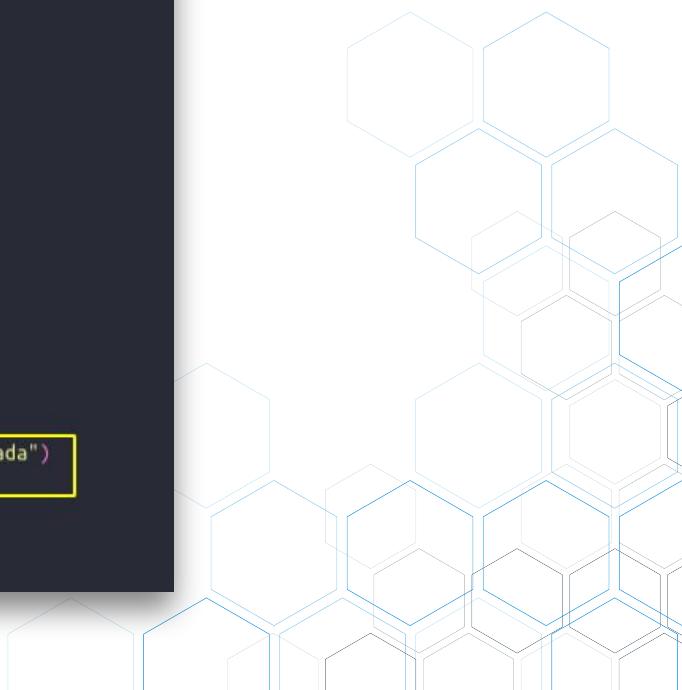
The screenshot shows the browser's developer tools with the 'Inspector' tab selected. At the top, there are tabs for Inspector, Debugger, Network, Style Editor, Console, Performance, Memory, Storage, Accessibility, and Application. Below the tabs is a search bar labeled 'Search HTML'. The main area displays the DOM structure of a web page. Two blue arrows point from the 'Inspector' tab towards the first dropdown menu in the screenshot above. The DOM structure shows two

elements containing elements and

```
ui <- function() {
  fluidPage(
    title = "Ejemplo 1",
    fluidRow(
      h2("Ejemplo 1", style = "margin-left: 50px"),
      column(
        width = 6,
        class = "col-sm-offset-4",
        letras_ui(
          id = "categoria",
          desde = 1,
          hasta = 5,
          label = "Seleccione una Categoría"
        ),
        letras_ui(
          id = "tipo",
          desde = 6,
          hasta = 10,
          label = "Seleccione un Tipo"
        )
      )
    )
}

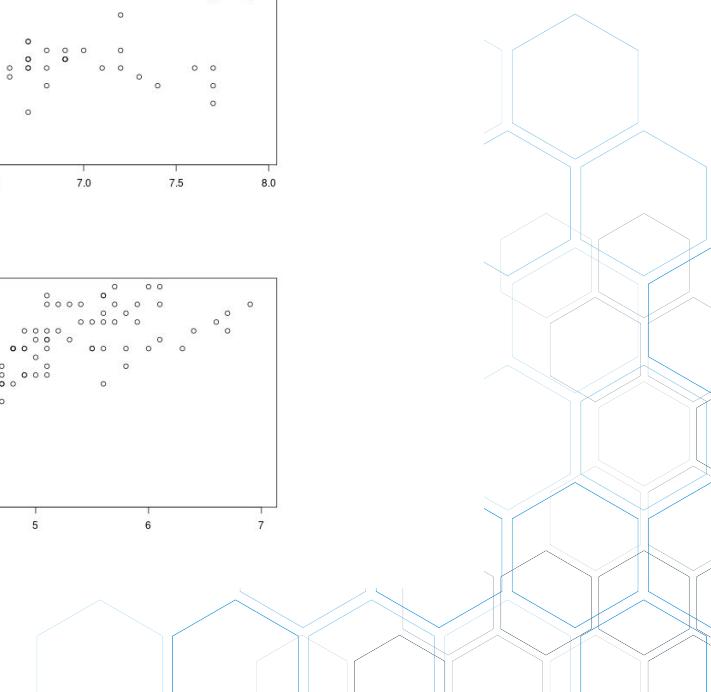
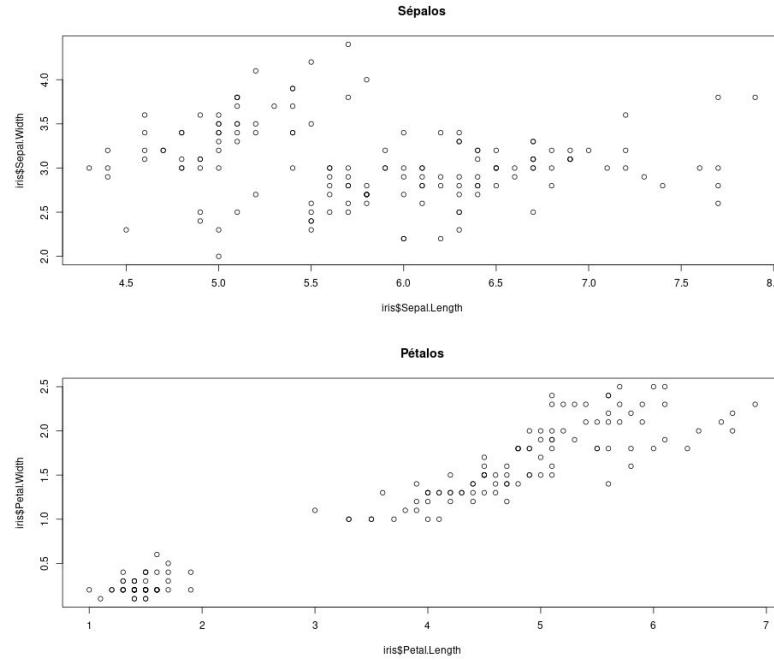
server <- function(input, output, session) {
  letras_server(id = "categoria", label = "Categoria seleccionada")
  letras_server(id = "tipo", label = "Tipo seleccionado")
}

shinyApp(ui, server)
```



Ejercicio - modulos_ejercicio_1.R

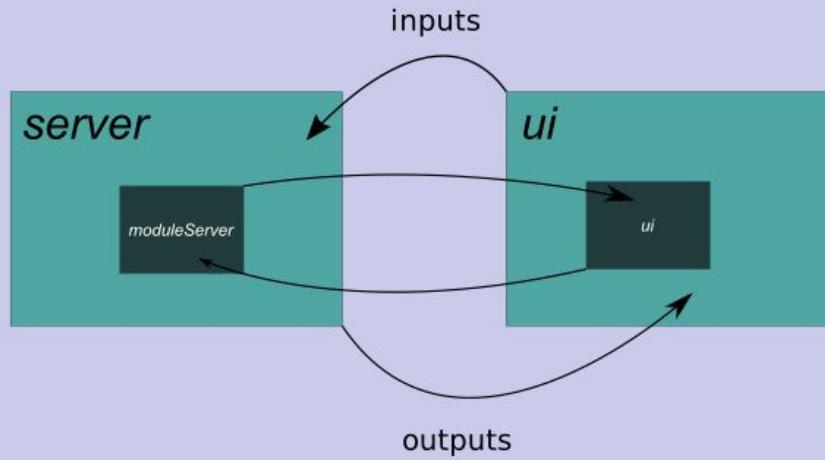
Ejercicio - Implementar módulos de Shiny



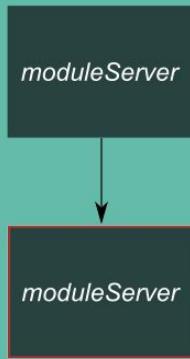
Comunicación entre módulos de Shiny



GLOBAL



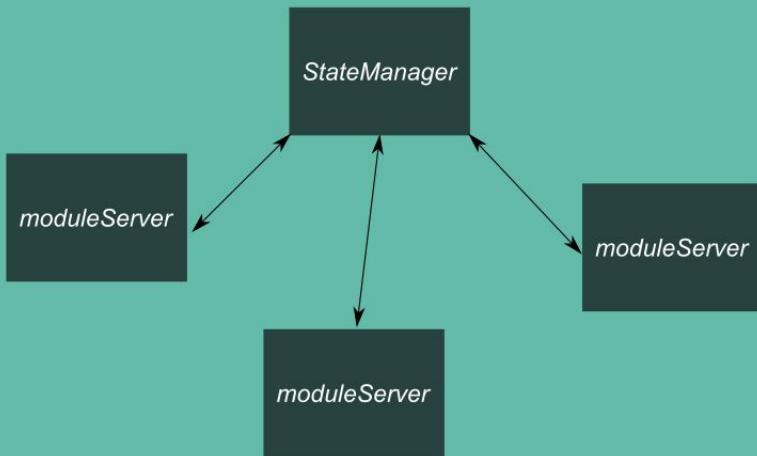
server



Los módulos se estructuran de manera jerárquica (de arriba hacia abajo)

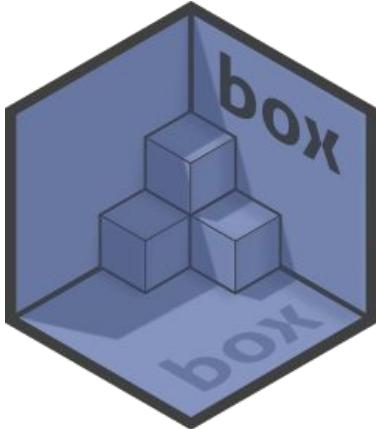
- + Es una estructura confiable (evita el reactive spaghetti)
- A medida que la app se vuelve compleja es cada vez más complicado manejar el estado de la app. (similar al *prop drilling* en otros frameworks)

server



Los módulos ya no requieren una estructura

- + Es una estructura/patrón más flexible, más rápidamente escalable
- Conlleva ciertos riesgos lo que implica un manejo más cuidadoso de la reactividad.
- Implica la escritura del módulo que maneja el estado (p. ej. R6)



Modularizando código con box

<https://klmr.me/box/>



¿Por qué usar box?

- Todas las razones anteriores acerca de la modularización
- Cachea los paquetes
- Da la oportunidad de solo cargar lo que se usa
- Aumenta notablemente el desempeño del tiempo de carga de una app al inicializar
- Fácil de aprender
- IMPORTANTE: Al hacer un cache de los paquetes cargados, requiere reinicializar la sesión ante cada cambio durante el desarrollo.

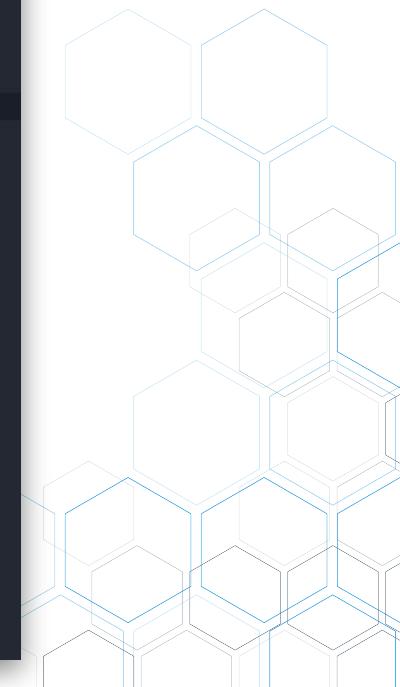


ejemplos/box

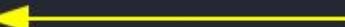
```
> modules > @< textInput.R > ui
  box::use(
    shiny[NS, div, h4,textInput, tagList, textOutput, renderText, moduleServer,
          | req|,
          stringr[str_squish],
    ]
  )

  ui <- function(id, label) {
    ns <- NS[id]
    tagList(
      textInput(inputId = ns("texto_in"), label = label),
      div(
        class = "well",
        textOutput(outputId = ns("texto_out"))
      )
    )
  }

  server <- function(id) {
    moduleServer(id, function(input, output, session) {
      output$texto_out <- renderText({
        req(input$texto_in)
        str_squish(input$texto_in)
      })
    })
  }
}
```



```
global.R x ui.R app.R text_input.R 3 server.R 2
app > global.R
1 library(shiny)
2
3 box::use(
4   texto = modules/text_input,
5 )
```



Ejercicio - ejercicios/box_ejercicio

- Modificar el comportamiento del procesado del texto
 - Formatear a título usando `str_to_title()` de `stringr`
 - Remover números usando `str_remove_all()` de `stringr`

```
str_remove_all(string = "a321 321 3dda", pattern = "[0-9]")
```

Referencias

- Módulos en Shiny <https://shiny.rstudio.com/articles/modules.html>
- Environments <https://adv-r.hadley.nz/environments.html>
- Box <https://klmr.me/box/>

¿Preguntas?

¡Gracias por su tiempo!