

# Dominando o Shiny: como construir seu primeiro dashboard interativo



Paula Maçaira (DEI, PUC-Rio)

SBPO 2022 - Juiz de Fora

## Quem sou eu

- ▶ Bacharelado em Estatística (ENCE, 2013)
- ▶ Mestrado em Eng. Elétrica (PUC-Rio, 2015)
- ▶ Doutorado em Eng. de Produção (PUC-Rio, 2018)
- ▶ Pós-doutorado em Eng. de Produção (PUC-Rio, 2019)
- ▶ Professora Adjunta desde 2019 (DEI, PUC-Rio)

We're data-driven!



## Me encontre em

- ▶ [github.com/paulamacaira](https://github.com/paulamacaira)
- ▶ [sites.google.com/view/paulamacaira](https://sites.google.com/view/paulamacaira)
- ▶ [paulamacaira@puc-rio.br](mailto:paulamacaira@puc-rio.br)



## Premissas

- ▶ Presumo que você conheça minimamente a **linguagem R** e o meio ambiente **RStudio**
- ▶ Preciso que você tenha instalado o software R e a IDE RStudio em seu computador

Falaremos sobre...

## Dia 1:

1. estrutura básica de um Shiny App
2. inputs e outputs comuns
3. o básico da programação reativa

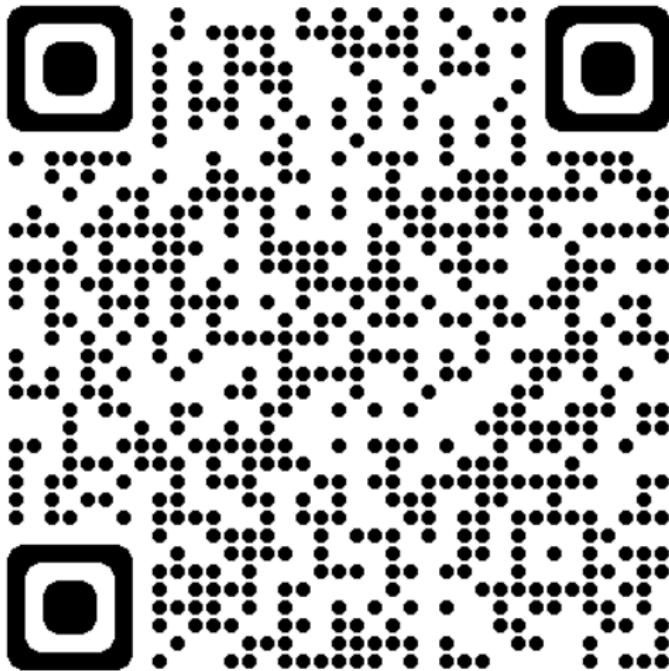
## Dia 2:

4. um pouco sobre layouts
5. pacotes plotly, dygraphs e leaflet
6. como publicar seu Shiny App



Let's Go

Para acessar todo material



Vamos começar!

## **Dia 1**

# O que é um Shiny App

- ▶ Shiny é um pacote R que permite criar facilmente aplicativos web interativos
- ▶ Esse pacote permite que você pegue seu trabalho em R e o exponha por meio de um navegador da web para que qualquer pessoa possa usá-lo de maneira simples e de graça
- ▶ No passado, criar aplicativos web era difícil para a maioria dos usuários de R porque precisava de um profundo conhecimento de tecnologias da web como HTML, CSS e JavaScript

## Vantagens de um Shiny App

- ▶ O Shiny tornou significativamente mais fácil para o programador de R criar aplicativos da web ao:
  - (i) fornecer um conjunto selecionado de funções de interface do usuário (UI para abreviar) que geram o HTML, CSS e JavaScript necessários para tarefas comuns, ou seja, você não precisa conhecer os detalhes de HTML/CSS/JavaScript se não quiser;
  - (ii) apresentar a programação reativa que sempre que uma entrada muda atualiza todas as saídas relacionadas.

## Para o que um Shiny App é utilizado

- ▶ O Shiny é utilizado para uma variedade de finalidades, por exemplo:
  - ▶ para criar painéis que mostrem importantes indicadores de desempenho de alto nível, enquanto facilitam o detalhamento das métricas que precisam de mais investigação;
  - ▶ substituir centenas de páginas de PDFs por aplicativos interativos que permitem ao usuário saltar para a fatia exata dos resultados que lhe interessam;
  - ▶ comunicar modelos complexos a um público não técnico com visualizações informativas e análise de sensibilidade interativa;
  - ▶ fornecer análise de dados de autoatendimento para fluxos de trabalho comuns, substituindo solicitações de e-mail por um aplicativo Shiny que permite que as pessoas carreguem seus próprios dados e realizem análises padrão

## Mais possibilidades de um Shiny App

- ▶ Com o Shiny você consegue criar demonstrações interativas para ensinar estatística e conceitos de ciência de dados que permitem que os usuários ajustem as entradas e observem os efeitos posteriores dessas alterações em uma análise

Resumindo, o Shiny lhe dá a habilidade de passar alguns de seus superpoderes R para qualquer um que possa usar a web e não tenham qualquer habilidade de programação.

## Exibição de alguns Shiny Apps

<https://shiny.rstudio.com/gallery/>

# First things first

- ▶ Se você ainda não instalou o Shiny, faça agora

```
install.packages("shiny")
```

- ▶ Em seguida, carregue em sua sessão R atual

```
library(shiny)
```

# Seu primeiro Shiny App

- ▶ Existem várias maneiras de criar um aplicativo Shiny
- ▶ O mais simples é criar um novo diretório para seu aplicativo e colocar um único arquivo chamado **app.R** nele
- ▶ Este arquivo **app.R** será usado para informar ao Shiny como seu aplicativo deve ser e como ele deve se comportar

Experimente criando um novo diretório e adicionando um arquivo **app.R** parecido com o seguinte

# Seu primeiro Shiny App

```
library(shiny)

ui <- fluidPage(
  "Hello, world!"
)

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

shinyApp(ui, server)
```

# Seu primeiro Shiny App

Embora trivial, este é um aplicativo Shiny completo!

- ▶ Observando atentamente o código anterior, nosso **app.R** faz quatro coisas:
  1. Ele chama **library(shiny)** para carregar o pacote shiny
  2. Ele define a interface do usuário, a página da Web HTML com a qual os humanos interagem. Neste caso, é uma página contendo as palavras “Hello, world!”
  3. Ele especifica o comportamento do nosso aplicativo definindo uma função de servidor. No momento, está vazio, então nosso aplicativo não faz nada, mas voltaremos para revisitar isso em breve
  4. Ele executa **shinyApp(ui, server)** para construir e iniciar um aplicativo Shiny a partir da interface do usuário e do servidor

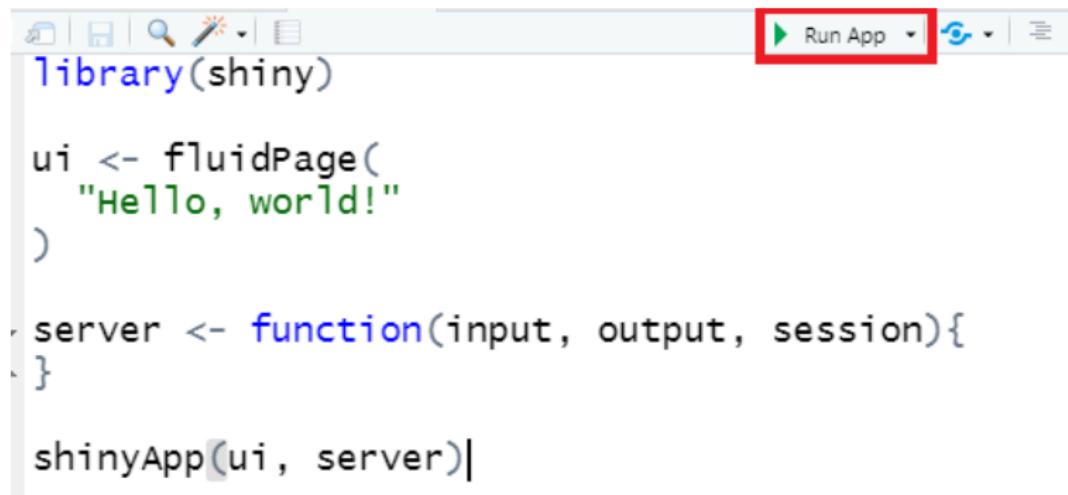
## Dica do RStudio

Existem três maneiras convenientes de criar um novo aplicativo no RStudio:

- ▶ Crie um novo diretório e um arquivo **app.R** contendo um aplicativo básico em uma etapa clicando em **File | New Project** e, em seguida, selecionando **New Directory** e **Shiny Web Application**
- ▶ Se você já criou o arquivo **app.R**, pode adicionar rapidamente a estrutura básica do aplicativo digitando “shinyapp” e pressionando **Shift+Tab**

## Como rodar o app

Para rodar o app, clique em **RunApp**



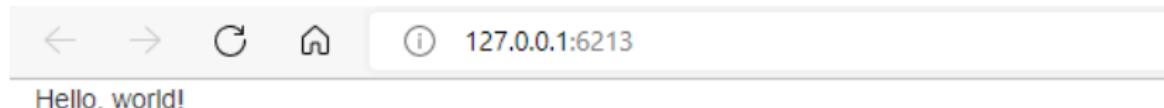
The screenshot shows the RStudio interface with a Shiny application code editor. The title bar says "library(shiny)". The code editor contains the following R code:

```
ui <- fluidPage(  
  "Hello, world!"  
)  
  
server <- function(input, output, session){  
}  
  
shinyApp(ui, server)
```

The "Run App" button in the toolbar is highlighted with a red box.

## Verificando se você fez certo

Rode o **app.R** e verifique se você vê o mesmo aplicativo da Figura abaixo



Parabéns! Você fez seu primeiro aplicativo Shiny.

# Workflow de um Shiny App

O workflow básico do desenvolvimento de aplicativos Shiny é:

- ▶ escrever algum código
- ▶ iniciar o aplicativo
- ▶ brincar com o aplicativo
- ▶ escrever mais código e repetir

Se você estiver usando o RStudio, nem precisa parar e reiniciar o aplicativo para ver suas alterações - você pode pressionar o botão **Reload App**.

## Adicionando controles UI

- ▶ Vamos adicionar algumas entradas e saídas à nossa UI para que nosso Shiny App não seja tão mínimo
- ▶ Inicialmente, vamos fazer um aplicativo muito simples que mostra todos os conjuntos de dados incluídos no pacote *datasets*

Substitua sua UI por este código:

```
ui <- fluidPage(  
  selectInput("dataset", label = "Dataset",  
             choices = ls("package:datasets")),  
  verbatimTextOutput("summary"),  
  tableOutput("table")  
)
```

## Destrinchando o que fizemos

Este exemplo usa quatro novas funções:

- ▶ **fluidPage()** é uma função de layout que configura a estrutura visual básica da página
- ▶ **selectInput()** é um controle de entrada que permite que o usuário interaja com o aplicativo fornecendo um valor. Nesse caso, é uma caixa de seleção com o rótulo “Dataset” e permite que você escolha um dos conjuntos de dados incluídos no pacote *datasets*
- ▶ **verbatimTextOutput()** e **tableOutput()** são controles de saída que informam ao Shiny onde colocar a saída
- ▶ **verbatimTextOutput()** exibe código e **tableOutput()** exibe tabelas

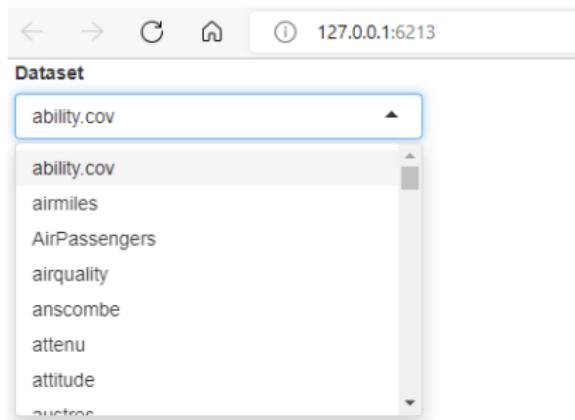
## Funções de layout, entradas e saídas

Em suma, funções de layout, entradas e saídas têm usos diferentes, mas são fundamentalmente as mesmas nos bastidores: são apenas maneiras sofisticadas de gerar HTML e, se você chamar qualquer um deles fora de um aplicativo Shiny, verá um HTML impresso no console

Não tenha medo de bisbilhotar para ver como esses vários layouts e controles funcionam nos bastidores.

## Verificando o Shiny App

- ▶ Vá em frente e execute o **app.R** novamente
- ▶ Agora você verá uma página contendo uma caixa de seleção



- ▶ Vemos apenas a entrada, não as duas saídas, porque ainda não dissemos ao Shiny como a entrada e as saídas estão relacionadas (o **server** continua vazio)

## Adicionando tipos de saída

- ▶ Agora, daremos vida às saídas definindo-as na função do servidor (**server**)
- ▶ O Shiny usa programação reativa para tornar os aplicativos interativos
- ▶ Por enquanto, é suficiente saber que programação reativa envolve dizer ao Shiny como realizar um cálculo
- ▶ Ou seja, diremos ao Shiny como preencher as saídas **verbatimTextOutput()** e **tableOutput()** no aplicativo de exemplo, fornecendo as “receitas” para essas saídas

## Adicionando tipos de saída

Substitua sua função de servidor vazia por esta:

```
server <- function(input, output, session) {  
  output$summary <- renderPrint({  
    dataset <- get(input$dataset, "package:datasets")  
    summary(dataset)  
  })  
  
  output$table <- renderTable({  
    dataset <- get(input$dataset, "package:datasets")  
    dataset  
  })  
}
```

## Destrinchando o que fizemos

- ▶ O lado esquerdo do operador de atribuição (<-), **output\$ID**, indica que você está fornecendo a receita para a saída Shiny com esse ID
- ▶ O lado direito da atribuição usa uma função de renderização específica para agrupar algum código que você fornece
- ▶ Cada função **render{Type}** é projetada para produzir um tipo específico de saída (por exemplo, texto, tabelas e gráficos) e geralmente é combinada com uma função **{type}Output**
- ▶ Por exemplo, neste aplicativo, **renderPrint()** é emparelhado com **verbatimTextOutput()** para exibir um resumo estatístico com texto de largura fixa (verbatim), e **renderTable()** é emparelhado com **tableOutput()** para mostrar os dados de entrada em uma tabela

# Verificando o Shiny App

- ▶ Vá em frente e execute o **app.R** novamente e brinque, observando o que acontece com a saída quando você altera uma entrada
- ▶ A Figura abaixo mostra o que você deve ver ao abrir o aplicativo

Dataset

ability.csv

	Length	Class	Mode
cov	36	-none-	numeric
center	6	-none-	numeric
n.obs	1	-none-	numeric

cov.general	cov.picture	cov.blocks	cov.maze	cov.reading	cov.vocab	center	n.obs
24.64	5.99	33.52	6.02	20.75	29.70	0.00	112.00
5.99	6.70	18.14	1.78	4.94	7.20	0.00	112.00
33.52	18.14	149.83	19.42	31.43	50.75	0.00	112.00
6.02	1.78	19.42	12.71	4.76	9.07	0.00	112.00
20.75	4.94	31.43	4.76	52.60	66.76	0.00	112.00
29.70	7.20	50.75	9.07	66.76	135.29	0.00	112.00

## Reducindo a duplicação com expressões reativas

- ▶ Mesmo neste exemplo simples, temos algum código duplicado: a linha a seguir está presente em ambas as saídas.

```
dataset <- get(input$dataset, "package:datasets")
```

- ▶ Em todo tipo de programação, é uma prática ruim ter código duplicado; pode ser um desperdício computacional e, mais importante, aumenta a dificuldade de manter ou depurar o código
- ▶ No script R tradicional, usamos duas técnicas para lidar com código duplicado: capturamos o valor usando uma variável ou capturamos a computação com uma função. Infelizmente, nenhuma dessas abordagens funciona aqui e precisamos de um novo mecanismo: expressões reativas

## Expressões reativas

- ▶ Você cria uma expressão reativa envolvendo um bloco de código em **reactive({...})** e atribuindo-o a uma variável, e usa uma expressão reativa chamando-a como uma função
- ▶ No entanto, enquanto parece que você está chamando uma função, uma expressão reativa tem uma diferença importante: ela só é executada na primeira vez que é chamada e depois armazena seu resultado até que precise ser atualizado

## Usando expressões reativas

- Podemos atualizar nosso **server()** para usar expressões reativas<sup>1</sup>, conforme mostrado abaixo

```
server <- function(input, output, session) {  
  dataset <- reactive({  
    get(input$dataset, "package:datasets")  
  })  
  
  output$summary <- renderPrint({  
    summary(dataset())  
  })  
  
  output$table <- renderTable({  
    dataset()  
  })  
}
```

<sup>1</sup>O aplicativo se comporta de forma idêntica, mas funciona com um pouco mais de eficiência porque só precisa recuperar o conjunto de dados uma vez, não duas

## Resumo desta primeira parte

- ▶ Nesta primeira parte do curso, você criou um aplicativo simples - não muito interessante ou útil, mas viu como é fácil construir um aplicativo da web usando seu conhecimento R existente
- ▶ Nos próximos momentos, você aprenderá mais sobre interfaces de usuário e programação reativa, os dois blocos básicos de construção do Shiny
- ▶ Agora é um ótimo momento para pegar uma cópia da folha de dicas Shiny: <https://www.rstudio.com/resources/cheatsheets/>
- ▶ Este é um ótimo recurso para ajudar a refrescar sua memória dos principais componentes de um aplicativo Shiny

## UI básico

- ▶ Como você acabou de ver, o Shiny incentiva a separação do código que gera sua interface de usuário (o front-end) do código que orienta o comportamento do seu aplicativo (o back-end)
- ▶ A partir de agora, focaremos no front-end e faremos um tour rápido pelas entradas e saídas HTML fornecidas pelo Shiny
- ▶ Isso lhe dá a capacidade de capturar muitos tipos de dados e exibir muitos tipos de saída R
- ▶ Nesse curso, vamos nos ater principalmente às entradas e saídas incorporadas ao próprio Shiny<sup>2</sup>.

---

<sup>2</sup>Você pode encontrar uma lista abrangente e mantida ativamente de outros pacotes em <https://github.com/nanxstats/awesome-shiny-extensions>, mantido por Nan Xiao

# Inputs

- ▶ Como vimos anteriormente, você usa funções como **sliderInput()**, **selectInput()**, **textInput()** e **numericInput()** para inserir controles de entrada em sua especificação de UI
- ▶ Agora vamos discutir a estrutura comum subjacente a todas as funções de entrada e dar uma rápida visão geral das entradas incorporadas ao Shiny

## Estrutura comum

- ▶ Todas as funções de entrada têm o mesmo primeiro argumento:  
**inputId**
  - ▶ este é o identificador usado para conectar o front-end com o back-end: se sua UI tiver uma entrada com ID “**name**”, a função do servidor irá acessá-la com **input\$name**
- ▶ O inputId tem duas restrições:
  - ▶ Deve ser uma string simples que contenha apenas letras, números e underlines (não são permitidos espaços, traços, pontos ou outros caracteres especiais!)
  - ▶ Deve ser único. Se não for único, você não terá como se referir a esse controle em sua função de servidor!

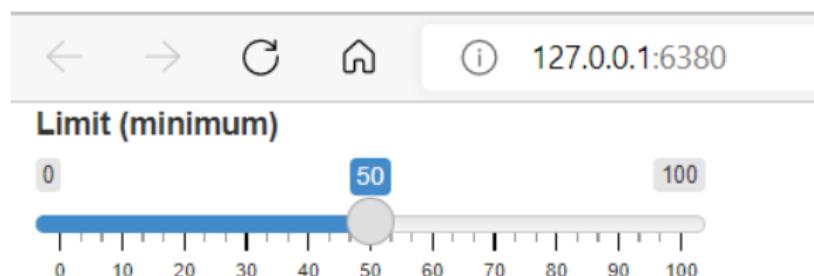
## Estrutura comum

- ▶ A maioria das funções de entrada tem um segundo parâmetro chamado label - usado para criar um rótulo legível para o controle
  - ▶ O Shiny não impõe restrições a essa string, mas você precisará pensar cuidadosamente sobre isso para garantir que seu aplicativo seja utilizável por humanos!
- ▶ O terceiro parâmetro normalmente é o valor, que, sempre que possível, permite definir o valor padrão
- ▶ Os demais parâmetros são exclusivos de cada controle

## Estrutura comum - exemplo

Ao criar uma entrada, recomendo fornecer os argumentos **inputId** e **label** por posição e todos os outros argumentos pelo seu nome, por exemplo:

```
sliderInput("min", "Limit (minimum)",  
           value = 50, min = 0, max = 100)
```



# Tipos de inputs

- A seguir vamos descrever - de forma rápida - as entradas incorporadas em Shiny, agrupadas de acordo com o tipo de controle que eles criam

http://127.0.0.1:3771 | [Open in Browser](#) | [Publish](#)

## Basic widgets

<b>Buttons</b> <input type="button" value="Action"/>  <input type="button" value="Submit"/>	<b>Single checkbox</b> <input checked="" type="checkbox"/> Choice A	<b>Checkbox group</b> <input checked="" type="checkbox"/> Choice 1 <input type="checkbox"/> Choice 2 <input type="checkbox"/> Choice 3	<b>Date input</b> <input type="text" value="2014-01-01"/>
<b>Date range</b> <input type="text" value="2017-06-21"/> to <input type="text" value="2017-06-21"/>	<b>File input</b> <input type="file" value="Browse..."/> No file selected	<b>Help text</b> Note: help text isn't a true widget, but it provides an easy way to add text to accompany other widgets.	<b>Numeric input</b> <input type="text" value="1"/>
<b>Radio buttons</b> <input checked="" type="radio"/> Choice 1 <input type="radio"/> Choice 2 <input type="radio"/> Choice 3	<b>Select box</b> <input type="select" value="Choice 1"/>	<b>Sliders</b>  	<b>Text input</b> <input type="text" value="Enter text..."/>

## Inputs de texto livre

- ▶ **textInput()**: coleta pequenas quantidades de texto
- ▶ **passwordInput()**: coleta senhas
- ▶ **textAreaInput()**: coleta parágrafos de texto

```
ui <- fluidPage(  
  textInput("name", "What's your name?"),  
  passwordInput("password", "What's your password?"),  
  textAreaInput("story", "Tell me about yourself",  
               rows = 3)  
)
```

# Inputs de texto livre



127.0.0.1:6380

What's your name?

Paula Medina Maçaira

What's your password?

.....

Tell me about yourself

Paula M Macaira - bacharel em Estatística,  
Msc em Eng. Elétrica e Dsc em Eng. de  
Produção.

## Inputs numéricos

- ▶ **numericInput()**: coleta valores numéricos através de uma caixa de texto restrita
- ▶ **sliderInput()**: coleta valores numéricos através de um controle deslizante
  - ▶ se você fornecer um vetor numérico de comprimento 2 para o valor padrão, obterá um controle deslizante de “intervalo” com duas extremidades

```
ui <- fluidPage(  
  numericInput("num", "Number one",  
               value = 0, min = 0, max = 100),  
  sliderInput("num2", "Number two",  
             value = 50, min = 0, max = 100),  
  sliderInput("rng", "Range",  
             value = c(10, 20), min = 0, max = 100)  
)
```

# Inputs numéricos



127.0.0.1:6380

## Number one

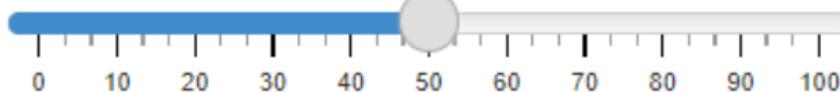
0

## Number two

0

50

100



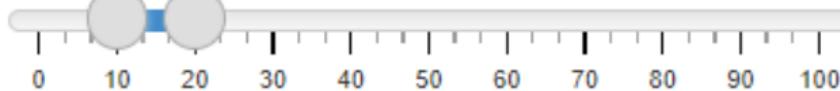
## Range

0

10

20

100



## Inputs de datas

- ▶ **dateInput()**: coleta um único dia
- ▶ **dateRangeInput()**: coleta um intervalo de dois dias

```
ui <- fluidPage(  
  dateInput("dob",  
            "When were you born?"),  
  dateRangeInput("holiday",  
                "When do you want to go on vacation next?")  
)
```

## Inputs de datas



When do you want to go on vacation next?

2022-11-09	to	2022-11-09
------------	----	------------

- ▶ O formato da data, o idioma e o dia em que a semana começa são padronizados para os padrões dos EUA
- ▶ Se você estiver criando um aplicativo com um público internacional, defina o formato, o idioma e o início da semana para que as datas sejam naturais para seus usuários

## Inputs limitados

- ▶ **selectInput()**: coleta uma única informação a partir de um dropdown
- ▶ **radioButtons()**: coleta uma única informação mostrando todas as opções possíveis

```
animals <- c("dog", "cat", "mouse", "bird",
           "other", "I hate animals")

ui <- fluidPage(
  selectInput("state",
              "What's your favourite state?",
              state.name),
  radioButtons("animal",
              "What's your favourite animal?",
              animals)
)
```

# Inputs limitados



127.0.0.1:6380

What's your favourite state?

Alabama



What's your favourite animal?

- dog
- cat
- mouse
- bird
- other
- I hate animals

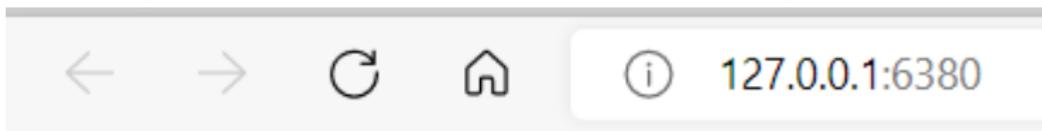
## Radio buttons

- ▶ Os Radio buttons têm dois recursos interessantes:
  - ▶ eles mostram todas as opções possíveis, tornando-os adequados para listas curtas e, por meio dos argumentos **choiceNames/choiceValues**, eles podem exibir outras opções além de texto simples
  - ▶ **choiceNames** determina o que é mostrado ao usuário; **choiceValues** determina o que é retornado em sua função de servidor

## Radio buttons

```
ui <- fluidPage(  
  radioButtons("rb", "Choose one:",  
    choiceNames = list(  
      icon("angry"),  
      icon("smile"),  
      icon("sad-tear"))  
  ),  
  choiceValues = list("angry", "happy", "sad")  
)  
)
```

## Radio buttons



**Choose one:**

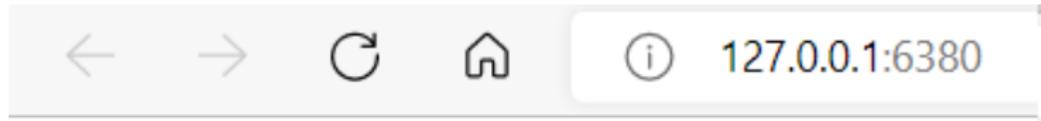
- 😠
- 😊
- 😢

## Diversas seleções de input

- ▶ Não há como selecionar vários valores com Radio Buttons, mas há uma alternativa conceitualmente semelhante:  
**checkboxGroupInput()**

```
ui <- fluidPage(  
  checkboxGroupInput("animal",  
    "What animals do you like?",  
    animals)  
)
```

## checkboxGroupInput()



What animals do you like?

- dog
- cat
- mouse
- bird
- other
- I hate animals

## checkboxInput()

- ▶ Se você quiser uma única caixa de seleção para uma única pergunta sim/não, use **checkboxInput()**

```
ui <- fluidPage(  
  checkboxInput("cleanup", "Clean up?", value = TRUE),  
  checkboxInput("shutdown", "Shutdown?")  
)
```



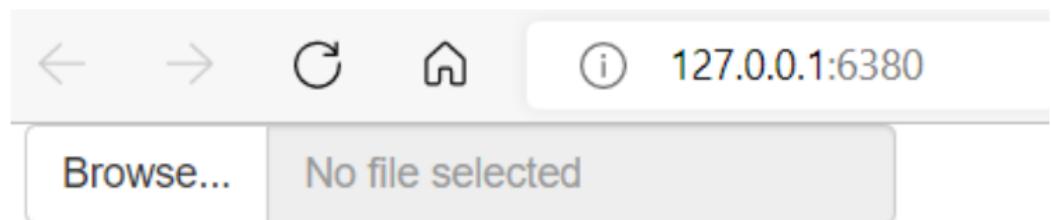
Clean up?

Shutdown?

## Upload de arquivos

- ▶ **fileInput()**: permite que o usuário carregue um arquivo
  - ▶ esse comando requer um tratamento especial pelo lado do servidor

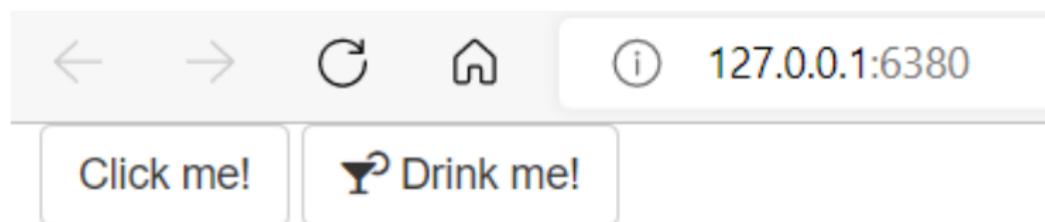
```
ui <- fluidPage(  
  fileInput("upload", NULL)  
)
```



## Botões de ação

- Deixe o usuário realizar uma ação com **actionButton()** ou **actionLink()**

```
ui <- fluidPage(  
  actionButton("click", "Click me!"),  
  actionButton("drink", "Drink me!",  
              icon = icon("cocktail"))  
)
```



## Outputs

- ▶ As saídas na interface do usuário criam espaços reservados que são posteriormente preenchidos pela função do servidor
- ▶ Assim como as entradas, as saídas recebem um ID exclusivo como primeiro argumento: se sua especificação de interface do usuário criar uma saída com o ID “**plot**”, você a acessará na função do servidor com **output\$plot**
- ▶ Cada função de saída no front-end é acoplada a uma função de renderização no back-end

## Tipos de output

- ▶ Existem três tipos principais de saída, correspondentes às três coisas que você normalmente inclui em um relatório: texto, tabelas e gráficos
- ▶ A seguir vamos aprender o básico das funções de saída no front-end, juntamente com as funções de renderização correspondentes no back-end

## Output texto

- ▶ **textOutput()**: saída de texto regular
- ▶ **verbatimTextOutput()**: saída de código e de console

```
ui <- fluidPage(  
  textOutput("text"),  
  verbatimTextOutput("code"))  
)  
server <- function(input, output, session) {  
  output$text <- renderText({  
    "Hello friend!"  
  })  
  output$code <- renderPrint({  
    summary(1:10)  
  })  
}
```

## Output texto

A screenshot of a web browser window. The address bar shows the URL `127.0.0.1:6380`. Below the address bar, there is a message `Hello friend!`. Underneath this message is a table containing statistical summary data.

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.00	3.25	5.50	5.50	7.75	10.00

## Diferenças entre os comandos

- ▶ Observe que existem duas funções de renderização que se comportam de forma ligeiramente diferente:
  - ▶ **renderText()** combina o resultado em uma única string e geralmente é emparelhado com **textOutput()**
  - ▶ **renderPrint()** imprime o resultado, como se você estivesse em um console R, e geralmente é emparelhado com **verbatimTextOutput()**

## Output tabelas

- ▶ Existem duas opções para exibir data frames em tabelas:
  - ▶ **tableOutput()** e **renderTable()** renderizam uma tabela estática de dados, mostrando todos os dados de uma vez. Mais útil para resumos pequenos e fixos (por exemplo, coeficientes de modelo).
  - ▶ **dataTableOutput()** e **renderDataTable()** renderizam uma tabela dinâmica, mostrando um número fixo de linhas junto com controles para alterar quais linhas são visíveis. Mais apropriado se você deseja expor um data frame completo ao usuário.

## Output tabelas

```
ui <- fluidPage(  
  tableOutput("static"),  
  dataTableOutput("dynamic"))  
)  
server <- function(input, output, session) {  
  output$static <- renderTable({  
    head(mtcars)  
  })  
  output$dynamic <- renderDataTable({  
    mtcars  
  }, options = list(pageLength = 5))  
}
```

# Output tabelas

← → ⌂ 127.0.0.1:6380

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21.00	6.00	160.00	110.00	3.90	2.62	16.46	0.00	1.00	4.00	4.00
21.00	6.00	160.00	110.00	3.90	2.88	17.02	0.00	1.00	4.00	4.00
22.80	4.00	108.00	93.00	3.85	2.32	18.61	1.00	1.00	4.00	1.00
21.40	6.00	258.00	110.00	3.08	3.21	19.44	1.00	0.00	3.00	1.00
18.70	8.00	360.00	175.00	3.15	3.44	17.02	0.00	0.00	3.00	2.00
18.10	6.00	225.00	105.00	2.76	3.46	20.22	1.00	0.00	3.00	1.00

Show  entries Search:

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21	6	160	110	3.9	2.62	16.46	0	1	4	4
21	6	160	110	3.9	2.875	17.02	0	1	4	4
22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
18.7	8	360	175	3.15	3.44	17.02	0	0	3	2

Showing 1 to 5 of 32 entries

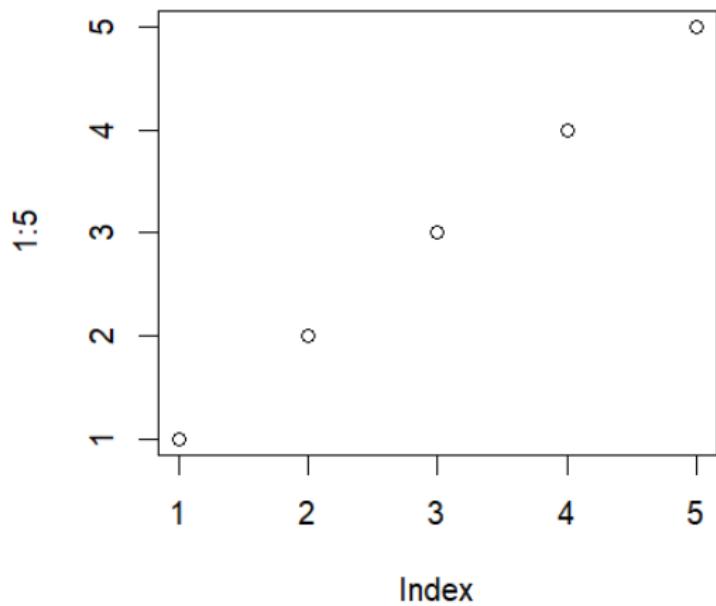
Previous 1 2 3 4 5 6 7 Next

## Output gráficos

- Você pode exibir qualquer tipo de gráfico R (base, ggplot2 ou outro) com **plotOutput()** e **renderPlot()**

```
ui <- fluidPage(  
  plotOutput("plot", width = "400px")  
)  
server <- function(input, output, session) {  
  output$plot <- renderPlot(plot(1:5), res = 96)  
}
```

# Output gráficos



## Output gráficos

- ▶ Por default, **plotOutput()** ocupará toda a largura de sua janela e terá 400 pixels de altura
- ▶ Você pode substituir esses padrões pelos argumentos de altura e largura - é recomendado sempre definir res = 96, pois isso fará com que seus gráficos Shiny correspondam ao que você vê no RStudio o mais próximo possível

## O básico da reatividade

- ▶ Um aplicativo será muito chato se tiver apenas entradas ou apenas saídas
- ▶ A verdadeira magia do Shiny acontece quando você tem um app com ambos
- ▶ No Shiny, você expressa a lógica do seu servidor usando programação reativa
- ▶ A ideia-chave da programação reativa é especificar dependências para que, quando uma entrada for alterada, todas as saídas relacionadas sejam atualizadas automaticamente
- ▶ Isso torna o fluxo de um aplicativo consideravelmente mais simples, mas leva um tempo para entender como tudo se encaixa
- ▶ Este curso fornecerá uma introdução suave à programação reativa, ensinando o básico das construções reativas mais comuns que você usará em aplicativos Shiny

## Um exemplo com reatividade

```
ui <- fluidPage(  
 textInput("name", "What's your name?"),  
  textOutput("greeting")  
)  
  
server <- function(input, output, session) {  
  output$greeting <- renderText({  
    paste0("Hello ", input$name, " !")  
  })  
}
```

---

What's your name?

Paula

Hello Paula!

## Um exemplo um pouco mais complexo<sup>3</sup>

```
dados = read.csv("https://raw.githubusercontent.com/wcota/covid19br/master/cases-brazil-states.csv")
dados = dados[which(dados$state == "TOTAL"),]
variaveis = names(dados)[6:26]

ui <- fluidPage(
  selectInput("variavel",
              "Qual informação?",
              variaveis),
  plotOutput("plot", width = "400px")
)

server <- function(input, output, session) {
  x1 <- reactive({
    dados_BR = dados[,which(names(dados) == input$variavel)]
  })
  output$plot <- renderPlot({
    plot(x1(), type = "l")
  }, res = 96)
}
```

---

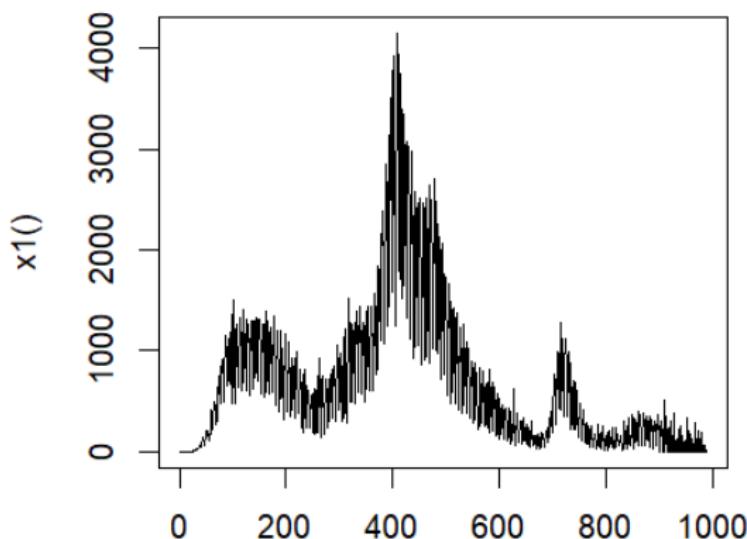
<sup>3</sup><https://raw.githubusercontent.com/wcota/covid19br/master/cases-brazil-states.csv>

## Um exemplo um pouco mais complexo



Qual informação?

newDeaths



## Resumo desta segunda parte

- ▶ Você foi apresentado as principais funções de entrada e saída que compõem o front-end de um aplicativo Shiny
- ▶ Este foi um grande despejo de informações, então não espere lembrar de tudo depois de uma única leitura
- ▶ Em vez disso, volte a estes slides quando estiver procurando por um componente específico

## **Dia 2:**

- ▶ um pouco sobre layouts
- ▶ pacotes plotly, dygraphs e leaflet
- ▶ como publicar seu Shiny App

Vamos recomeçar!

## **Dia 2**

## Layouts

- ▶ Até agora a gente simplesmente usou o **fluidPage()** para organizar as entradas e saídas do nosso app
- ▶ Embora essa função seja boa para aprender Shiny, ela não cria aplicativos úteis ou visualmente atraentes, então agora é hora de aprender mais algumas funções de layout

## Funções de layout

- ▶ As funções de layout fornecem a estrutura visual de alto nível de um aplicativo
- ▶ Os layouts são criados por uma hierarquia de chamadas de função, onde a hierarquia em R corresponde à hierarquia no HTML gerado
- ▶ Isso ajuda você a entender o código de layout

## Exemplo de layout

- Quando você olha para o código de layout como este:

```
fluidPage(  
  titlePanel("Hello Shiny!"),  
  sidebarLayout(  
    sidebarPanel(  
      sliderInput("obs", "Observations:",  
                  min = 0, max = 1000,  
                  value = 500)  
    ),  
    mainPanel(  
      plotOutput("distPlot")  
    )  
  )  
)
```

## Exemplo de layout

- ▶ Concentre-se na hierarquia das chamadas de função:

```
fluidPage(  
  titlePanel(),  
  sidebarLayout(  
    sidebarPanel(),  
    mainPanel()  
  )  
)
```

## Compreendendo o exemplo de layout

- ▶ Mesmo que você ainda não tenha aprendido essas funções, você pode adivinhar o que está acontecendo lendo seus nomes
- ▶ Você pode imaginar que esse código gerará um design de aplicativo clássico:
  - ▶ uma barra de título na parte superior,
  - ▶ seguida por uma barra lateral (contendo um controle deslizante),
  - ▶ e um painel principal (contendo um gráfico)
- ▶ A capacidade de ver facilmente a hierarquia por meio do recuo é uma das razões pelas quais é uma boa ideia usar um estilo consistente

## `fluidPage()`, `fixedPage()` e `fillPage()`

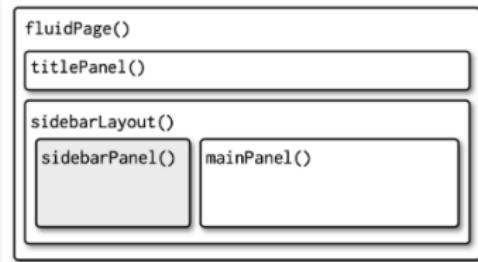
- ▶ A função de layout mais importante, mas menos interessante, é **fluidPage()**, que você viu em praticamente todos os exemplos até agora
- ▶ Além de **fluidPage()**, o Shiny fornece algumas outras funções de página que podem ser úteis em situações mais especializadas:
  - ▶ **fixedPage()** funciona como `fluidPage()`, mas tem uma largura máxima fixa, o que impede que seus aplicativos se tornem excessivamente largos em telas maiores
  - ▶ **fillPage()** preenche toda a altura do navegador e é útil se você quiser fazer um gráfico que ocupe toda a tela

## Página com barra lateral

- ▶ Para fazer layouts mais complexos, você precisará chamar funções de layout dentro de **fluidPage()**
- ▶ Por exemplo, para fazer um layout de duas colunas com entradas à esquerda e saídas à direita, você pode usar **sidebarLayout()**
  - ▶ junto com seus amigos **titlePanel()**, **sidebarPanel()** e **mainPanel()**

# Página com barra lateral

```
fluidPage(  
  titlePanel(  
    # app title/description  
)  
,  
  sidebarLayout(  
    sidebarPanel(  
      # inputs  
)  
,  
    mainPanel(  
      # outputs  
)  
)  
)
```



## Exemplo de página com barra lateral

- ▶ Aplicativo muito simples que demonstra o Teorema do Limite Central

```
ui <- fluidPage(  
  titlePanel("Central limit theorem"),  
  sidebarLayout(  
    sidebarPanel(  
      numericInput("m", "Number of samples:",  
                  2, min = 1, max = 100)  
    ),  
    mainPanel(  
      plotOutput("hist")  
    )  
  )  
)  
  
server <- function(input, output, session) {  
  output$hist <- renderPlot({  
    means <- replicate(1e4, mean(runif(input$m)))  
    hist(means, breaks = 20)  
  }, res = 96)  
}
```

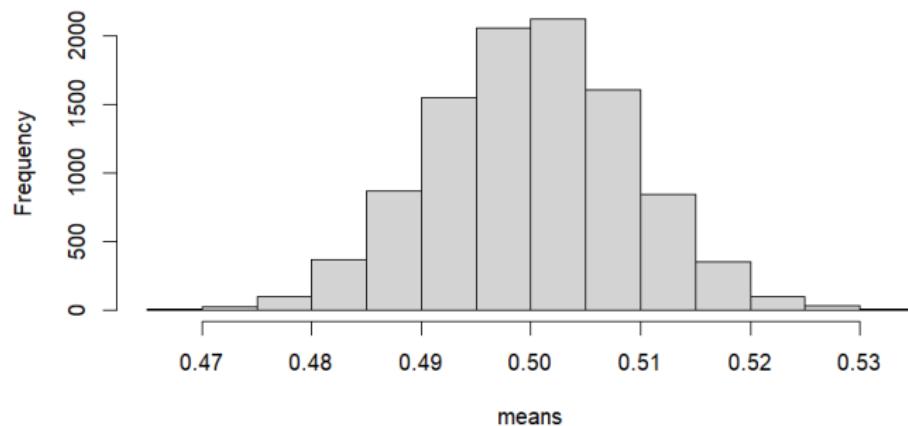
# Exemplo de página com barra lateral

## Central limit theorem

Number of samples:

1000

Histogram of means

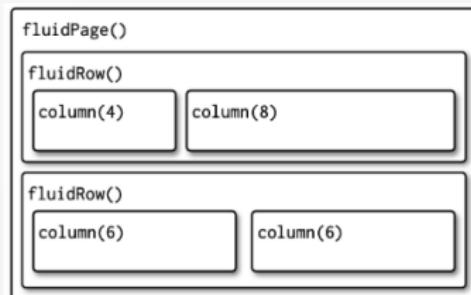


## Layout com várias linhas

- ▶ Nos bastidores, **sidebarLayout()** é construído em cima de um layout flexível de várias linhas, que você pode usar diretamente para criar aplicativos visualmente mais complexos
- ▶ Como de costume, você começa com **fluidPage()**, em seguida, você cria linhas com **fluidRow()** e colunas com **column()**

# Layout com várias linhas

```
fluidPage(  
  fluidRow(  
    column(4,  
      ...  
    ),  
    column(8,  
      ...  
    )  
  ),  
  fluidRow(  
    column(6,  
      ...  
    ),  
    column(6,  
      ...  
    )  
  )  
)
```



## Layout com várias linhas

- ▶ Cada linha é composta por **12 colunas** e o primeiro argumento para **column()** fornece quantas dessas colunas devem ser ocupadas
- ▶ Um layout de 12 colunas oferece flexibilidade substancial porque você pode criar facilmente layouts de 2, 3 ou 4 colunas ou usar colunas estreitas para criar espaçadores

## Layouts de várias páginas

- ▶ À medida que seu aplicativo cresce em complexidade, pode se tornar impossível ajustar tudo em uma única página
- ▶ Vamos aprender vários usos de **tabPanel()** que criam a ilusão de várias páginas
- ▶ Isso é uma ilusão porque você ainda terá um único aplicativo com um único arquivo HTML subjacente, mas agora está dividido em partes e apenas uma parte fica visível por vez

## Tabssets

- ▶ A maneira simples de dividir uma página em pedaços é usar **tabsetPanel()** e seu amigo próximo **tabPanel()**
- ▶ Como você pode ver no código a seguir, **tabsetPanel()** cria um container para qualquer número de **tabPanels()**, que por sua vez pode conter quaisquer outros componentes HTML

## Exemplo de uso do tabsetPanel()

```
ui <- fluidPage(  
  tabsetPanel(  
    tabPanel("Import data",  
      fileInput("file", "Data", buttonLabel = "Upload..."),  
      textInput("delim", "Delimiter (leave blank to guess)", ""),  
      numericInput("skip", "Rows to skip", 0, min = 0),  
      numericInput("rows", "Rows to preview", 10, min = 1)  
    ),  
    tabPanel("Set parameters"),  
    tabPanel("Visualise results")  
  )  
)
```

# Exemplo de uso do tabsetPanel()

Import data   Set parameters   Visualise results

**Data**

Upload... No file selected

**Delimiter (leave blank to guess)**

**Rows to skip**

**Rows to preview**

## Navlists e navbars

- ▶ Como as abas (tabset) são exibidas horizontalmente, há um limite fundamental para quantas guias você pode usar, principalmente se tiverem títulos longos
- ▶ **navbarPage()** e **navbarMenu()** fornecem dois layouts alternativos que permitem usar mais guias com títulos mais longos
  - ▶ **navlistPanel()** é semelhante a **tabsetPanel()**, mas em vez de executar os títulos das guias horizontalmente, ele os mostra verticalmente em uma barra lateral

## Exemplo de uso do navlistPanel()

```
ui <- fluidPage(  
  navlistPanel(  
    id = "tabset",  
    "Heading 1",  
    tabPanel("panel 1", "Panel one contents"),  
    "Heading 2",  
    tabPanel("panel 2", "Panel two contents"),  
    tabPanel("panel 3", "Panel three contents")  
  )  
)
```

# Exemplo de uso do navlistPanel()

Panel one contents

Heading 1

panel 1

Heading 2

panel 2

panel 3

## Usando o navbarPage()

- ▶ Outra abordagem é usar **navbarPage()**: ele ainda executa os títulos das guias horizontalmente, mas você pode usar **navbarMenu()** para adicionar menus suspensos para um nível adicional de hierarquia

```
ui <- navbarPage(  
  "Page title",  
  tabPanel("panel 1", "one"),  
  tabPanel("panel 2", "two"),  
  tabPanel("panel 3", "three"),  
  navbarMenu("subpanels",  
    tabPanel("panel 4a", "four-a"),  
    tabPanel("panel 4b", "four-b"),  
    tabPanel("panel 4c", "four-c"))  
)  
)
```

## Exemplo de uso do navbarPage()

Page title	panel 1	panel 2	panel 3	subpanels ▾	
one				<ul style="list-style-type: none"><li>panel 4a</li><li>panel 4b</li><li>panel 4c</li></ul>	

## Temas pré-fabricados

- ▶ A maneira mais fácil de alterar a aparência geral do seu aplicativo é escolher um tema “bootswatch” pré-fabricado usando o argumento **bootswatch** para **bslib::bs\_theme()**
- ▶ A Figura do próximo slide mostra os resultados do código a seguir, trocando “darkly” por outros temas

```
ui <- fluidPage(  
  theme = bslib::bs_theme/bootswatch = "darkly"/),  
  sidebarLayout(  
    sidebarPanel(  
     textInput/"txt", "Text input:", "text here"),  
      sliderInput/"slider", "Slider input:", 1, 100, 30)  
    ),  
    mainPanel(  
      h1(paste0/Theme: darkly/)),  
      h2/Header 2/),  
      p/Some text/  
    )  
  )  
)
```

# Temas pré-fabricados

Text input:

Slider input:  
  
1 20 30 40 50 60 70 80 90 100

Theme: darkly

## Header 2

Some text

Text input:

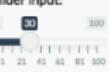
Slider input:  
  
1 20 30 40 50 60 70 80 90 100

Theme: sandstone

## Header 2

Some text

Text input:

Slider input:  
  
1 20 30 40 50 60 70 80 90 100

Theme: flatly

## Header 2

Some text

Text input:

Slider input:  
  
1 20 30 40 50 60 70 80 90 100

Theme: united

## Header 2

Some text

The same app styled with four bootswatch themes: darkly, flatly, sandstone, and united

## Resumo desta terceira parte

- ▶ Esta terceira parte forneceu as ferramentas necessárias para criar aplicativos Shiny complexos e atraentes
- ▶ Você aprendeu as funções Shiny que permitem criar layouts de aplicativos de página única e de várias páginas (como **fluidPage()** e **tabsetPanel()**) e como personalizar a aparência visual geral com temas

## Outputs interativos (gráficos e mapas)

- ▶ Quando estamos construindo páginas ou aplicações Web (ou apresentações de slides em HTML), além de gráficos em formato de imagem, podemos construir visualizações utilizando bibliotecas JavaScript
- ▶ Essas visualizações permitem animações e possuem diversas funcionalidades interativas, como tooltips, filtros, zoom e drilldown
- ▶ Neste curso vamos falar especificamente das bibliotecas `plotly`, `dygraphs` e `leaflet`

## plotly

- ▶ Para adicionar um plotly no Shiny, criado a partir da função **ggplotly()** ou da função **plot\_ly()**, utilizamos o par de funções plotly:
  - ▶ **plotlyOutput()** e **renderPlotly()**
- ▶ Na função **renderPlotly()**, basta passarmos um código que retorne um gráfico plotly, utilizando os inputs para especificar as variáveis

# Exemplo com o plotly

```
library(shiny)
library(ggplot2)
library(tidyverse)

vars <- names(mtcars)

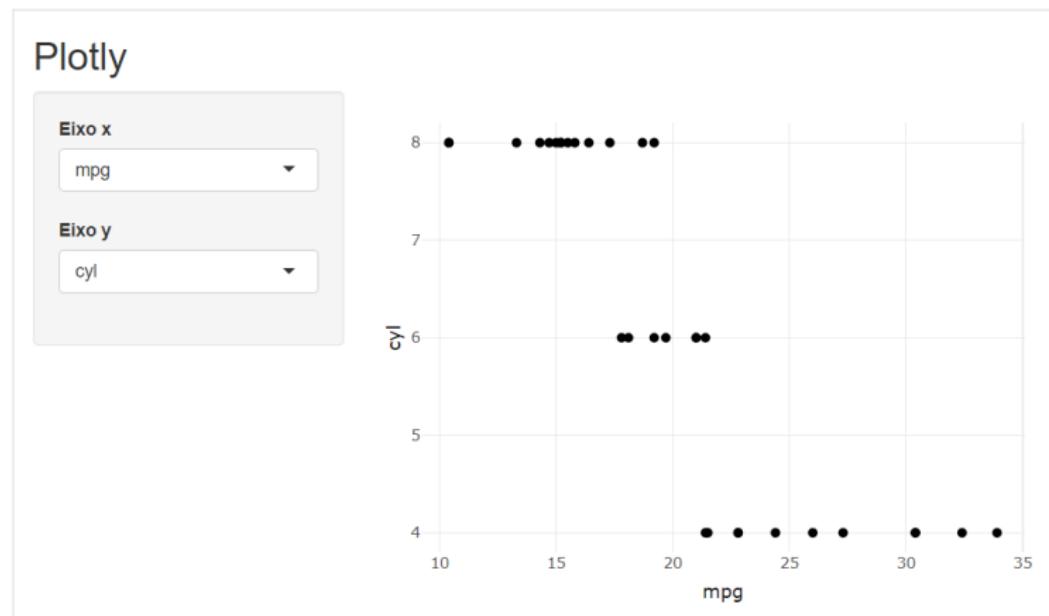
ui <- fluidPage(
  titlePanel("Plotly"),
  sidebarLayout(
    sidebarPanel(
      selectInput("x","Eixo x",choices = vars),
      selectInput("y","Eixo y",choices = vars,
                 selected = vars[2])
    ),
    mainPanel(plotly::plotlyOutput("plot"))
  )
)

server <- function(input, output, session) {
  output$plot <- plotly::renderPlotly({
    p <- mtcars %>%
      tibble::rownames_to_column() %>%
      ggplot(aes(x = .data[[input$x]],
                 y = .data[[input$y]],
                 text = rowname)) +
      geom_point() +
      theme_minimal()

    plotly::ggplotly(p)
  })
}

shinyApp(ui, server)
```

# Exemplo com o plotly



## dygraphs

- ▶ Para adicionar um dygraph no Shiny, utilizamos o par de funções dygraphs:
  - ▶ **dygraphOutput()** e **renderDygraph()**
- ▶ Na função **renderDygraph()**, basta passarmos um código que retorne um gráfico dygraph, utilizando os inputs para especificar as variáveis

## Exemplo com o dygraphs

```
library(shiny)
library(dygraphs)

ui <- fluidPage(
  titlePanel("Predicted Deaths from Lung Disease (UK)"),
  sidebarLayout(
    sidebarPanel(
      numericInput("months", label = "Months to Predict",
                  value = 72, min = 12, max = 144, step = 12),
      selectInput("interval", label = "Prediction Interval",
                  choices = c("0.80", "0.90", "0.95", "0.99"),
                  selected = "0.95"),
      checkboxInput("showgrid", label = "Show Grid", value = TRUE)
    ),
    mainPanel(
      dygraphOutput("dygraph")
    )
  )
)

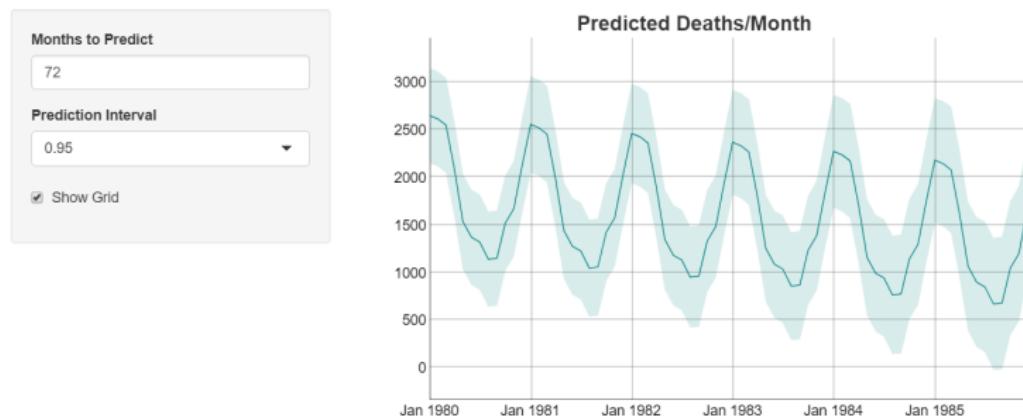
server <- function(input, output, session) {
  predicted <- reactive({
    hw <- HoltWinters(ldeaths)
    predict(hw, n.ahead = input$months,
            prediction.interval = TRUE,
            level = as.numeric(input$interval))
  })

  output$dygraph <- renderDygraph({
    dygraph(predicted(), main = "Predicted Deaths/Month") %>%
      dySeries(c("lwr", "fit", "upr"), label = "Deaths") %>%
      dyOptions(drawGrid = input$showgrid)
  })
}

shinyApp(ui, server)
```

# Exemplo com o dygraphs

Predicted Deaths from Lung Disease (UK)



## leaflet

- ▶ Para adicionar um leaflet (mapa) no Shiny, utilizamos o par de funções leaflet:
  - ▶ **leafletOutput()** e **renderLeaflet()**
- ▶ Na função **renderLeaflet()**, basta passarmos um código que retorne um mapa leaflet, utilizando os inputs para especificar as variáveis

# Exemplo com o leaflet

```
library(shiny)
library(leaflet)

r_colors <- rgb(t(col2rgb(colors()) / 255))
names(r_colors) <- colors()

ui <- fluidPage(
  leafletOutput("mymap"),
  p(),
  actionButton("recalc", "New points")
)

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

  points <- eventReactive(input$recalc, {
    cbind(rnorm(40) * 2 + 13, rnorm(40) + 48)
  }, ignoreNULL = FALSE)

  output$mymap <- renderLeaflet({
    leaflet() %>%
      addProviderTiles(providers$Stamen.TonerLite,
                      options = providerTileOptions(noWrap = TRUE))
    ) %>%
    addMarkers(data = points())
  })
}

shinyApp(ui, server)
```

# Exemplo com o leaflet



## Publicando seu ShinyApp

- ▶ Agora que você já sabe criar um aplicativo Shiny útil, vamos compartilhá-lo com outras pessoas
- ▶ Se você estiver familiarizado com hospedagem na web ou tiver acesso a um departamento de TI, você mesmo pode hospedar seus aplicativos Shiny
- ▶ Se você preferir uma experiência mais fácil ou precisar de suporte, o RStudio oferece três maneiras de hospedar seu aplicativo Shiny como uma página da web:
  - ▶ shinyapps.io
  - ▶ Shiny Server
  - ▶ RStudio Connect

## Shinyapps.io

- ▶ A maneira mais fácil de transformar seu aplicativo Shiny em uma página da web é usar **shinyapps.io**, o serviço de hospedagem do RStudio para aplicativos Shiny
- ▶ O **shinyapps.io** permite que você carregue seu aplicativo diretamente da sua sessão R para um servidor hospedado pelo RStudio
- ▶ Você tem controle total sobre seu aplicativo, incluindo ferramentas de administração do servidor
- ▶ A conta gratuita permite até 5 aplicativos simultâneos e 25 horas mensais de uso

## Passo a passo

1. Criar uma conta em <https://www.shinyapps.io/>
2. Vá em **Accounts** -> **Tokens** -> *Show* -> *Copy to clipboard* -> *Ctrl+c*
3. **Run App** -> **Publish** (aceita a instalação de todos os pacotes)  
-> *Connect Account ShinyApps.io* -> *Ctrl+v* (colar o Token da sua conta aqui)
4. **Publish**

# Vamos publicar o nosso app

[https://paulamacaira.shinyapps.io/SBPO2022\\_MinicursoShiny/](https://paulamacaira.shinyapps.io/SBPO2022_MinicursoShiny/)

The screenshot shows a web browser window with the URL [https://paulamacaira.shinyapps.io/SBPO2022\\_MinicursoShiny/](https://paulamacaira.shinyapps.io/SBPO2022_MinicursoShiny/). The page title is "Dataset". A dropdown menu is open, showing the option "ability.csv". Below the dropdown, there is a table with the following data:

	Length	Class	Mode
cov	36	-none-	numeric
center	6	-none-	numeric
n.obs	1	-none-	numeric

Below this table is another table with the following data:

cov.general	cov.picture	cov.blocks	cov.maze	cov.reading	cov.vocab	center	n.obs
24.64	5.99	33.52	6.02	20.75	29.70	0.00	112.00
5.99	6.70	18.14	1.78	4.94	7.20	0.00	112.00
33.52	18.14	149.83	19.42	31.43	50.75	0.00	112.00
6.02	1.78	19.42	12.71	4.76	9.07	0.00	112.00
20.75	4.94	31.43	4.76	52.60	66.76	0.00	112.00
29.70	7.20	50.75	9.07	66.76	135.29	0.00	112.00

# Alguns exemplos de app

[https://paulamacaira.shinyapps.io/app\\_map\\_SDG/](https://paulamacaira.shinyapps.io/app_map_SDG/)

Use of wind potential as a driver for socio-economic leverage in underdeveloped locations

Proposed indicator | About the app and us

Choose the variables and the weights to build the proposed indicator  
Proposed Indicator = Normalized Wind Power + Socio Variable + Environmental Variable  
Normal weight indicator is 0.4 (0.1-1)

Wind potential proxy variable  
Select the wind speed metric  
Mean Wind Speed (20m)  
Choose the weight for the wind speed variable (w)  
The recommended proxy variable is 0.6

Social variable  
Select the social variable  
Municipal Human Development Index (2010)  
Choose the weight for the social variable (z)  
The recommended social variable is 0.2

Environmental variable  
Select the environmental conservation variable  
Hab (1) or net (0) environmental conservation area  
Choose the weight for the environmental conservation variable (y)  
The recommended environmental conservation area is 0.2

DEI - DEPARTAMENTO DE INVESTIGACIONES  
FROG - FONDO REGIONAL PARA EL DESARROLLO  
CTC - CENTRO TECNICO DE CONSULTORES  
PLATAFORMA LATAM

Row	Municipal Code	Municipal Name	Proposed Indicator	Normalized Mean Wind Speed (20m)	Municipal Human Development Index (2010)	Hab (1) or net (0) environmental conservation area
1	241001	Tucumán	0.00	1	0.07	0
2	240008	Petén Grande	0.00	1	0.00	0
3	241009	Sikyukap de Sotales	0.00	1	0.00	0
4	240102	Carcara de Tumé	0.04	2.00	0.07	0
5	240102	Rancho Perneras	0.04	2.00	0.00	0
6	241008	Purén	0.04	2.00	0.00	0
7	240202	Casablanca	0.03	2.00	0.00	0
8	241008	Popo Bracho	0.03	2.00	0.10	0
9	240001	Indio Chávez	0.03	2.00	0.0	0
10	240104	Gaitán	0.03	2.00	0.0	0

Showing 1 of 10 of 107 entries

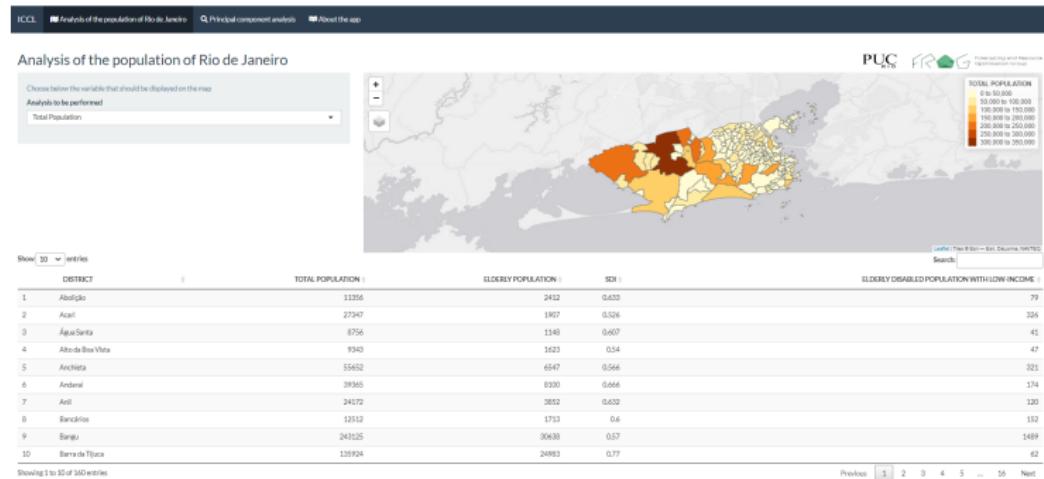
Using the proposed indicator

Using only wind potential proxy variable

Using only social variable

# Alguns exemplos de app

<https://paulamacaira.shinyapps.io/ICCL/>



# Alguns exemplos de app

<https://noispuc.shinyapps.io/effect-br-monitor/>

EFFECT-Brazil Home Cases Deaths Hospitalizations Vaccinations About us

Welcome to EFFECT-Brazil Monitor app!

Cases Explore the data on confirmed COVID-19 cases for Brazil [Take me there](#)

Deaths Explore the data on confirmed COVID-19 deaths for Brazil [Take me there](#)

Hospitalizations Explore the data on confirmed COVID-19 hospitalizations for Brazil [Take me there](#)

Vaccinations Explore data on COVID-19 vaccine uptake and immunization coverage over time in Brazil [Take me there](#)

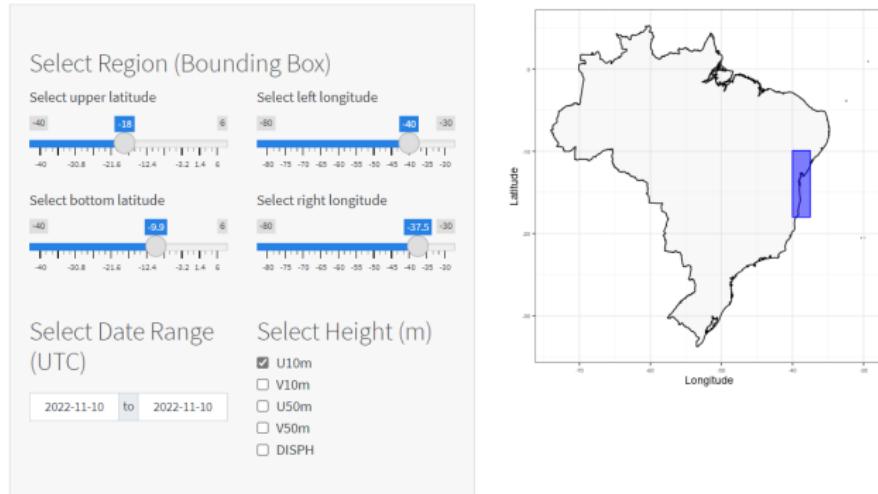
[Twitter](#) [Facebook](#) [Link](#)



# Alguns exemplos de app

[https://paulamacaira.shinyapps.io/MERRA2\\_treatment/](https://paulamacaira.shinyapps.io/MERRA2_treatment/)

Download and treatment of MERRA-2 for Brazil (MERRA-2 BR treatment app)



Source: The Modern-Era Retrospective Analysis for Research and Applications, Version 2 (MERRA-2), Ronald Gelaro, et al., 2017, J. Clim., doi: 10.1175/JCLI-D-16-0758.1

Me ajude avaliando o minicurso

