08 - Interactive web applications with shiny

Data Science with R · Summer 2021

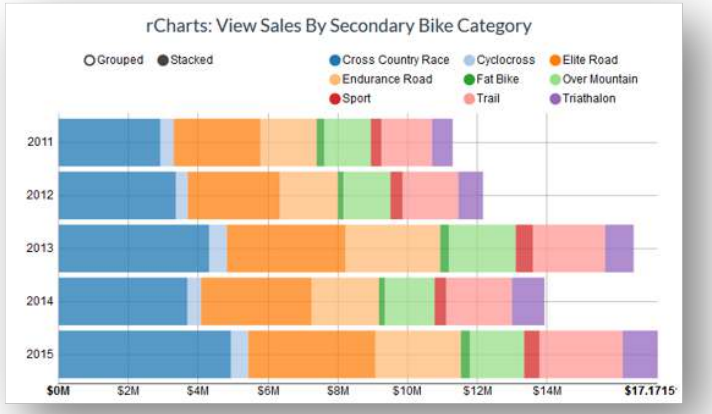
Uli Niemann · Knowledge Management & Discovery Lab

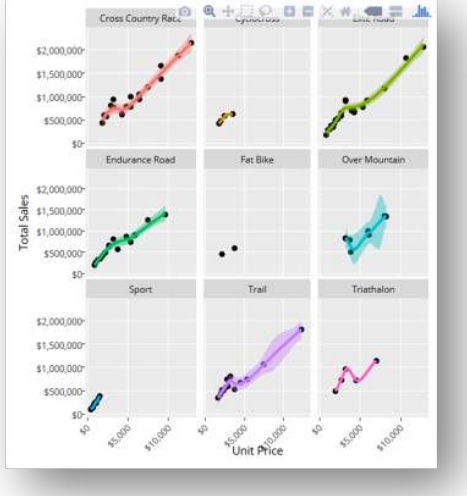
https://brain.cs.uni-magdeburg.de/kmd/DataSciR/

Photo courtesy of Ulrich Arendt

Sales analytics dashboard of a bycicle shop

Analysis and visualization of historic sale transactions, stratified by bike type, bike model and US state.

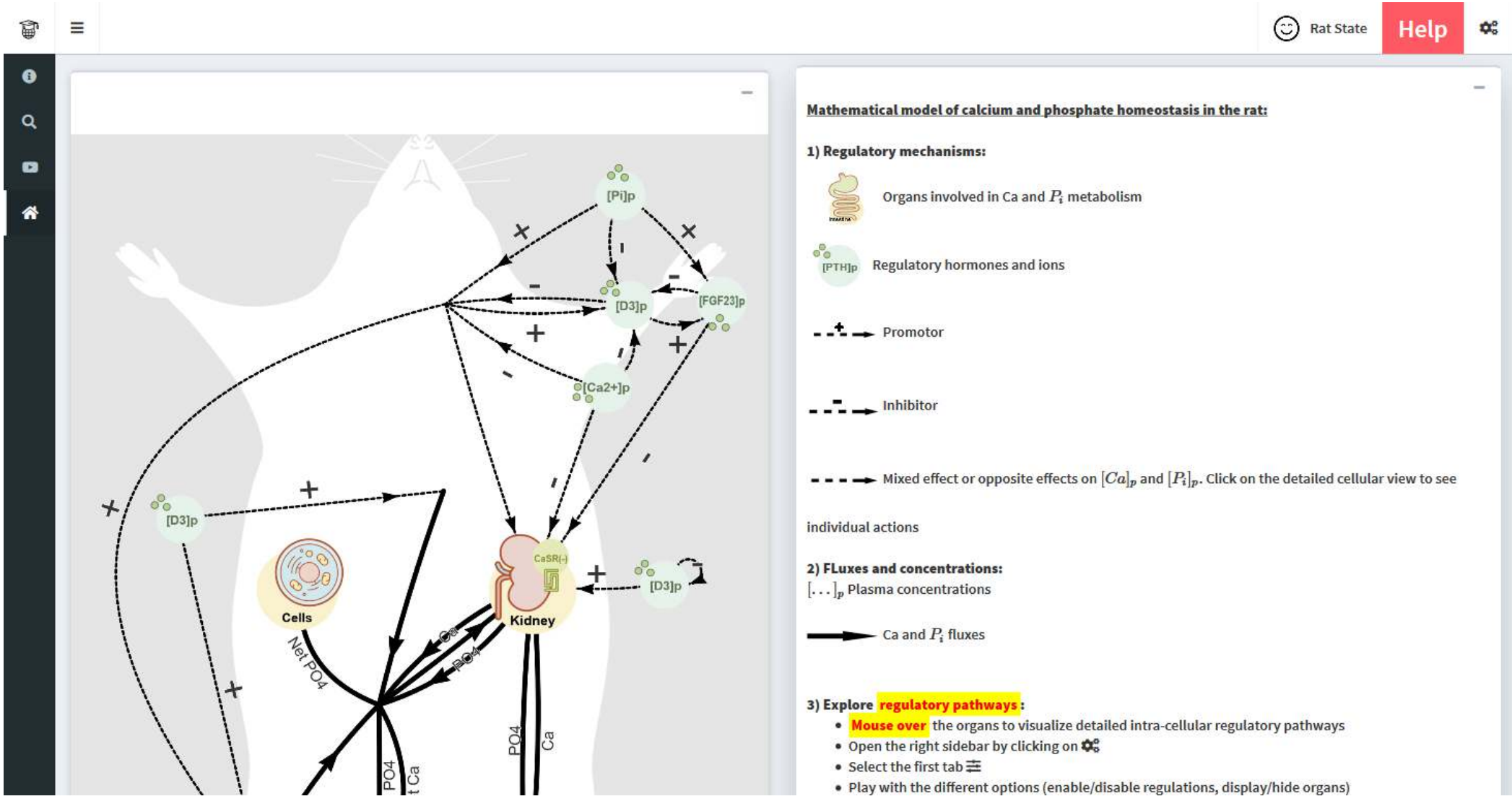




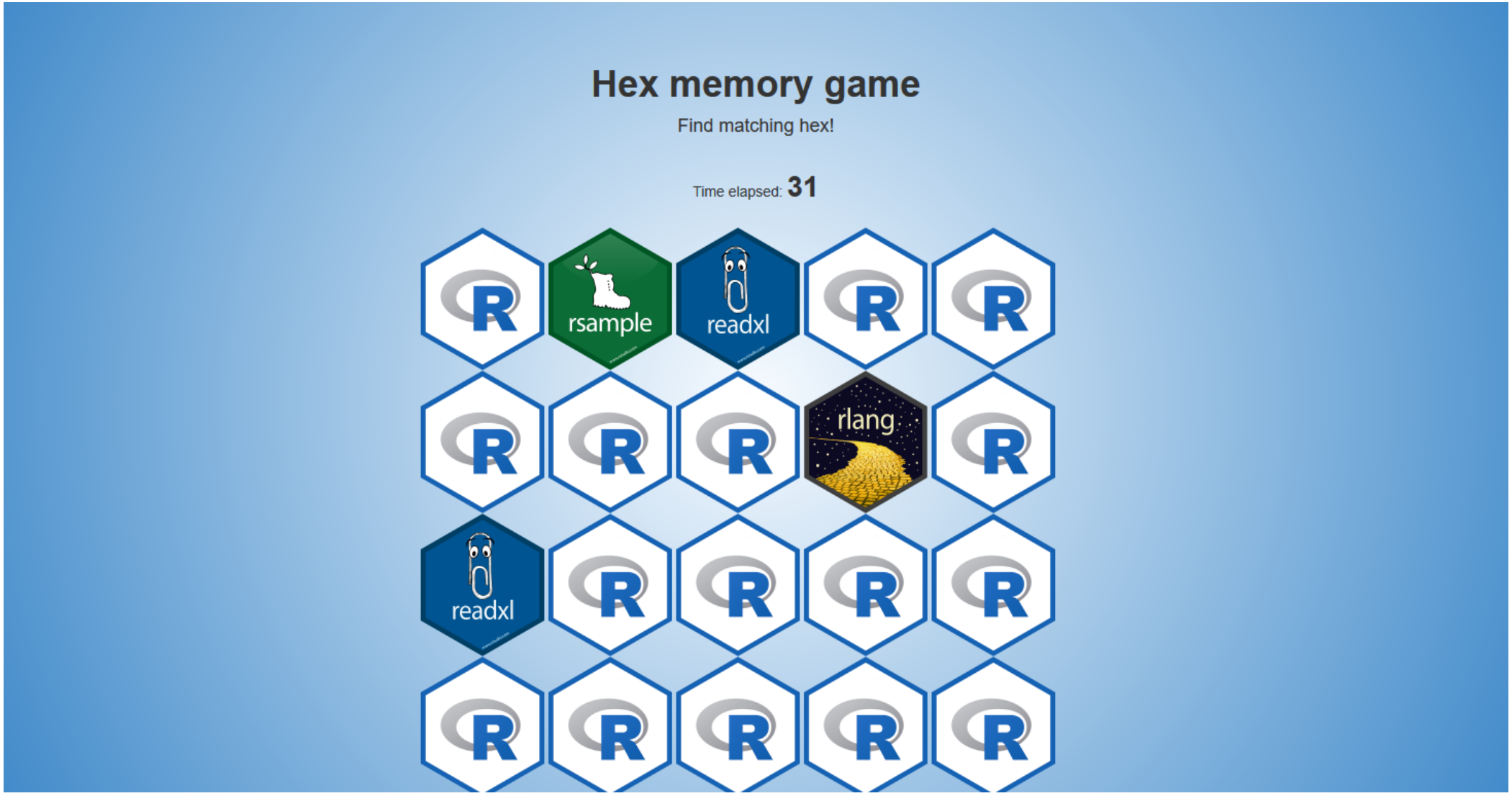
Prevention of hospital infections

Frequency of hospital infections, use of antibiotic agents, antibiotic resistance, operations, ... over time

Network visualiztion of biochemical processes



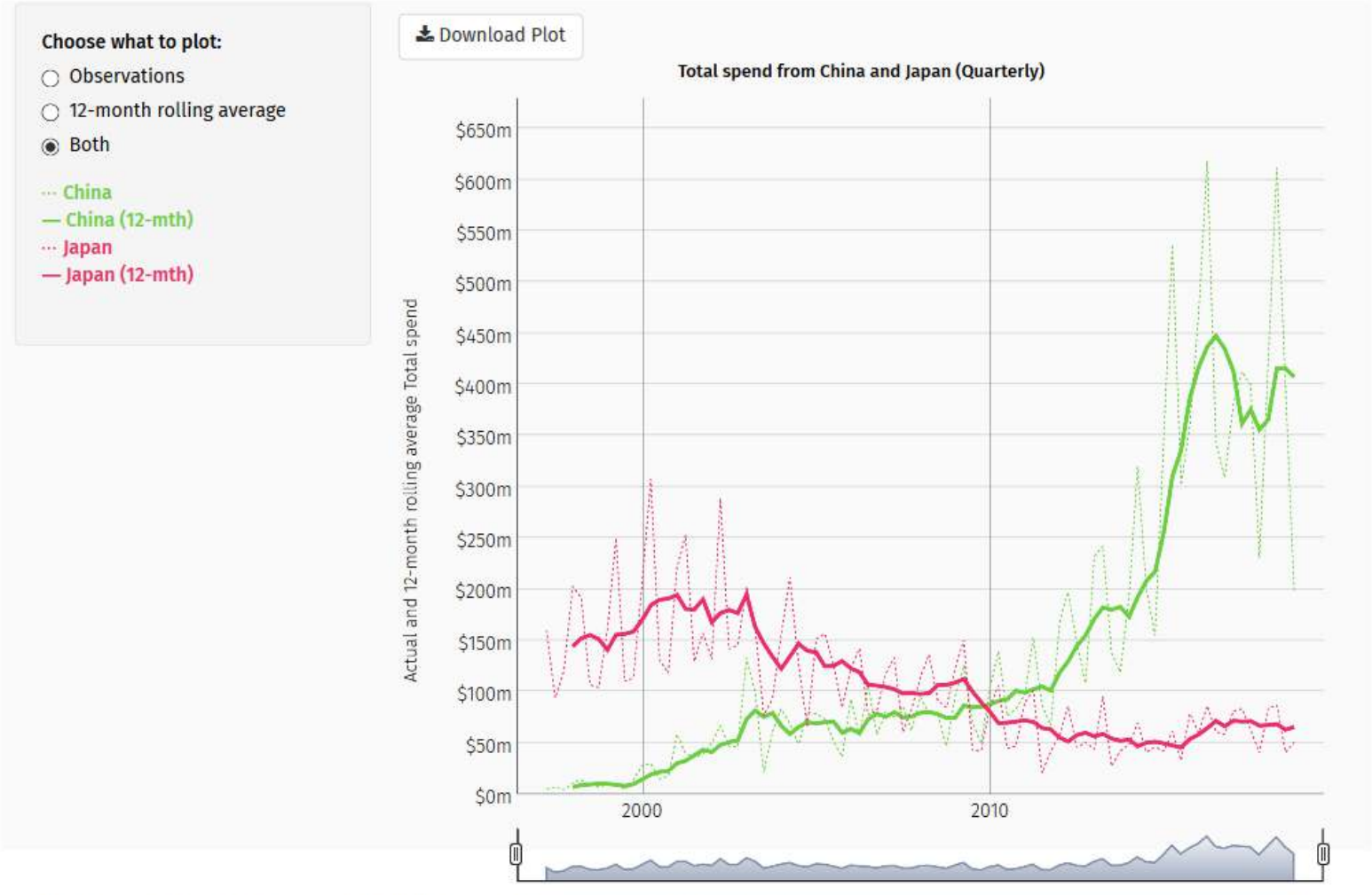
Memory game app



New Zealand tourism dashboard

Interactive Graph Example

This time series graph, plotting total visitor spend from China and Japan, can be used to practice the interactive graph functions described above.



These time series graphs are powered by dygraphs for R. More information is available here.

Shiny gallery: user showcases & demos

Industry Specific Shiny Apps

TOURISM DASHBOARD

User friendly portal to New Zealand tourism data (Code)

GENOME BROWSER

Visualize and explore an entire genome.

ER OPTIMIZATION

An app that models patient flow.

SUPPLY AND DEMAND

Forecast demand to plan resource allocation.

ECONOMIC DASHBOARD

Economic forecasting with macroeconomic indicators.

OMIM EXPLORER

Use phenotypes and genotypes to detect novel disease gene candidates (Video)

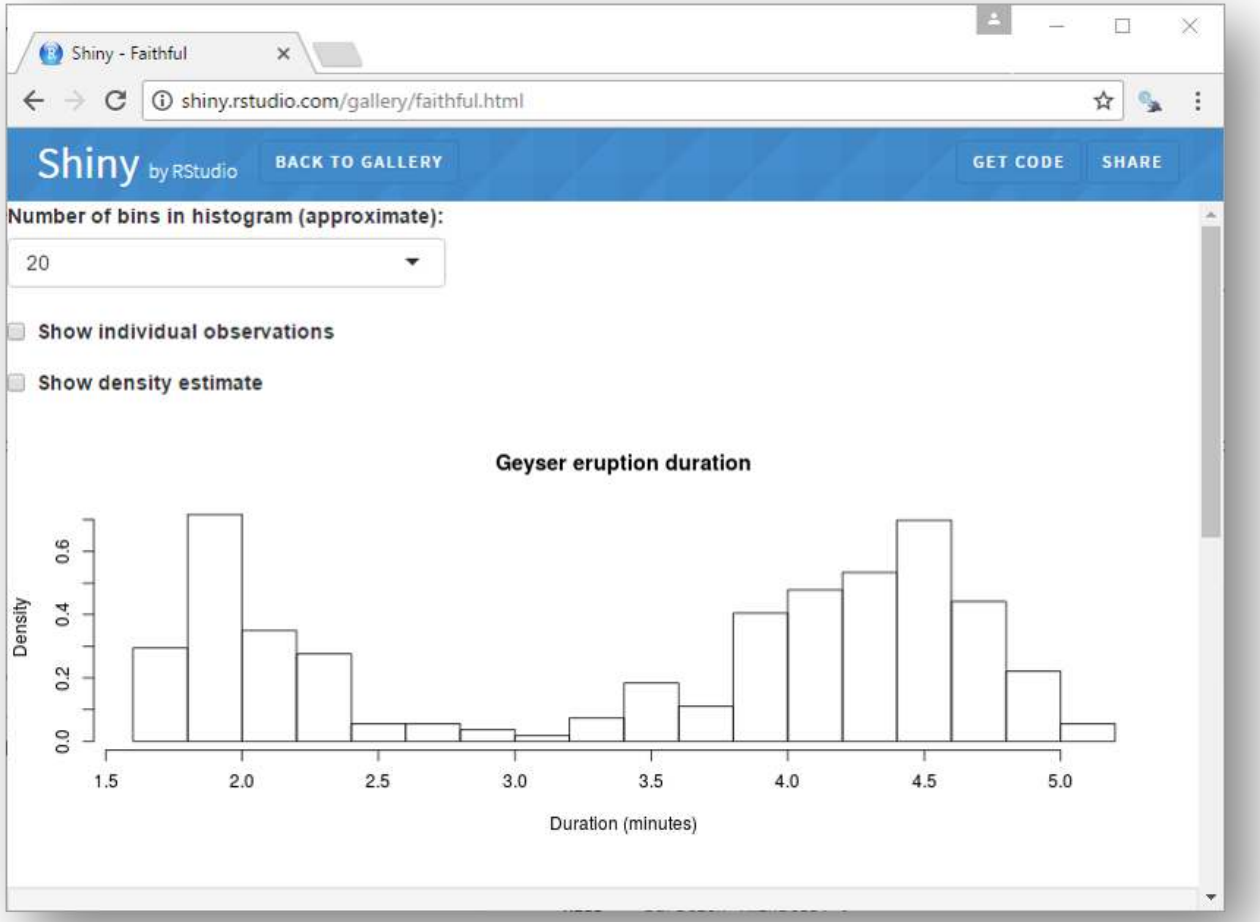
PHARMACOMETRICS

Apps that support pharmacometrics.

WILDLIFE SURVEY

Diagnostics for the Lake Erie fisheries.

"Hello, World!" app



What is Shiny?

- `R` package to create **interactive web apps** without deep knowledge of HTML, CSS and JavaScript
- based on **reactive programming**: manipulable **inputs** trigger automatic re-computation of **outputs**

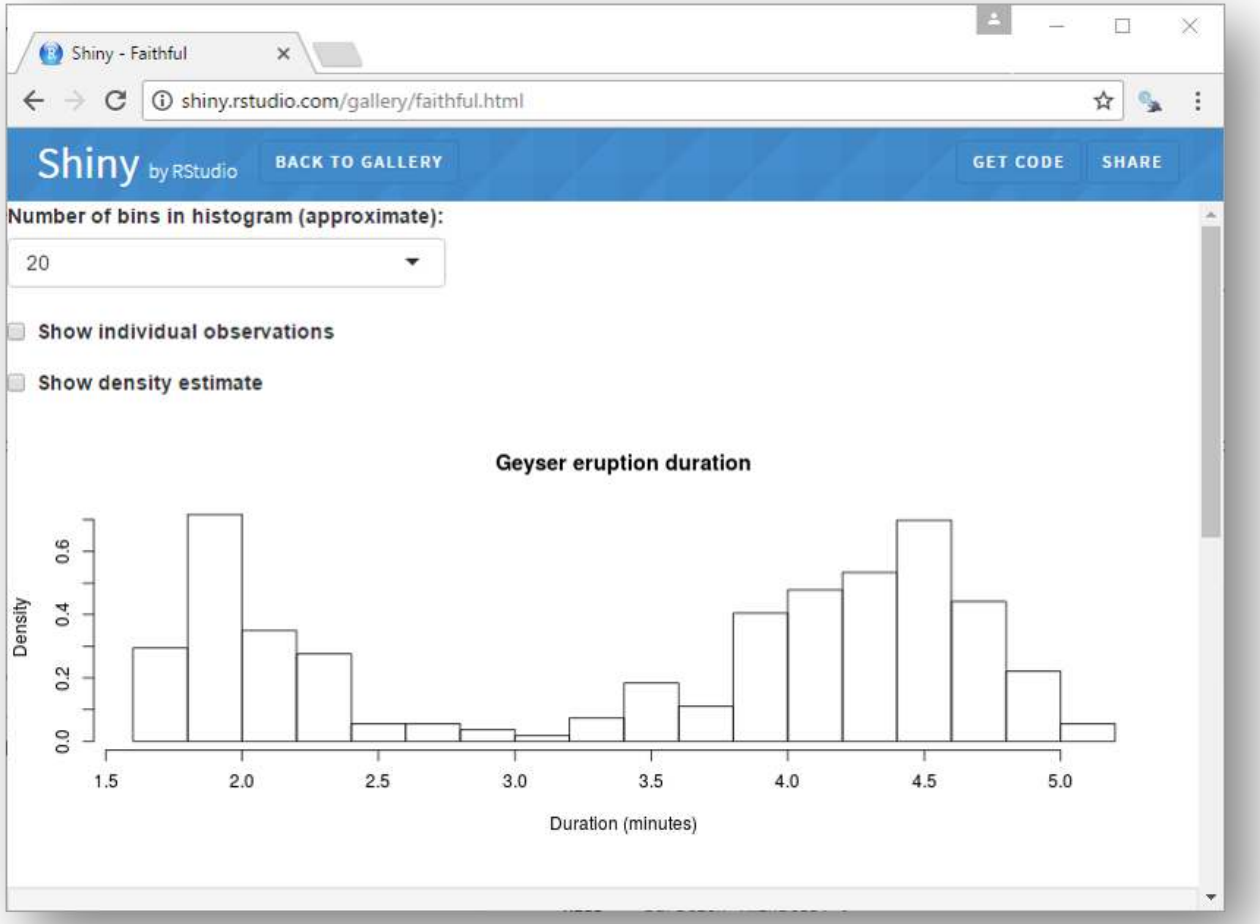
Application scenarios

- Dashboards that allow users to interactively toggle between compact at-a-glace summary and specific performance values → customizable to the level of detail the users need
- present complex models to a non-technical audience with interactive visualizations
- replace very long PDF documents or printed reports: allow users to drill down to sections they are most interested in

Minimal template of a Shiny app

```
library(shiny)
ui <- fluidPage() # controls layout & appearance (e.g. `bootstrapPage()`, `fluidPage()`, `fixedPage()`)
server <- function(input, output, session) {} # function that maps inputs to outputs
shinyApp(ui = ui, server = server) # runs the app
```

"Hello, World!" app



server.R | ui.R | ⤵ show below

```
bootstrapPage(

  selectInput(inputId = "n_breaks",
      label = "Number of bins in histogram (approximate):",
      choices = c(10, 20, 35, 50),
      selected = 20),

  checkboxInput(inputId = "individual_obs",
      label = strong("Show individual observations"),
      value = FALSE),

  checkboxInput(inputId = "density",
      label = strong("Show density estimate"),
      value = FALSE),

  plotOutput(outputId = "main_plot", height = "300px"),

  # Display this only if the density is shown
  conditionalPanel(condition = "input.density == true",
    sliderInput(inputId = "bw_adjust",
        label = "Bandwidth adjustment:",
        min = 0.2, max = 2, value = 1, step = 0.2)
  )

)
```

server.R | ui.R | ⤵ show below

```
function(input, output) {

  output$main_plot <- renderPlot({

    hist(faithful$eruptions,
      probability = TRUE,
      breaks = as.numeric(input$n_breaks),
      xlab = "Duration (minutes)",
      main = "Geyser eruption duration")

    if (input$individual_obs) {
      rug(faithful$eruptions)
    }

    if (input$density) {
      dens <- density(faithful$eruptions,
          adjust = input$bw_adjust)
      lines(dens, col = "blue")
    }

  })
}
```

ui.R

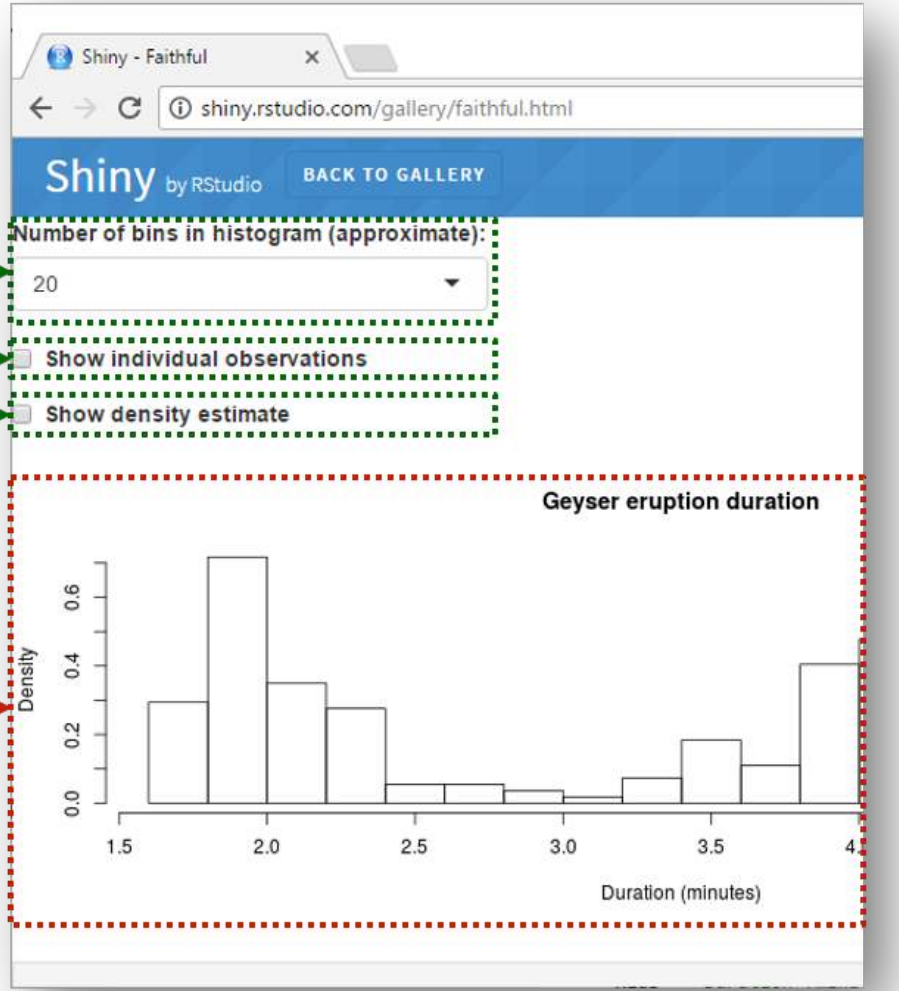
server.R | ui.R | show below

```
bootstrapPage(
```

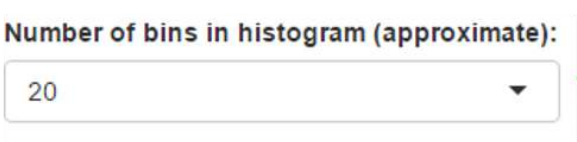
*Input()

```
  selectInput(inputId = "n_breaks",
      label = "Number of bins in histogram (approximate):",
      choices = c(10, 20, 35, 50),
      selected = 20),

  checkboxInput(inputId = "individual_obs",
      label = strong("Show individual observations"),
      value = FALSE),

  checkboxInput(inputId = "density",
      label = strong("Show density estimate"),
      value = FALSE),

  plotOutput(outputId = "main_plot", height = "300px"),

  # Display this only if the density is s
```

*Output()



Example input: dropdown list



Rendered HTML

```
selectInput(inputId = "n_breaks",
            label = "Number of bins in
                     histogram (approximate):",
            choices = c(10, 20, 35, 50),
            selected = 20)
```

```
<div class="form-group shiny-input-container">
    <label class="control-label" for="n_breaks">
          Number of bins in histogram (approximate):
    </label>
<div>
<select id="n_breaks">
     <option value="10">10</option>
     <option value="20" selected>20</option>
     <option value="35">35</option>
     <option value="50">50</option>
</select>
<script type="application/json" data-for="n_breaks"
     data-nonempty="">{}</script>
</div></div>
```

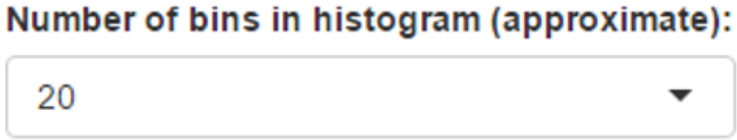
R Code

```
server.R   ui.R                                    show below

bootstrapPage(

  selectInput(inputId = "n_breaks",
      label = "Number of bins in histogram (approximate):",
      choices = c(10, 20, 35, 50),
      selected = 20),

  checkboxInput(inputId = "individual_obs",
      label = strong("Show individual observations"),
      value = FALSE),

  checkboxInput(inputId = "density",
      label = strong("Show density estimate"),
      value = FALSE),

  plotOutput(outputId = "main_plot", height = "300px"),

  # Display this only if the density is shown
  conditionalPanel(condition = "input.density == true",
    sliderInput(inputId = "bw_adjust",
        label = "Bandwidth adjustment:",
        min = 0.2, max = 2, value = 1, step = 0.2)
  )
```

Generated HTML Code

Example input: dropdown list

```
selectInput(inputId = "n_breaks",
            label = "Number of bins in
                     histogram (approximate):",
            choices = c(10, 20, 35, 50),
            selected = 20)
```



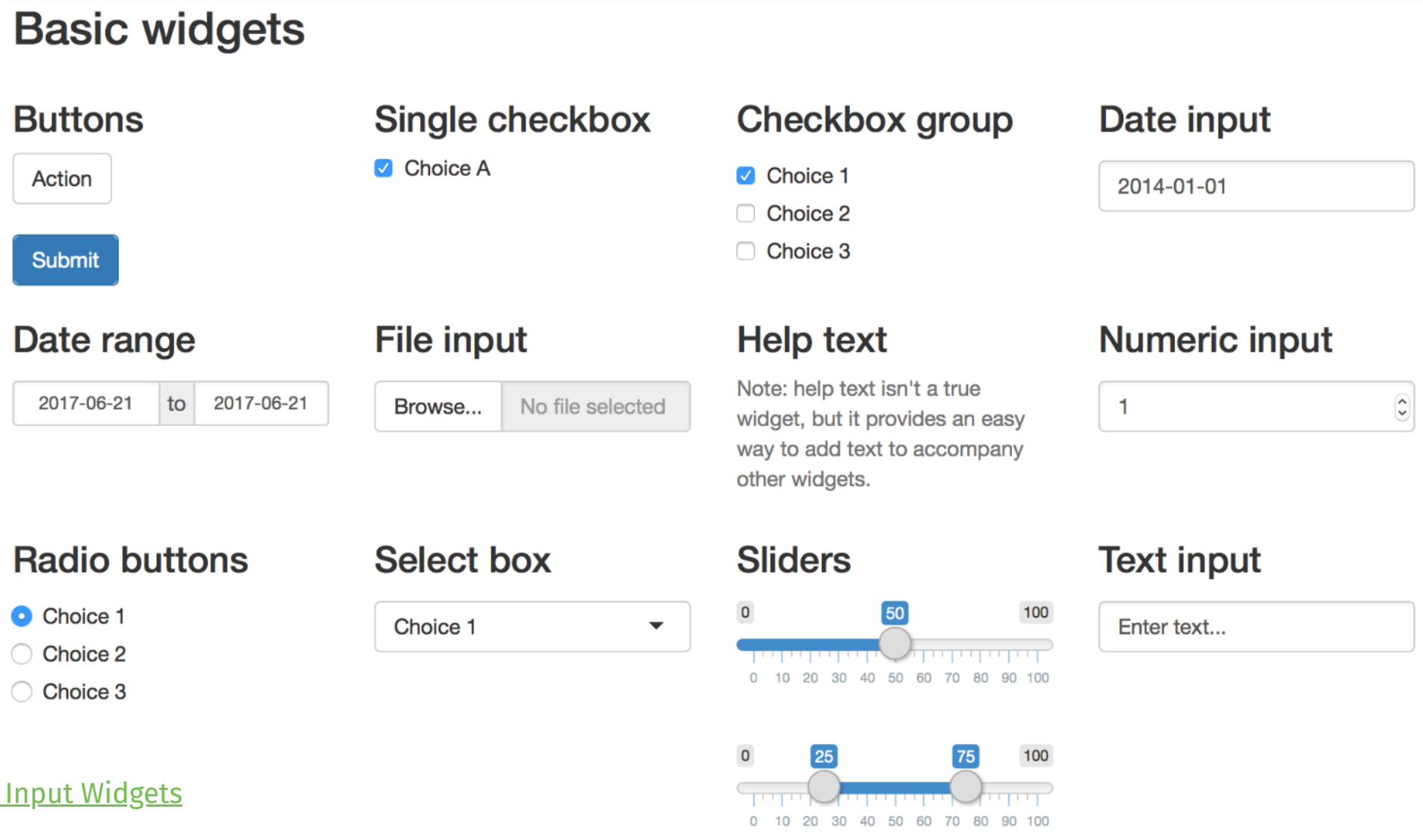
Mandatory arguments of input widgets:

- `inputId`: unique identifier
- `label`: descriptive label

Input-specific arguments (see `?selectInput`):

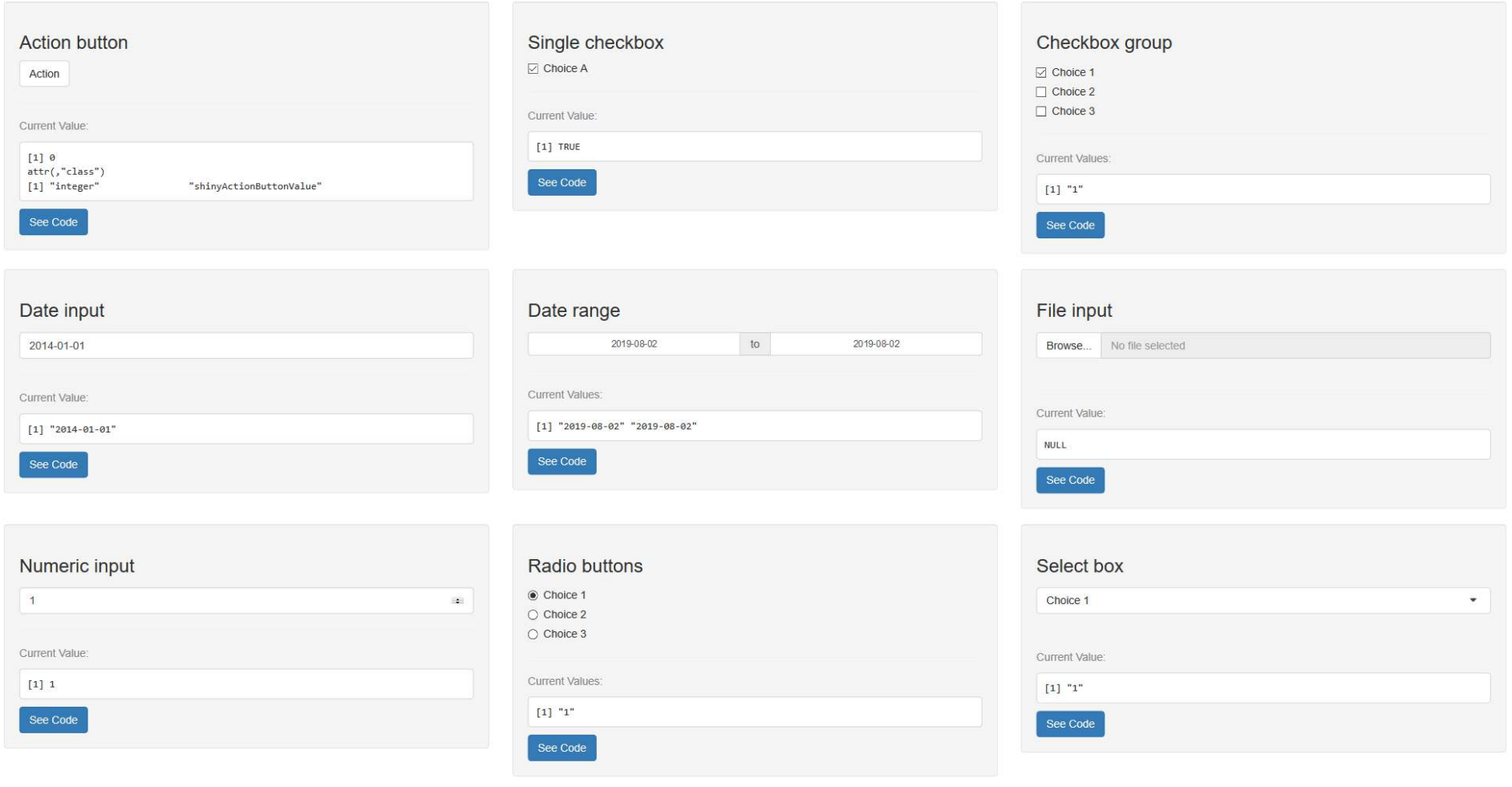
- `choices`: list of values
- `selected`: selected value at the start
- `multiple`: whether the selection of multiple values is allowed
- ...

Input widgets



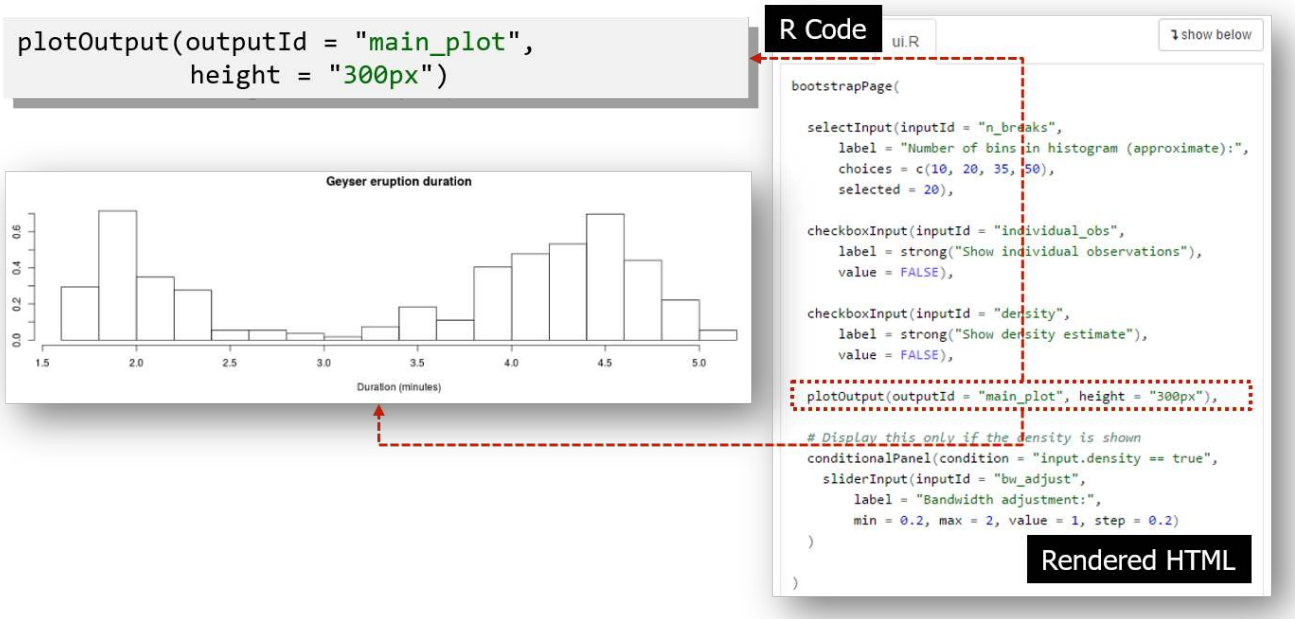
Shiny Input Widgets

Inputs have values...



Widget gallery

Outputs



Mandatory argument:

- `outputId`: unique identifier

output-specific arguments (see `?plotOutput`):

- `width`, `height`: width and height as CSS command
- ...

```
textOutput() # text
htmlOutput() # raw HTML
imageOutput() # image
plotOutput() # plot
tableOutput() # table
uiOutput() # Shiny UI element
```

server.R

Access current value of an input in server.R

Use an output from ui.R

server.R | ui.R | ⤵ show below

```
bootstrapPage(

  selectInput(inputId = "n_breaks",
      label = "Number of bins in histogram (approximate):",
      choices = c(10, 20, 35, 50),
      selected = 20),

  checkboxInput(inputId = "individual_obs",
      label = strong("Show individual observations"),
      value = FALSE),

  checkboxInput(inputId = "density",
      label = strong("Show density estimate"),
      value = FALSE),

  plotOutput(outputId = "main_plot", height = "300px"),

  # Display this only if the density is shown
  conditionalPanel(condition = "input.density == true",
    sliderInput(inputId = "bw_adjust",
        label = "Bandwidth adjustment:",
        min = 0.2, max = 2, value = 1, step = 0.2)
  )

)
```

server.R | ui.R | ⤵ show below

```
function(input, output) {

  output$main_plot <- renderPlot({

    hist(faithful$eruptions,
      probability = TRUE,
      breaks = as.numeric(input$n_breaks),
      xlab = "Duration (minutes)",
      main = "Geyser eruption duration")

    if (input$individual_obs) {
      rug(faithful$eruptions)
    }

    if (input$density) {
      dens <- density(faithful$eruptions,
          adjust = input$bw_adjust)
      lines(dens, col = "blue")
    }

  })
}
```

Render functions

Output objects are created within `render*()` functions.

server.R | ui.R | show below

```
function(input, output) {

  output$main_plot <- renderPlot({

    hist(faithful$eruptions,
      probability = TRUE,
      breaks = as.numeric(input$n_breaks),
      xlab = "Duration (minutes)",
      main = "Geyser eruption duration")

    if (input$individual_obs) {
      rug(faithful$eruptions)
    }

    if (input$density) {
      dens <- density(faithful$eruptions,
          adjust = input$bw_adjust)
      lines(dens, col = "blue")
    }

  })
}
```

Example:

`renderPlot(boxplot(iris$Sepal.Length))`

```
renderText()
renderPrint()
renderImage()
renderPlot()
renderTable()
renderUI()
```

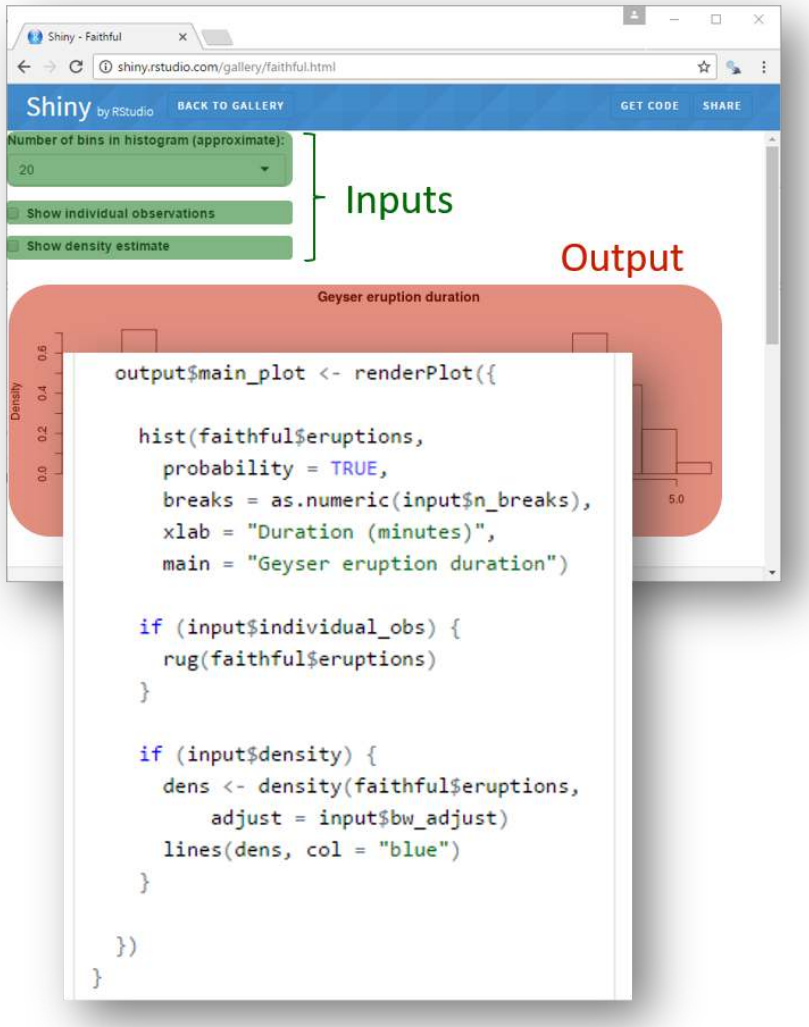
Reactivity

Inputs & Outputs

Reactive Programming: when an input changes, all related outputs are automatically updated.

Compare this to the normal behavior of `R`:

```
x <- 1
y <- x + 2
x <- 9
# What value does y have? 3 or 11?
```

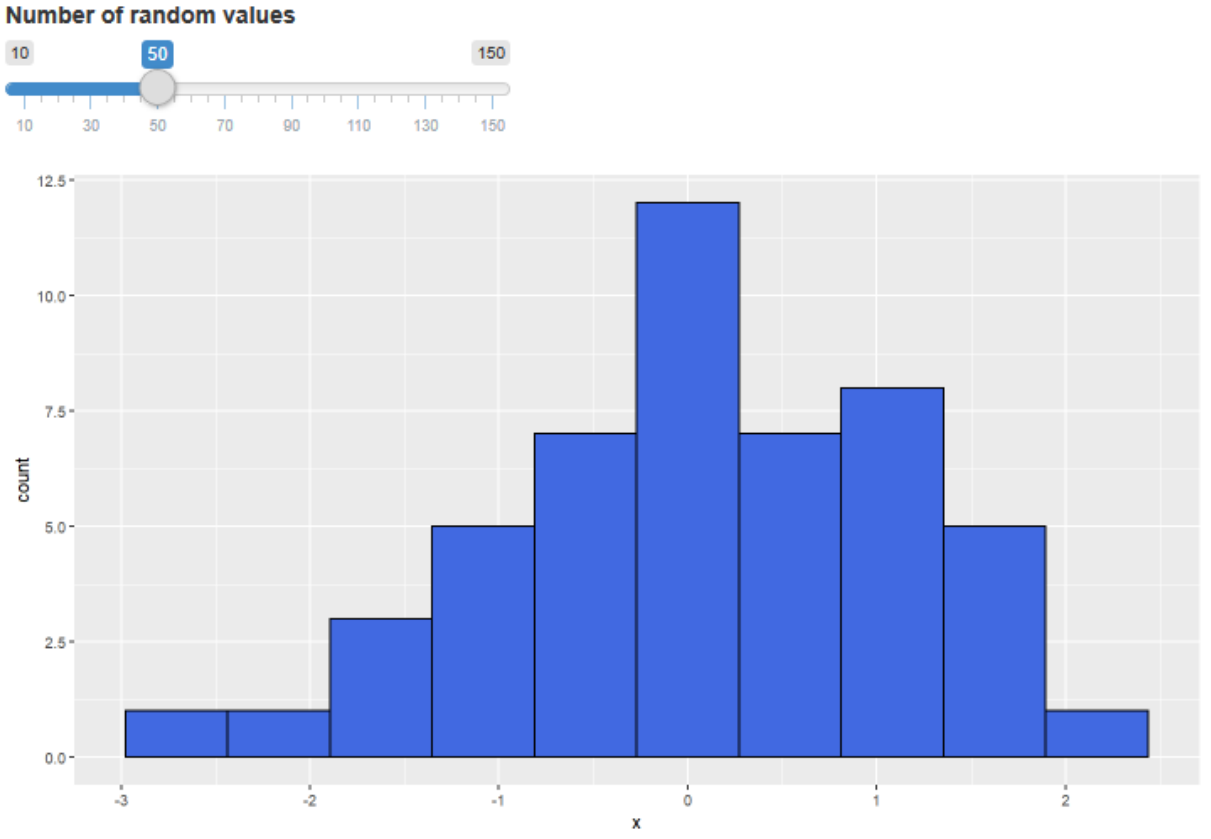


Recap: basic structure of a Shiny app

- **ui** component and **server** component
- **ui** contains **reactive inputs** (dropdown lists, radio buttons, ...) and **reactive outputs** (charts, tables, ...)
- reactive inputs are placed into the ui using **input functions** (`selectInput()`, `radioButtons()`, ...)
- reactive output containers are placed into the ui using **output functions** (`plotOutput()`, `tableOutput()`, ...)
- **server** contains code to create outputs, whose re-computation is triggered by one of the associated inputs
- output values (plots, tables) are generated within **render functions** (`renderPlot()`, `render(Table)`, ...)

Code modularization

Often, we need intermediate results (computed from reactive inputs) for multiple render functions.




```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-2.57546 -0.50136  0.18432  0.09257  0.93395  2.29736
```

Code modularization

```
library(shiny)
library(ggplot2)
library(dplyr)

ui <- fluidPage(
  sliderInput(inputId = "slider", label = "Number of random values",
              min = 10, max = 150, step = 10, value = 50),
  plotOutput(outputId = "plot"),
  verbatimTextOutput(outputId = "summary")
)

server <- function(input, output) {
  output$plot <- renderPlot({
    x <- tibble(x = rnorm(input$slider))
    ggplot(x, aes(x)) +
      geom_histogram(color = "black", fill = "royalblue", bins = 10)
  })
  output$summary <- renderPrint({
    x <- rnorm(input$slider)
    summary(x)
  })
}

shinyApp(ui, server)
```

🤔 *Which 2 problems does this code have?*

→ Histogram and summary console output do not show the same data.
→ **Code duplication** leads to problems in the long run. For example, if `runif()` shall be used instead of `rnorm()`, we would have to change the code at 2 places.

⟶ Solution: we create a **reactive object** for random values which can be used by both render functions.

Code modularization

```
library(shiny)
library(ggplot2)
library(tibble)

ui <- fluidPage(
  sliderInput(inputId = "slider", label = "Number of random values",
              min = 10, max = 150, step = 10, value = 50),
  plotOutput(outputId = "plot"),
  verbatimTextOutput(outputId = "summary")
)

server <- function(input, output) {
  values <- rnorm(input$slider)

  output$plot <- renderPlot({
    ggplot(tibble(x = values), aes(x)) +
      geom_histogram(color = "black", fill = "royalblue", bins = 10)
  })

  output$summary <- renderPrint({
    summary(values)
  })
}

shinyApp(ui, server)
```

```
Error in .getReactiveEnvironment()$currentContext() : Operation not allowed without an active reactive context. (You tried to do something that can only be done from inside a reactive expression or observer.)
```

Code modularization

→ Enclose with `reactive()`: `values <- reactive(rnorm(input$slider))`

We can access reactive values in reactive functions using `values()` (mind the parentheses):

```
library(shiny)
library(ggplot2)
library(tibble)

ui <- fluidPage(
  sliderInput(inputId = "slider", label = "Number of random values",
              min = 10, max = 150, step = 10, value = 50),
  plotOutput(outputId = "plot"),
  verbatimTextOutput(outputId = "summary")
)

server <- function(input, output) {
  values <- reactive({rnorm(input$slider)})

  output$plot <- renderPlot({
    ggplot(tibble(x = values()), aes(x)) +
      geom_histogram(color = "black", fill = "royalblue", bins = 10)
  })

  output$summary <- renderPrint({
    summary(values())
  })
}

shinyApp(ui, server)
```

Prevent re-computations

Prevent re-computations

```
library(shiny)
library(ggplot2)
library(tibble)

ui <- fluidPage(
  sliderInput(inputId = "slider", label = "Number of random values",
              min = 10, max = 150, step = 10, value = 50),
  textInput(inputId = "text", label = "histogram title", value = "fancy title"),
  plotOutput(outputId = "plot")
)

server <- function(input, output) {
  output$plot <- renderPlot({
    ggplot(tibble(x = rnorm(input$slider)), aes(x)) +
      geom_histogram(color = "black", fill = "royalblue", bins = 10) +
      labs(title = input$text)
  })
}

shinyApp(ui, server)
```

Prevent re-computations

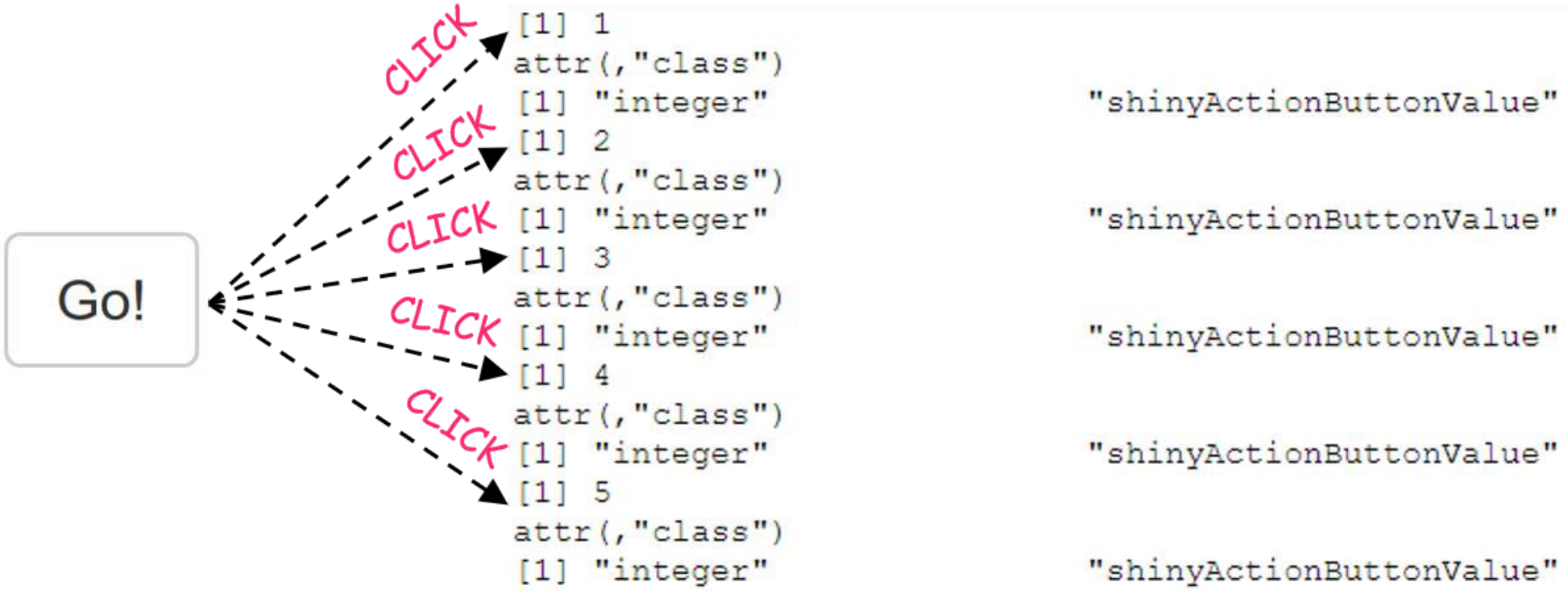
→ Enclose with `isolate()`: `isolate({input$title})` returns the result as non-reactive value.

```
library(shiny)
library(ggplot2)
library(tibble)

ui <- fluidPage(
  sliderInput(inputId = "slider", label = "Number of random values",
              min = 10, max = 150, step = 10, value = 50),
  textInput(inputId = "text", label = "histogram title", value = "fancy title"),
  plotOutput(outputId = "plot")
)

server <- function(input, output) {
  output$plot <- renderPlot({
    ggplot(tibble(x = rnorm(input$slider)), aes(x)) +
      geom_histogram(color = "black", fill = "royalblue", bins = 10) +
      labs(title = isolate({input$text}))
  })
}

shinyApp(ui, server)
```

The histogram will only be re-rendered if the value of the slider changes.

Trigger computations with reactive values

Example: action button



Trigger computations with reactive values

```
library(shiny)

ui <- fluidPage(
  actionButton(inputId = "button", label = "Go!")
)

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

}

shinyApp(ui, server)
```

→ ... with `observeEvent(<input$inputId>, <CODE>)`

Only if `<input$inputId>` changes, the `<CODE>` will be run!

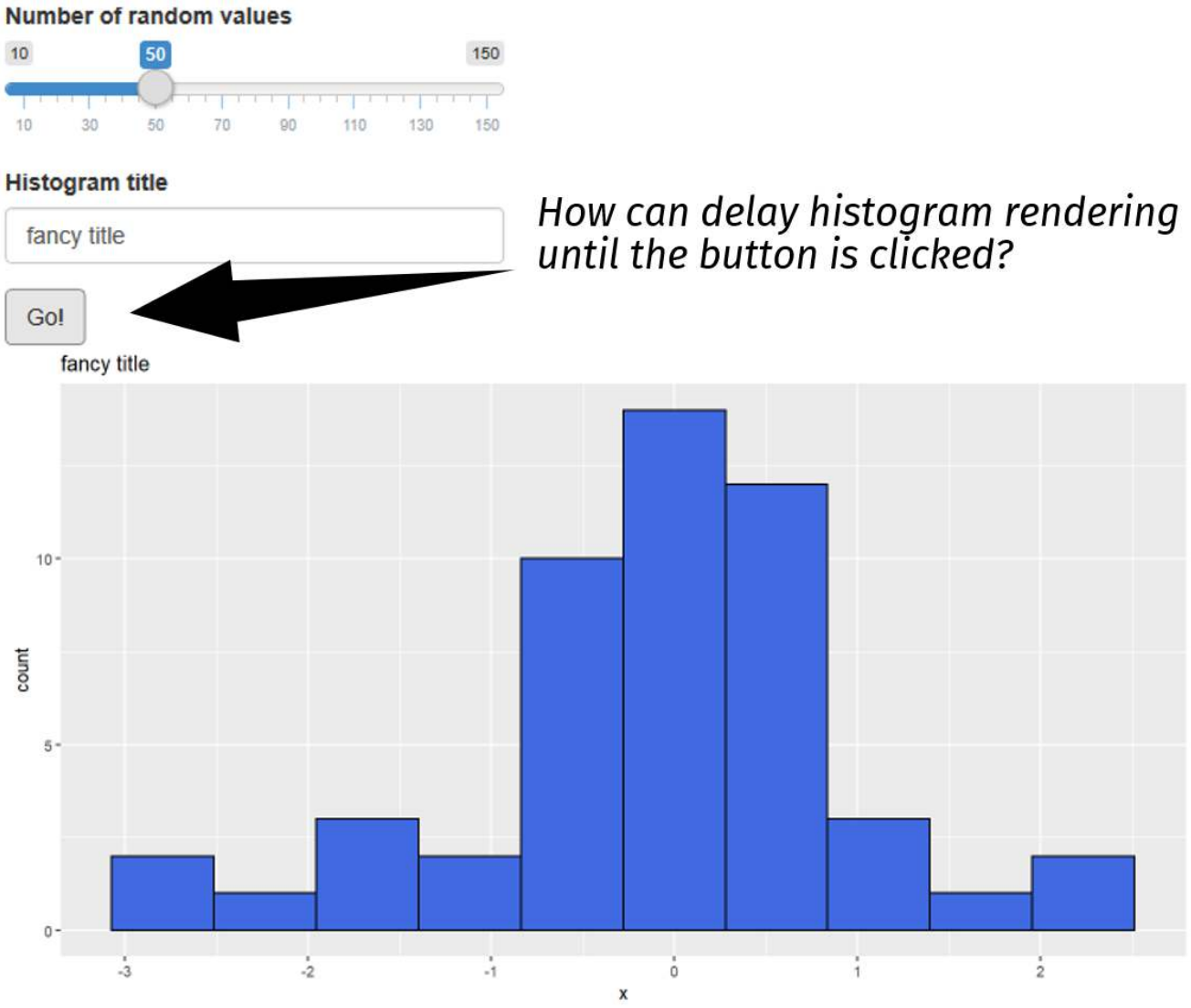
This means, if `<input$otherInputId>` (that is used within `<CODE>`) changes, re-computation of `<CODE>` will not be triggered.

Create an reactive observer

```
library(shiny)

ui <- fluidPage(
  actionButton(inputId = "button", label = "Go!")
)

server <- function(input, output) {
  observe(print(input$button))
}

shinyApp(ui, server)
```

In contrast to `reactive()`, `observe()` does not return a value. It is called for its *side effects*.

Delay re-computations



How can delay histogram rendering until the button is clicked?

Delay re-computations

```
library(shiny)
library(ggplot2)
library(tibble)

ui <- fluidPage(
  sliderInput(inputId = "slider", label = "Number of random values",
              min = 10, max = 150, step = 10, value = 50),
  textInput(inputId = "text", label = "histogram title", value = "fancy title"),
  actionButton(inputId = "button", label = "Go!"),
  plotOutput(outputId = "plot")
)

server <- function(input, output) {
  rp <- eventReactive(input$button, {
    ggplot(tibble(x = rnorm(input$slider)), aes(x)) +
      geom_histogram(color = "black", fill = "royalblue", bins = 10) +
      labs(title = input$text)
  })

  output$plot <- renderPlot(rp())
}
shinyApp(ui, server)
```

Template: `rv <- eventReactive(<input$inputId>, <CODE>)`

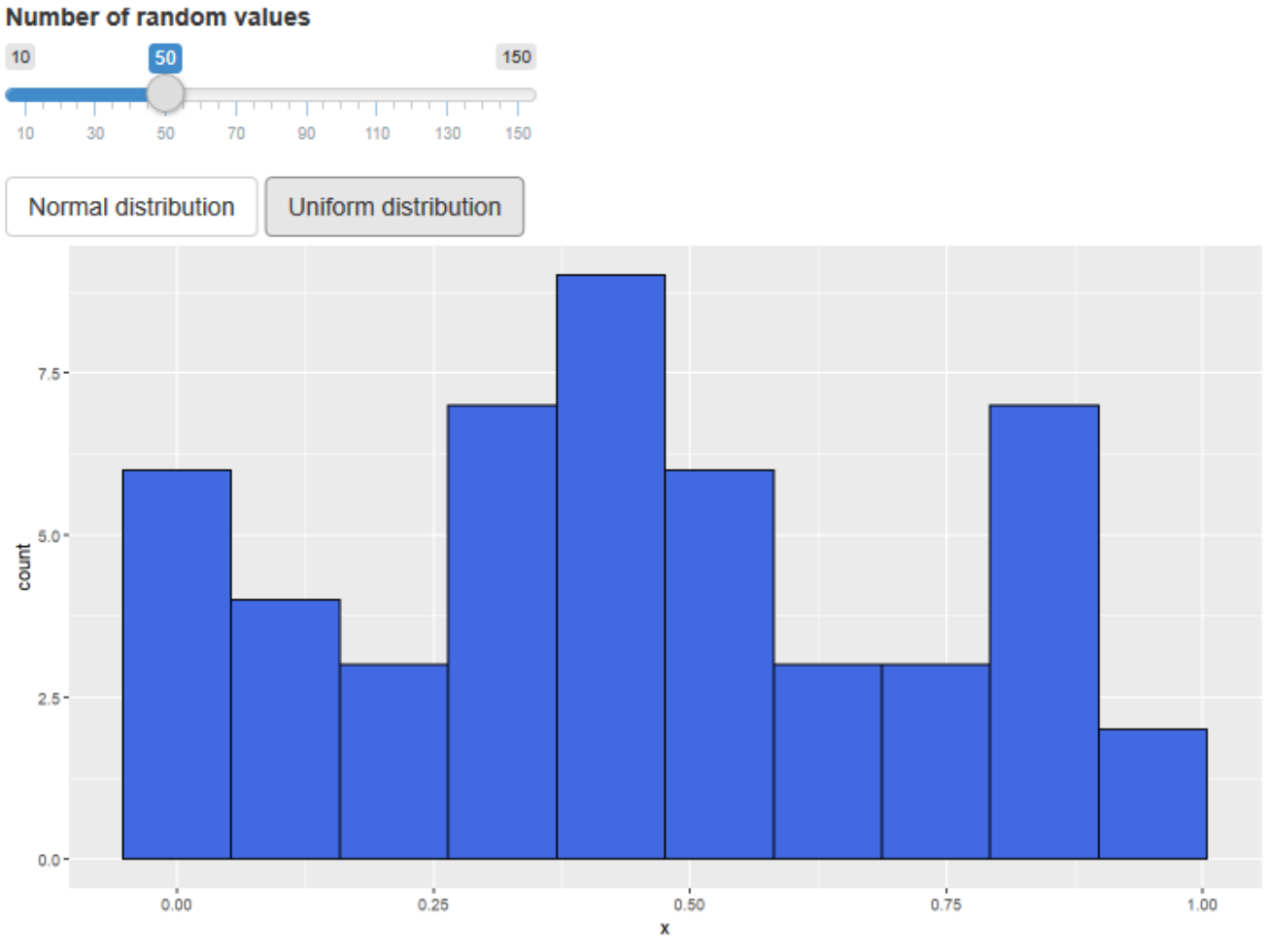
Only if `<input$inputId>` changes, `<CODE>` will be run and the output will be stored as `rv`.

Similar to `reactive()`, you can access reactive values created by `eventReactive()` using `rv()` (mind the parentheses).

Manipulable reactive values

Create a list of reactive values that can later be modified by input changes.

Template: `data <- reactiveValues(x1 = ..., x2 = ...)`



Manipulable reactive values

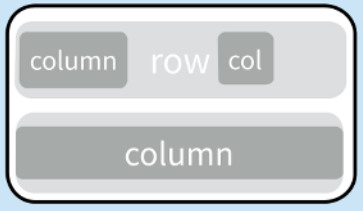
```
library(shiny)
library(ggplot2)
library(tibble)

ui <- fluidPage(
  sliderInput(inputId = "slider", label = "Number of random values",
              min = 10, max = 150, step = 10, value = 50),
  actionButton(inputId = "normal", label = "normal distribution"),
  actionButton(inputId = "unif", label = "uniform distribution"),
  plotOutput(outputId = "plot")
)

server <- function(input, output) {

  data <- reactiveValues(x = rnorm(100))

  observeEvent(input$normal, data$x <- rnorm(input$slider))
  observeEvent(input$unif, data$x <- runif(input$slider))

  output$plot <- renderPlot({
    ggplot(tibble(x = data$x), aes(x)) +
      geom_histogram(color = "black", fill = "royalblue", bins = 10)
  })
}

shinyApp(ui, server)
```

Recap: reactivity

- `render*()`: create **reactive outputs** that will be re-computed each time one of the values of the inputs they depend on is changed
- `reactive()`: create a **reactive expression**, that will be re-computed each time each time one of the values of the inputs they depend on is changed
- `isolate()`: **prevent the reactive binding** between an input and an output
- `observeEvent(x, ...)`: code will be executed only if the value of the reactive expression `x` changes. Does *not* return a value.
- `eventReactive(x, ...)`: code will be executed only if the value of a reactive expression `x` changes. Does return a value.
- `observe()`: code is executed in a reactive context. Does *not* return a value.
- `reactiveValues()`: creates a list of reactive values, which can be modified later.

Layouts

fluidRow()




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

sidebarLayout()

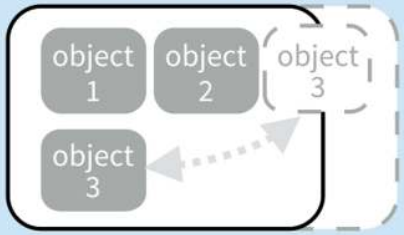

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

splitLayout()


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

Application layout guide

flowLayout()




```
ui <- fluidPage(
  flowLayout( # 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"))
```

HTML tags

- Shiny apps are websites.
- Websites are written in HTML.
- Shiny apps are written in `R`.
- HTML element can be inserted with `tags$<element>()` into the UI.

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

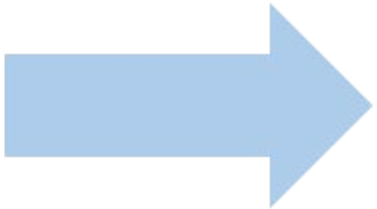
Customize your UI with HTML

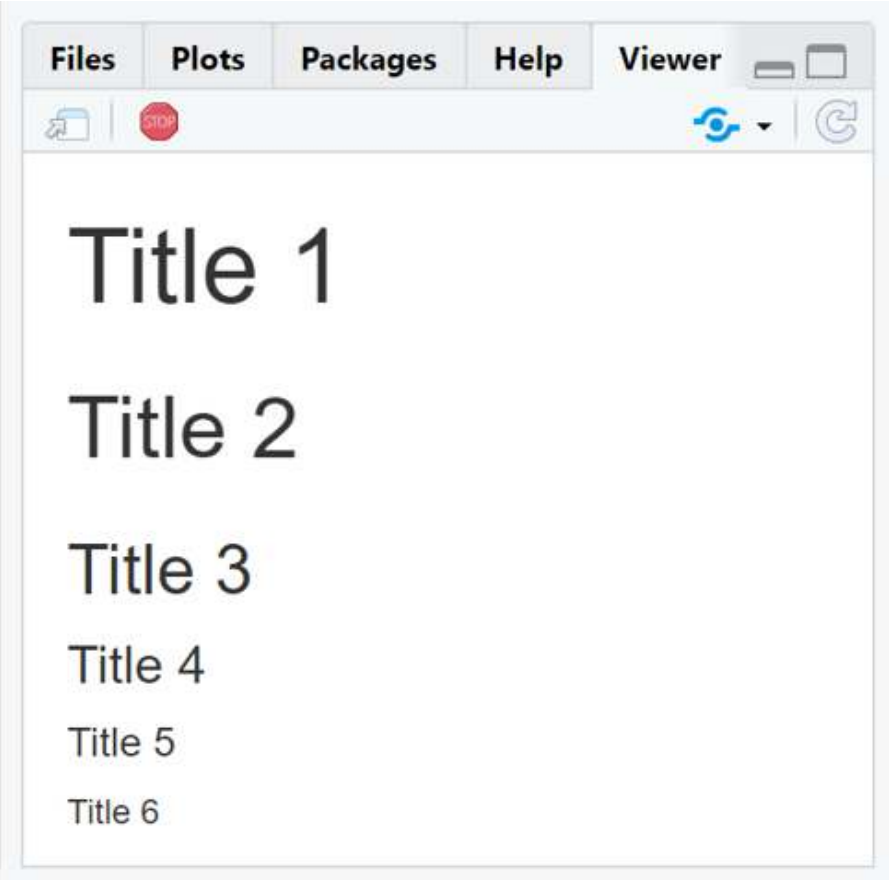
```
tags$h1("Title")
```
↔
```
<h1>Title</h1>
```

```
tags$strong("Boldface")
```
↔
```
<strong>Boldface</strong>
```

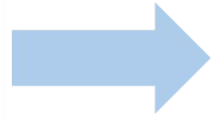
```
tags$a("URL", href = "https://www.google.com")
```
↔
```
<a href="https://www.google.com">URL</a>
```

```r
ui <- fluidPage(

    tags$h1("Title 1"),
    tags$h2("Title 2"),
    tags$h3("Title 3"),
    tags$h4("Title 4"),
    tags$h5("Title 5"),
    tags$h6("Title 6")
)
```

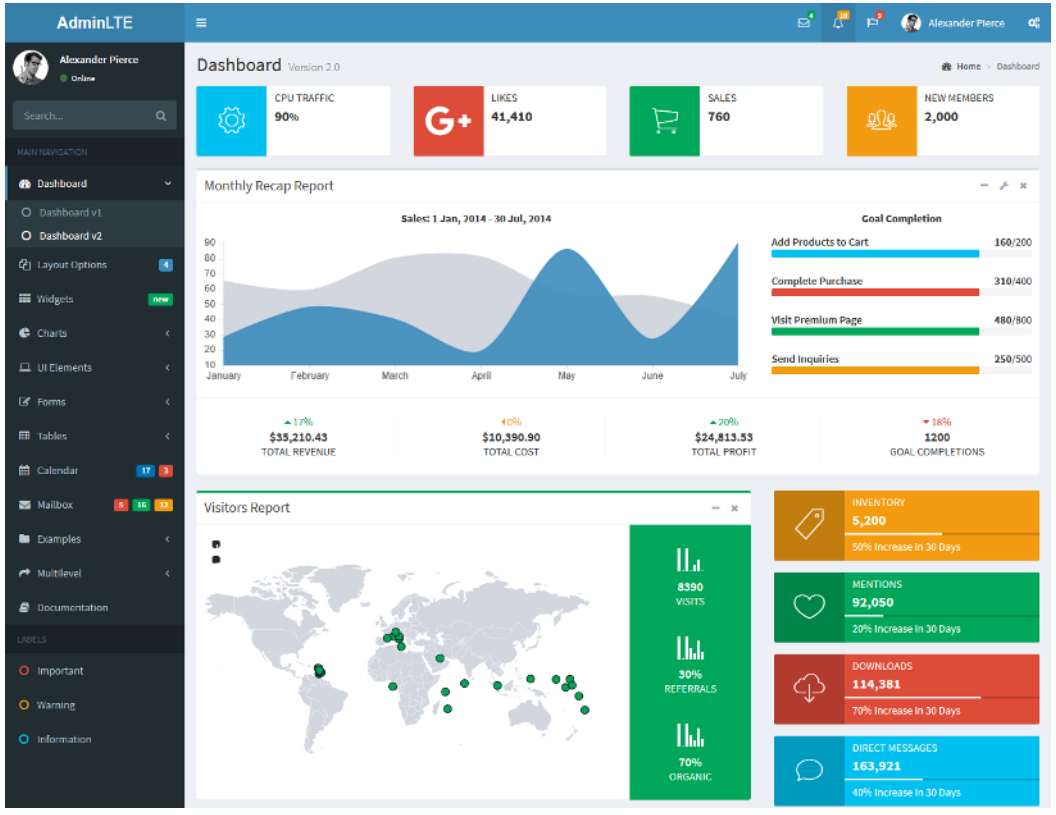




```r
ui <- fluidPage(

  tags$p("First paragraph."),
  tags$p(stringi::stri_rand_lipsum(1)),
  tags$a("Google main page", href = "https://www.google.com/"),
  tags$br(), # Manual line break
  tags$code("y <- 5*x + z^3"),
  tags$br(),
  tags$img(src = "https://upload.wikimedia.org/wikipedia/commons/thumb/0/0c/E_mei_zhi_wu_tu_zhi_%281945%29_%2821100899519%29.jpg/280px-E_mei_zhi_wu_tu_zhi_%281945%29_%2821100899519%29.jpg")
)
```



Dashboards with AdminLTE theme



Features: Notification Menus, Task Menus, infoBoxes, valueBoxes, ...

Getting Started Guide, Webinar Dynamic Dashboards with Shiny, Building Dashboards with Shiny Tutorial

App Deployment

	Pros	Cons
local	+ free + data remains local	- every user has to install R and shiny - maintenance
shinyapps.io	+ free up to 5 apps / account + no installation or hardware necessary + minimal maintenance + push button publishing	- tariffs with authentication 99+ USD/Monat - data in external cloud
Shiny Server	+ free (open source license) + data remains behind own firewall + authentication	- deployment and maintenance require more effort compared to shinyapps.io
RStudio Connect	+ easy "push button" deployment + data remain behind own firewall + authentication	- more expensive than shinyapps.io
electricShine (unofficial)	+ free + local offline app (.exe)	- requires more effort to set up compared to shinyapps.io - no official support - large file size (portable R installation)

What is the difference between RStudio Connect, Shiny Server Pro, and Shinyapps.io?


Thank you! Questions?
Photo courtesy of Stefan Berger