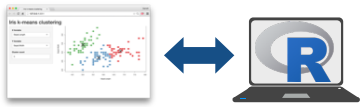
# Interactive Web Apps with shiny Cheat Sheet

learn more at shiny.rstudio.com

RStudio

## Basics

A **Shiny** app is a web page (**UI**) connected to a computer running a live R session (**Server**)



Users can manipulate the UI, which will cause the server to update the UI's displays (by running R code).

### App template

Begin writing a new app with this template. Preview the app by running the code at the R command line.

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

- **ui** - nested R functions that assemble an HTML user interface for your app
- **server** - a function with instructions on how to build and rebuild the R objects displayed in the UI
- **shinyApp** - combines `ui` and `server` into a functioning app. Wrap with **runApp()** if calling from a sourced script or inside a function.

### Share your app


The easiest way to share your app is to host it on shinyapps.io, a cloud based service from RStudio

1. Create a free or professional account at **http://shinyapps.io**
2. Click the **Publish** icon in the RStudio IDE (>=0.99) or run: **rsconnect::deployApp("**<path to directory>**")**

### Build or purchase your own Shiny Server

at www.rstudio.com/products/shiny-server/

## Building an App - Complete the template by adding arguments to fluidPage() and a body to the server function.

Add inputs to the UI with ***Input()** functions

Add outputs with ***Output()** functions

Tell server how to render outputs with R in the server function. To do this:

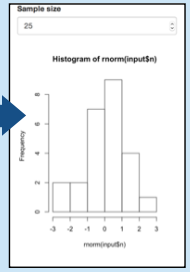
1. Refer to outputs with **output$<id>**
2. Refer to inputs with **input$<id>**
3. Wrap code in a **render*()** function before saving to output

```
library(shiny)

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

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

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




Save your template as **app.R**. Alternatively, split your template into two files named **ui.R** and **server.R**.

```
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", 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",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
```

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

**ui.R** contains everything you would save to ui.

**server.R** ends with the function you would save to server.

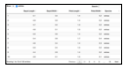
No need to call **shinyApp()**.

Save each app as a directory that contains an **app.R** file (or a **server.R** file and a **ui.R** file) plus optional extra files.


- The directory name is the name of the app
- (optional) defines objects available to both ui.R and server.R
- (optional) used in showcase mode
- (optional) data, scripts, etc.
- (optional) directory of files to share with web browsers (images, CSS, .js, etc.) Must be named "**www**"

Launch apps with `runApp(<path to directory>)`

## Outputs - render*() and *Output() functions work together to add R output to the UI



DT::**renderDataTable(**expr, options, callback, escape, env, quoted**)** works with **dataTableOutput(**outputId, icon, …**)**

**renderImage(**expr, env, quoted, deleteFile**)** — **imageOutput(**outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline**)**

**renderPlot(**expr, width, height, res, …, env, quoted, func**)** — **plotOutput(**outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline**)**

**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, …**)**

## Inputs - collect values from the user

Access the current value of an input object with **input $<inputId>**. Input values are **reactive**.

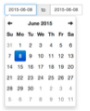
Action — **actionButton(**inputId, label, icon, …**)**

Link — **actionLink(**inputId, label, icon, …**)**

Choice 1 / Choice 2 / Choice 3 — **checkboxGroupInput(**inputId, label, choices, selected, inline**)**

Check me — **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**)**

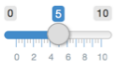
Choose File — **fileInput(**inputId, label, multiple, accept**)**

1 — **numericInput(**inputId, label, value, min, max, step**)**

••••••• — **passwordInput(**inputId, label, value**)**

Choice A / Choice B / Choice C — **radioButtons(**inputId, label, choices, selected, inline**)**

Choice 1 / Choice 1 / Choice 2 — **selectInput(**inputId, label, choices, selected, multiple, selectize, width, size**)** (also **selectizeInput()**)



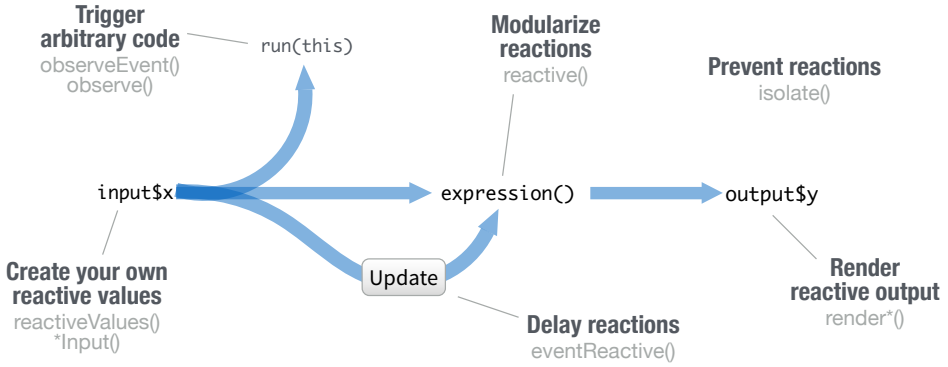

**sliderInput(**inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post**)**

Apply Changes — **submitButton(**text, icon**)** (Prevents reactions across entire app)

Enter text — **textInput(**inputId, label, value**)**

# Reactivity

Reactive values work together with reactive functions. Call a reactive value from within the arguments of one of these functions to avoid the error `Operation not allowed without an active reactive context.`




## Create your own reactive values

```
library(shiny)

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

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

shinyApp(ui, server)
```

***Input() functions**
(see front page)

**reactiveValues(...)**

Each input function creates a reactive value stored as **input$<inputId>**

**reactiveValues()** creates a list of reactive values whose values you can set.

## Render reactive output

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

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

shinyApp(ui, server)
```

**render*() functions**
(see front page)

Builds an object to display. Will rerun code in body to rebuild the object whenever a reactive value in the code changes.

Save the results to **output$<outputId>**

## Prevent reactions

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

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

shinyApp(ui, server)
```

**isolate**(expr)

Runs a code block. Returns a **non-reactive** copy of the results.

## Trigger arbitrary code

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

Runs code in 2nd argument when reactive values in 1st argument change. See **observe()** for alternative.

## Modularize reactions

```
library(shiny)
ui <- fluidPage(
 textInput("a",""),
 textInput("z", "")
)

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

shinyApp(ui, server)
```

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

Creates a **reactive expression** that

- **caches** its value to reduce computation
- can be called by other code
- notifies its dependencies when it ha been invalidated

Call the expression with function syntax, e.g. `re()`

## Delay reactions

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

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)

Creates reactive expression with code in 2nd argument that only invalidates when reactive values in 1st argument change.

# UI

An app's UI is an HTML document. Use Shiny's functions to assemble this HTML with 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>
```

Returns HTML

Add static HTML elements with **tags**, a list of functions that parallel common HTML tags, e.g. **tags$a()**. Unnamed arguments will be passed into the tag; named arguments will become tag attributes.

| | | | | |
|---|---|---|---|---|
| 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 |

The most common tags have wrapper functions. You do not need to prefix their names with **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>")
)
```


To include a CSS file, use **includeCSS()**, or
1. Place the file in the **www** subdirectory
2. Link to it with

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

To include JavaScript, use **includeScript()** or
1. Place the file in the **www** subdirectory
2. Link to it with

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

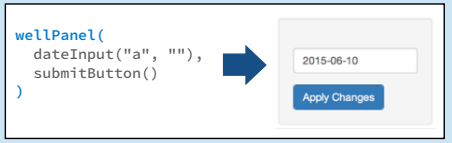
**IMAGES**

To include an image
1. Place the file in the **www** subdirectory
2. Link to it with `img(src="<file name>")`

# Layouts

Combine multiple elements into a "single element" that has its own properties with a panel function, e.g.

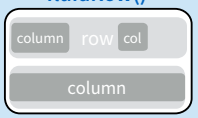
```
wellPanel(
  dateInput("a", ""),
  submitButton()
)
```




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

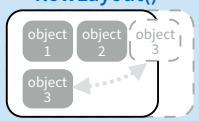
Organize panels and elements into a layout with a layout function. Add elements as arguments of the layout functions.

**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
  )
)
```

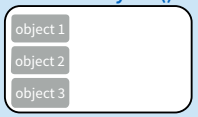
**sidebarLayout()**


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

**splitLayout()**


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

**verticalLayout()**




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

Layer tabPanels on top of each other, and navigate between them, with:

```
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"))
```