shiny basics

How can shiny help for simulations, courses and collaborations?

Pierre Gestraud - Benjamin Sadacca

11-06-19

State Of The R

What is shiny?

- R package developed by Winston Chang (Rstudio)
- initial github commit: 20/06/2012
- more than 3300 * on github
- package Description: Makes it incredibly **easy** to build **interactive** web applications with R. **Automatic** "reactive" binding between inputs and outputs and extensive **prebuilt** widgets make it possible to build **beautiful**, **responsive**, and **powerful** applications with **minimal effort**.

Get started

Rstudio > File > New File > Shiny Web App

Shiny app anatomy

Shiny app anatomy - structure

Two parts:

- **ui**: defines graphical interface
- **server**: performs all calculations
- ui and server can be in separate files (ui.R and server.R) or in single file (app.R)

and optionally:

- **global**: functions definitions, data… (global.R)

Shiny app anatomy - Inputs and Outputs

shiny interacts with the user through the **ui** with:

- **inputs**: set of parameters and data defined by the user with html widgets
    - in ui: `*Input`
- **outputs**: results of calculations (graphs, tables…)
    - in ui: `*Output`
    - in server: `render*`
- input and output function produce html code

```
textInput(inputId = "test", label = "Test")
```

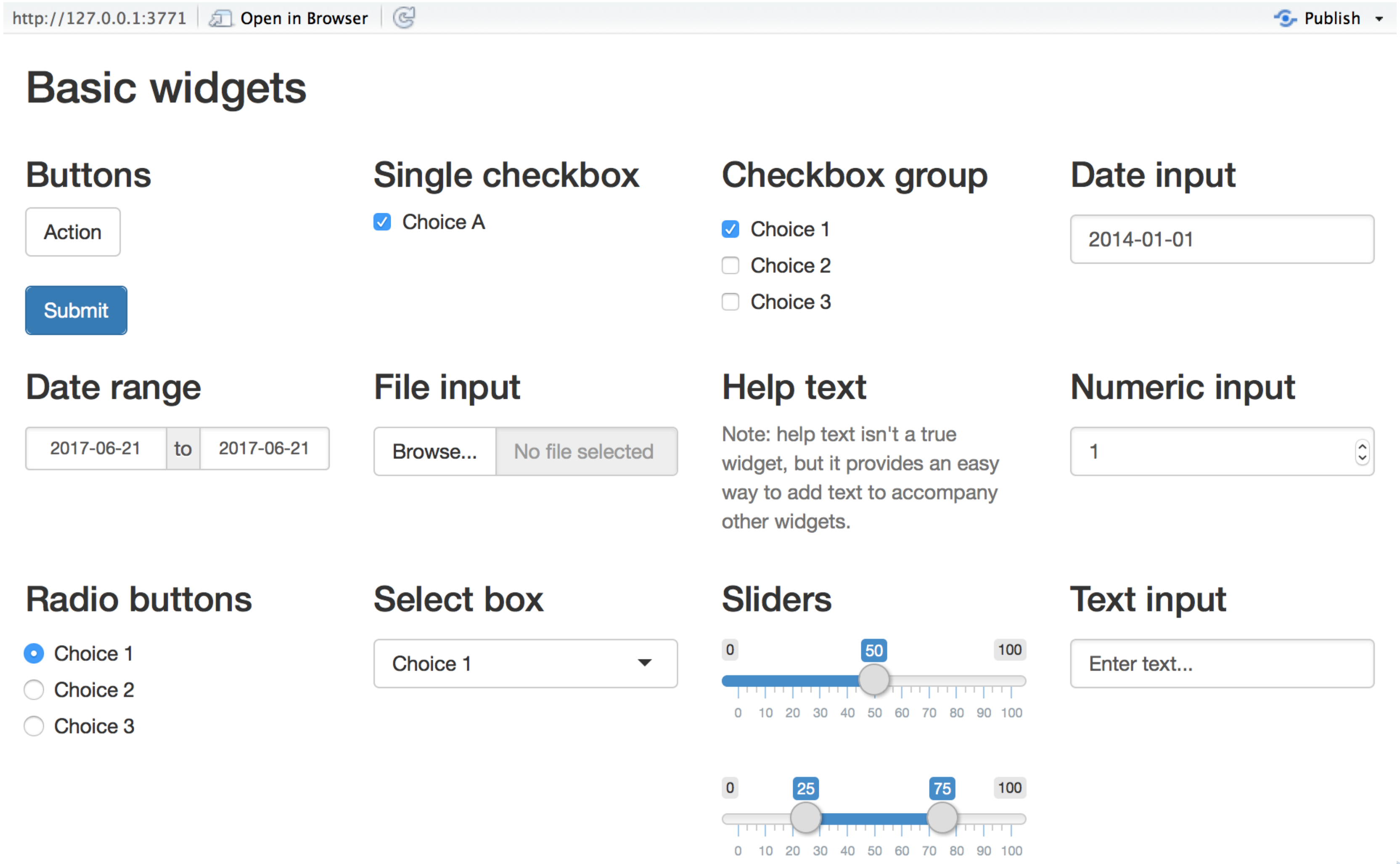
[1] "<div class=\"form-group shiny-input-container\">\n <label for=\"test\">Test</label>\n <input id=\"test\" type=\"text\" class=\"form-control\" value=\"\"/>\n</div>"

UI basics

UI - inputs

- inputs are widgets taking value from the user
- possible inputs:
  - `actionButton` Action Button
  - `checkboxGroupInput` A group of check boxes
  - `checkboxInput` A single check box
  - `dateInput` A calendar to aid date selection
  - `dateRangeInput` A pair of calendars for selecting a date range
  - `fileInput` A file upload control wizard
  - `helpText` Help text that can be added to an input form
  - `numericInput` A field to enter numbers
  - `radioButtons` A set of radio buttons
  - `selectInput` A box with choices to select from
  - `sliderInput` A slider bar
  - `submitButton` A submit button
  - `textInput` A field to enter text

UI - inputs



Basic widgets

Buttons

Single checkbox

Checkbox group

Date input

Date range

File input

Help text

Note: help text isn't a true widget, but it provides an easy way to add text to accompany other widgets.

Numeric input

Radio buttons

Select box

Sliders

Text input

State Of The R

UI - inputs structure

For each input we have:

- `inputId`: name of the input, used by the server
- `label`: name of the input displayed to the user
- other parameters depending on the input type

```r
ui <- fluidPage(
  ## some text
  textInput(inputId = "caption",
            label = "Caption",
            value = "Data Summary"),
  ## a slider
  sliderInput(inputId = "slide_val",
              label = "Value",
              min = 0, max = 10, value = 5
  )
)
```

UI - outputs

- outputs are html components that display results coming from server
- possible outputs:
  - `dataTableOutput` DataTable
  - `htmlOutput` raw HTML
  - `imageOutput` image
  - `plotOutput` plot
  - `tableOutput` table
  - `textOutput` text
  - `uiOutput` raw HTML
  - `verbatimTextOutput` text

UI - outputs

For each output we have:

- `outputId`: name of the output, used by the server
- other parameters (such as image size)

```
ui <- fluidPage(
  ## a plot
  plotOutput(outputId = "plot1",
             width = "80%"),
  ## a table
  tableOutput(outputId = "table")
  )
```

UI - layouts

- simple app:
  - fluidpage
    - sidebarPanel
    - titlePanel
    - mainPanel
      - fluidRow
      - columns
- shinydashboard for more complex app, very popular
- custom html/css
- https://rinterface.com/

https://shiny.rstudio.com/articles/layout-guide.html

UI - Customization

Shiny provides a list of functions named tags to produce HTML tag.

- section: `tags$h1()`
- div creaction: `tags$div()`
- bold text: `tags$b()`
- list: `tags$ol()` or `tags$ul()`
- paragraph: `tags$p()`
- …

https://shiny.rstudio.com/articles/tag-glossary.html

server

server

- the `server` is the computational part of the app

- `server` is a function

- communicates with `ui` through 2 lists:

    - `input`: get values from input widgets
    - `output`: set values for outputs

- `server` can also have a `session` component to get info on the user

server - get inputs

Retrieve inputs values with `input$...`

```r
ui <- fluidPage(
  textInput("caption", "Caption", "Data Summary"),
  verbatimTextOutput("value")
)
server <- function(input, output) {
  output$value <- renderText({
  input$caption
  })
}
```

server - set outputs

Set values to outputs with `output$...<-`

```r
ui <- fluidPage(
  textInput("caption", "Caption", "Data Summary"),
  verbatimTextOutput("value")
)
server <- function(input, output) {
  output$value <- renderText({
  input$caption
  })
}
```

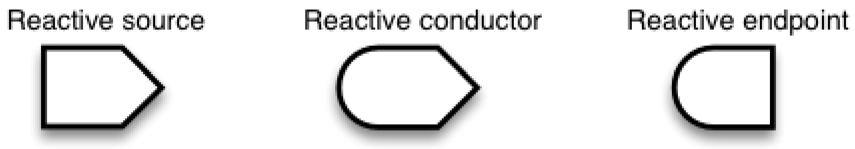
Reactivity

Reactivity

Reactivity is the feature of shiny that allows to have interactivity in the code: a piece of code is run in reaction on a change in the ui.

Three kind of reactive objects:

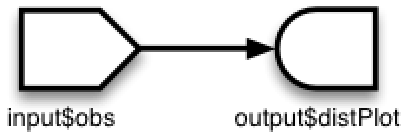
- reactive sources (inputs)
- reactive endpoints (outputs): `render*` functions
- reactive conductors (intermediate objects): `reactive`, `reactiveValues`



Reactivity

Simplest example, one input, one output, directly linked together:

```
ui <- fluidPage(
  numericInput("obs", label = "Obs number", value = 10),
  plotOutput("distPlot")
  )
server <- function(input, output) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })
}
```

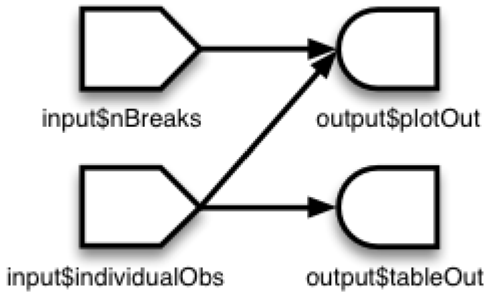


Reactivity

We can have multiple links between outputs and inputs:

```
server <- function(input, output) {
  output$plotOut <- renderPlot({
    hist(faithful$eruptions, breaks = as.numeric(input$nBreaks))
    if (input$individualObs) rug(faithful$eruptions)
  })

  output$tableOut <- renderTable({
    if (input$individualObs)
      faithful
    else  NULL
  })
}
```



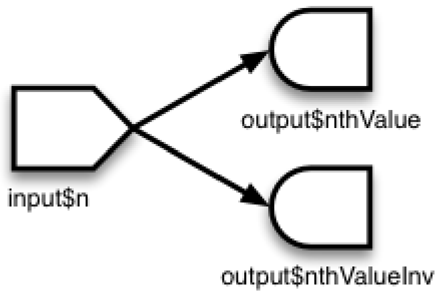
Reactivity

Fibonacci sequence example.

What's wrong with this app?

```r
# Calculate nth number in Fibonacci sequence
fib <- function(n) ifelse(n<3, 1, fib(n-1)+fib(n-2))

server <- function(input, output) {
  output$nthValue    <- renderText({ fib(as.numeric(input$n))})
  output$nthValueInv <- renderText({ 1 / fib(as.numeric(input$n))})
}
```



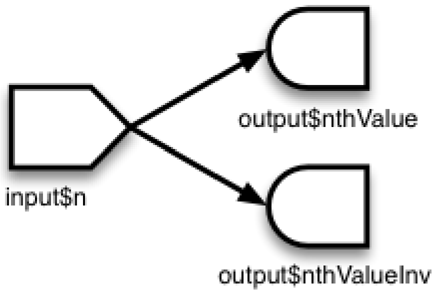
Reactivity

Fibonacci sequence example.

What's wrong with this app? We compute twice the Fibonacci sequence…

```r
# Calculate nth number in Fibonacci sequence
fib <- function(n) ifelse(n<3, 1, fib(n-1)+fib(n-2))

server <- function(input, output) {
  output$nthValue    <- renderText({ fib(as.numeric(input$n))})
  output$nthValueInv <- renderText({ 1 / fib(as.numeric(input$n))})
}
```



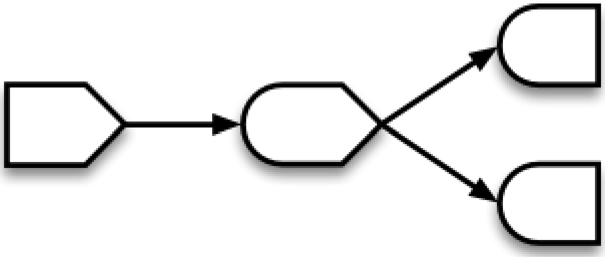
reactive

To avoid useless computation or use same objects in different expression we can store intermediate object inside `reactive()`

```r
fib <- function(n) ifelse(n<3, 1, fib(n-1)+fib(n-2))

server <- function(input, output) {
   currentFib         <- reactive({ fib(as.numeric(input$n)) })

  output$nthValue    <- renderText({ currentFib() })
  output$nthValueInv <- renderText({ 1 / currentFib() })
}
```



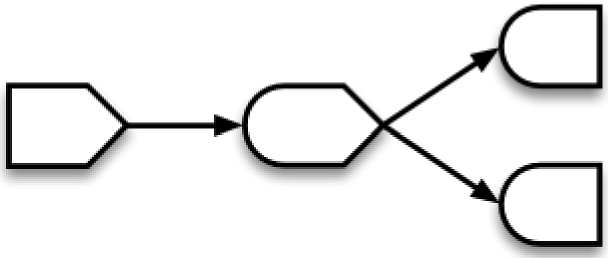
reactive

Use value from the reactive expression. Don't forget the `()`!

```r
fib <- function(n) ifelse(n<3, 1, fib(n-1)+fib(n-2))

server <- function(input, output) {
  currentFib         <- reactive({ fib(as.numeric(input$n)) })

  output$nthValue    <- renderText({ currentFib() })
  output$nthValueInv <- renderText({ 1 / currentFib() })
}
```



Reactivity in action

- In standard R code, the value of an object is updated when needed in an expression
- In shiny a mechanism is set that look for updates in the reactive tree
- Reactive expressions are lazy. Only needed expressions are re-evaluated (update of inputs **and** call from one of its dependency)

In practice, reactive expression are updated in reaction to user actions.

https://shiny.rstudio.com/articles/understanding-reactivity.html

Handling missing inputs

Handling missing inputs

What happens if we run this app?

```r
ui <- fluidPage(
  selectInput("datasetName", "Dataset", c("", "pressure", "cars")),
  tableOutput("table")
)

server <- function(input, output, session) {
  dataset <- reactive({
    get(input$datasetName, "package:datasets", inherits = FALSE)
  })

  output$table <- renderTable({
    head(dataset(), 10)
  })
}

shinyApp(ui, server)
```

Handling missing values - the old way

```r
ui <- fluidPage(
  selectInput("datasetName", "Dataset", c("", "pressure", "cars")),
  tableOutput("table")
)

server <- function(input, output, session) {
  dataset <- reactive({
      if (input$datasetName == "")
        return(NULL)
    get(input$datasetName, "package:datasets", inherits = FALSE)
  })

  output$table <- renderTable({
      if (is.null(dataset()))
       return(NULL)
    head(dataset(), 10)
  })
}
shinyApp(ui, server)
```

Handling missing values - the modern way

```r
ui <- fluidPage(
  selectInput("datasetName", "Dataset", c("", "pressure", "cars")),
  tableOutput("table")
)

server <- function(input, output, session) {
  dataset <- reactive({
    # Make sure requirements are met
    req(input$datasetName)
    get(input$datasetName, "package:datasets", inherits = FALSE)
  })

  output$table <- renderTable({
    head(dataset(), 10)
  })
}
```

`req` stops silently the execution of the callstack, preventing errors with missing objects.

See also `validate`/`need`

https://shiny.rstudio.com/articles/req.html

Interactive plots

Make plots interactive

shiny has built-in support for interacting with static plots generated by base graphics functions and ggplot2.

- Basic example with response to click

```
ui <- basicPage(
  plotOutput("plot1", click = "plot_click"),
  verbatimTextOutput("info")
)

server <- function(input, output) {
  output$plot1 <- renderPlot({
    plot(mtcars$wt, mtcars$mpg)
  })
  output$info <- renderText({
    paste0("x=", input$plot_click$x, "\ny=", input$plot_click$y)
  })
}
```

- The other types of interactions are **double-clicking**, **hovering**, and **brushing**. They can be enabled with the `dblclick`, `hover`, and `brush` options.

https://shiny.rstudio.com/articles/plot-interaction.html

Make plots interactive - ggplot2

```r
ui <- basicPage(
  plotOutput("plot1", click = "plot_click"),
  verbatimTextOutput("info")
)

server <- function(input, output) {
  output$plot1 <- renderPlot({
    ggplot(mtcars, aes(x = wt, y = mpg)) + geom_point()
  })
  output$info <- renderText({
    paste0("x=", input$plot_click$x, "\ny=", input$plot_click$y)
  })
}
shinyApp(ui, server)
```

Make plots interactive - selection in dataset

We can select rows of a dataset in response of an interaction on a plot

- `nearPoints()`: Uses the x and y value from the interaction data; to be used with `click`, `dblclick`, and `hover`.
- `brushedPoints()`: Uses the xmin, xmax, ymin, and ymax values from the interaction data; to be used with `brush`.

```r
ui <- basicPage(
  plotOutput("plot1", brush = "plot_brush", height = 250),
  verbatimTextOutput("info")
)

server <- function(input, output) {
  output$plot1 <- renderPlot({
    ggplot(mtcars, aes(x=wt, y=mpg)) + geom_point() +
      facet_grid(. ~ cyl) +
      theme_bw()
  })

  output$info <- renderPrint({
    brushedPoints(mtcars, input$plot_brush)
  })
}
shinyApp(ui, server)
```

Miscellaneous

actionButton

- actionButton and actionLink are widgets designed to react to a user click
- the value of an actionButton is not meaningful by itself but should be observed by `observeEvent()` or `eventReactive()`

https://shiny.rstudio.com/articles/action-buttons.html

actionButton

Use `observeEvent()` to trigger a command with an action button.

```
ui <- fluidPage(
  tags$head(tags$script(src = "message-handler.js")),
  actionButton("do", "Click Me")
)

server <- function(input, output, session) {
   observeEvent(input$do, {
    session$sendCustomMessage(type = 'testmessage',
      message = 'Thank you for clicking')
  })
}
shinyApp(ui, server)
```

https://shiny.rstudio.com/articles/action-buttons.html

actionButton

```r
ui <- fluidPage(
  actionButton("go", "Go"),
  numericInput("n", "n", 50),
  plotOutput("plot")
)

server <- function(input, output) {
  randomVals <- eventReactive(input$go, {
    runif(input$n)
  })

  output$plot <- renderPlot({
    hist(randomVals())
  })
}

shinyApp(ui, server)
```

https://shiny.rstudio.com/articles/action-buttons.html

Progress bar

```r
server <- function(input, output) {
  output$plot <- renderPlot({
    input$goPlot # Re-run when button is clicked
    dat <- data.frame(x = numeric(0), y = numeric(0))
    n <- 10
    withProgress(message = 'Making plot', value = 0, {
      for (i in seq_len(n)) {
        dat <- rbind(dat, data.frame(x = rnorm(1), y = rnorm(1)))
        # Increment the progress bar, and update the detail text.
        incProgress(1/10, detail = paste("Doing part", i))
        # Pause for 0.2 seconds to simulate a long computation.
        Sys.sleep(0.2)
      }
    })
    plot(dat$x, dat$y)
  })
}
ui <- basicPage(
  plotOutput('plot', width = "300px", height = "300px"),
  actionButton('goPlot', 'Go plot')
)
shinyApp(ui = ui, server = server)
```

Dynamic UI

Four ways to make UI dynamic in response to user actions

- `conditionalPanel` function which is used in `ui.R` and wraps a set of UI elements that need to be dynamically shown/hidden.
- `renderUI` function which is used in `server.R` in conjunction with the `uiOutput` function in ui.R, lets you generate calls to UI functions and make the results appear in a predetermined place in the UI.
- `insertUI` and `removeUI` functions, which are used in `server.R` and allow you to add and remove arbitrary chunks of UI code.
- JavaScript to modify the webpage directly.

Dynamic UI - conditionnalPanel

```r
ui <- basicPage(
  checkboxInput("smooth", "Smooth", value = FALSE),
  conditionalPanel(
    condition = "input.smooth == true",
    selectInput("smoothMethod", "Method",
                list("lm", "glm", "gam", "loess", "rlm"))
))
server <- function(input, output){NULL}
shinyApp(ui, server)
```

Dynamic UI - renderUI

```r
ui <- basicPage(
  selectInput(inputId = "dataset", label = "Dataset",
              choices = c("cars", "faithful", "iris")),
  uiOutput("col_select")
  )
server <- function(input, output){
  dataset <- reactive(get(input$dataset, "package:datasets"))
  output$col_select <- renderUI({
    values <- colnames(dataset())
    selectInput("col", "Column", choices = values)
    })
}
shinyApp(ui, server)
```

Going further with modules

- Fundamental units of abstraction of `R` are functions
- A shiny app can be modularised using functions but since inputs and outputs id are globally shared we need to have a mechanism to handle name conflicts
- A `module` is a piece of shiny app which can be used in several places in one app or in different app
    - we can see a module as a function + a namespace
    - a module is composed of 2 parts (ui and server)

https://shiny.rstudio.com/articles/modules.html

htmlwidgets

`htmlwidgets` is framework for embedding JavaScript visualizations into R.

Ready to use examples include:

- `leaflet` - Geo-spatial mapping
- `dygraphs` - Time series charting
- `MetricsGraphics` - Scatterplots and line charts with D3
- `networkD3` - Graph data visualization with D3
- `DataTables` - Tabular data display
- `threejs` - 3D scatterplots and globes
- `rCharts` - Multiple JavaScript charting libraries
- `d3heatmap` - Heatmaps
- `diagrammeR` - Graph and flowchart diagrams

Deployment

How to access a shiny app?

- R session
- shinyapps.io
- shiny server
- shinyproxy
- home made solution

https://shiny.rstudio.com/articles/#deployment

Useful companion tools

- `shinydahsboard`: dashboard layout
- `shinipsum`: fill app with random content
- `golem`: template for app development
- curated list https://github.com/nanxstats/awesome-shiny-extensions