Sebastian Didusch

A friendly introduction into R Shiny

HDYDI, 25th February, 2022

Outlook

1) Introduction and Motivation
2) The basics of web applications (Front end and Back end)
3) Mastering your User Inteface (UI)
4) Mastering your Server side
5) Styles (Layouts and Themes)

Introduction

1) Shiny is a library for the R programming language
2) Web development used to be (and still is) painful
3) Shiny has curated **UI**s (you don't need HTML/CSS/Javascript)
4) Provides **reactive** programming which automatically tracks the dependencies of pieces of code.

Motivation

- R is really popular in the life sciences
- Shiny is used to create stunning dashboards
- Shiny makes it quite easy transforming your analysis pipeline into an interactive web app
- Wonderful teaching, visualization and communication tool
- Check out https://shiny.rstudio.com/gallery/ for demo and example apps

https://vac-lshtm.shinyapps.io/ncov_tracker/

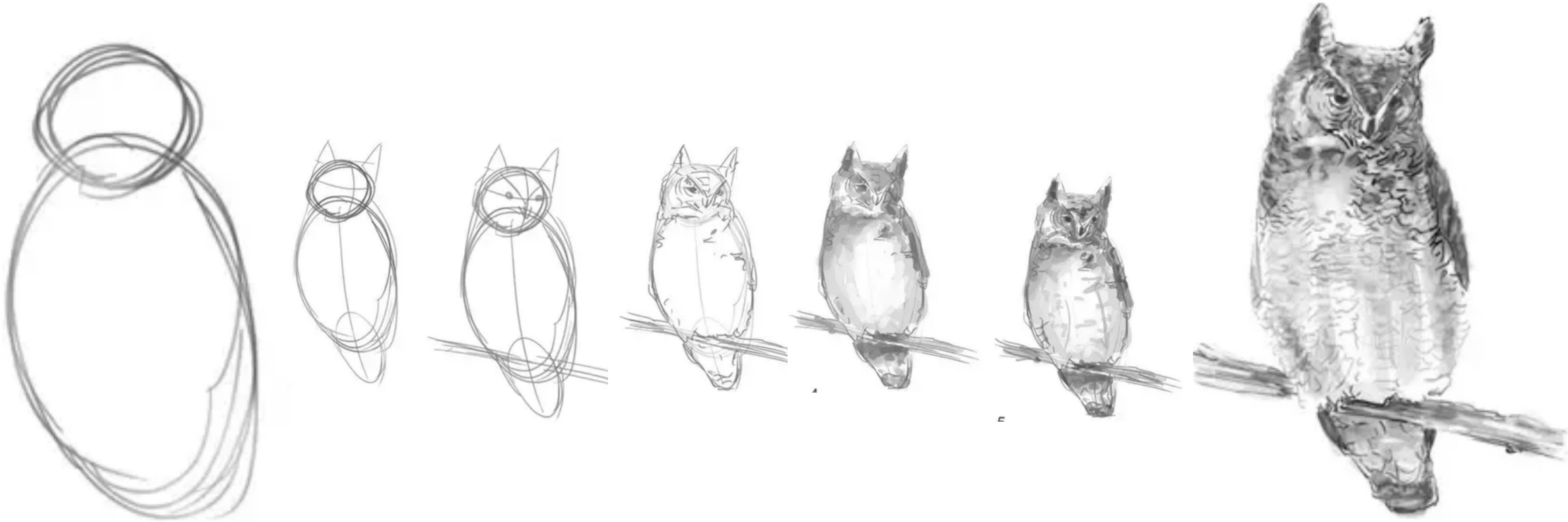
HOW TO DRAW AN OWL

1. Draw some circles

2. Draw the rest of the owl

https://www.quora.com/How-do-you-draw-an-owl

HOW TO DRAW AN OWL



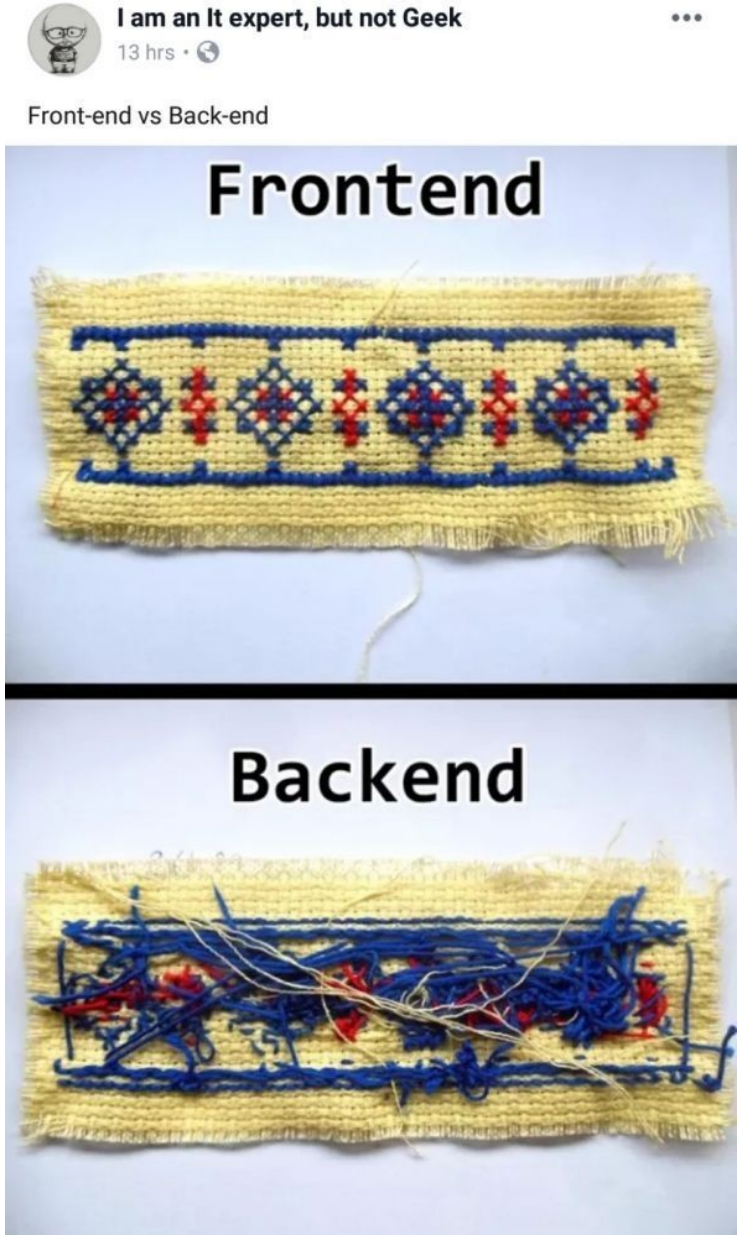
1. Draw some circles

2. Draw the rest of the owl

All code of this HDYDI is available at: https://github.com/tbaccata/hdydi_shiny

Front - and Back-end

- Front-end or **UI** contains the layout, the styling, **input** widgets and the **display of outputs** (e.g text, tables and plots)
- Back-end or **server** contains code that allows for **processing of data** and the **generation of outputs** (e.g text, tables and plots)



Getting started

- Several options to create Shiny app:
  - Create a new directory for your app, and put a single file called **app.R**
  - From Rstudio: **File | New Project**, then selecting **New Directory** and **Shiny Web Application**
- In app.R you need to assign a **ui** and a **server**

```
install.packages('shiny')
```

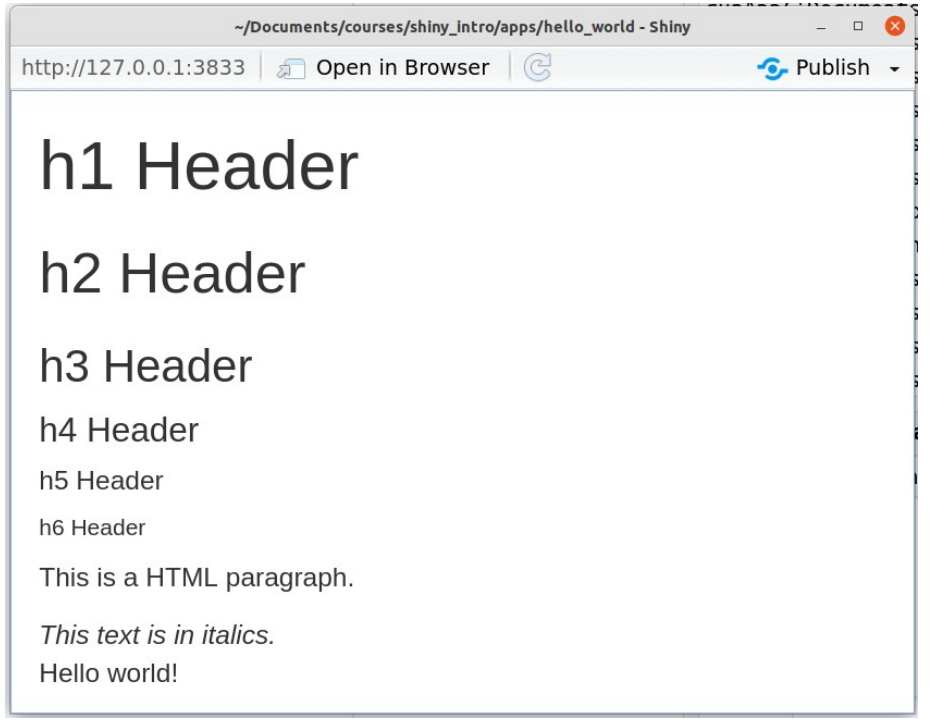
```
library(shiny)

ui <- fluidPage(
  h1("h1 Header"), # HTML header
  h2("h2 Header"), # HTML header
  h3("h3 Header"), # HTML header
  h4("h4 Header"), # HTML header
  h5("h5 Header"), # HTML header
  h6("h6 Header"), # HTML header
  p("This is a HTML paragraph."), # HTML paragraph
  em("This text is in italics."), # HTML italics
  p("Hello world!")
)

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

shinyApp(ui, server)
```

We will follow loosely an introduction from https://mastering-shiny.org/

Getting started

- Need to load shiny
- The ui defines the HTML page the user interacts with (a simple text greeting the world wholeheartedly)
- The behaviour of the app is defined in server (which is currently empty)
- App executes *shinyApp(ui, server)* to construct and start a Shiny application from UI and server.
- The app is listening on http://127.0.0.1:3833
- 127.0.0.1 is your computer
- Randomly assings a port

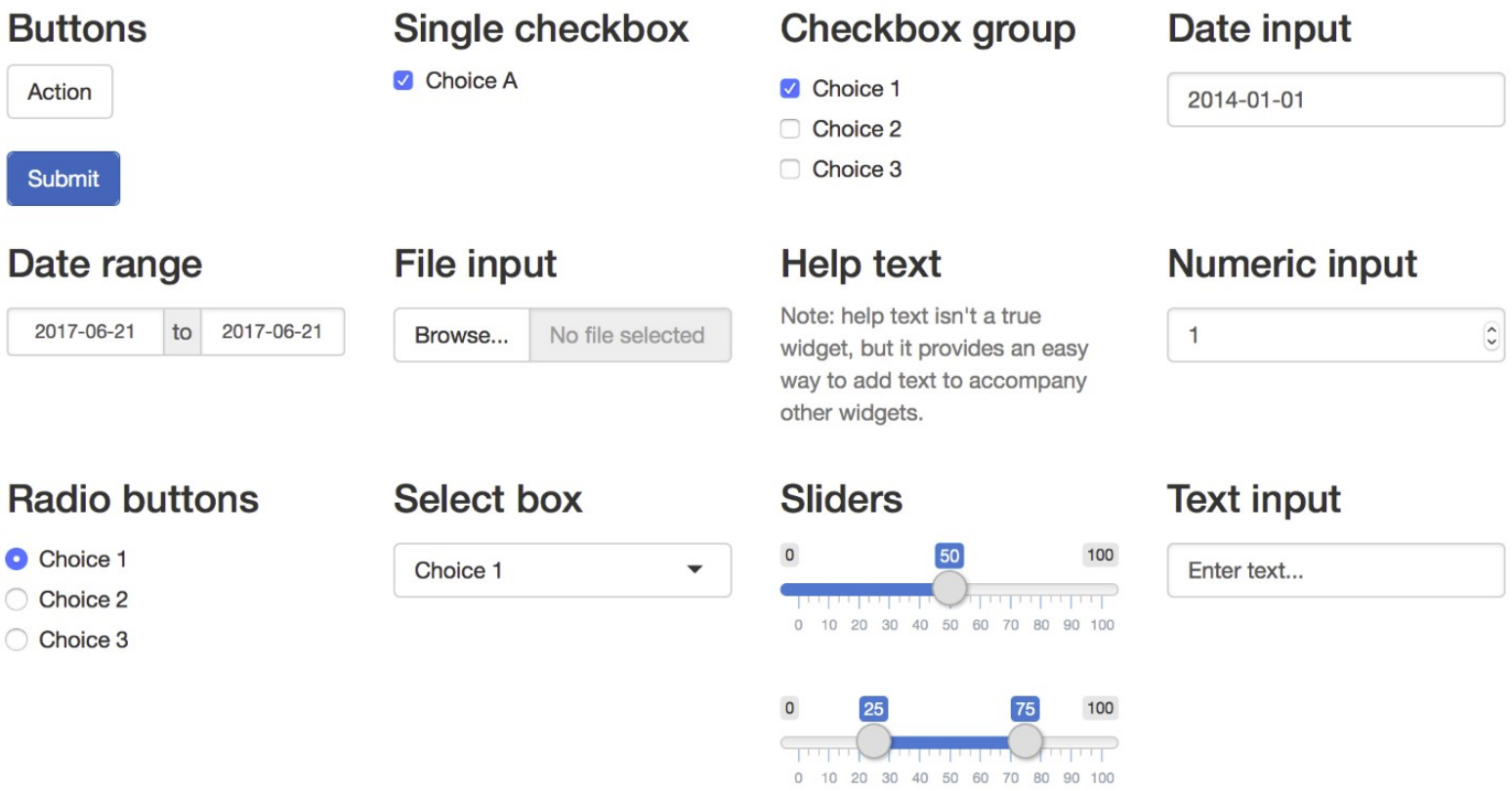
```
library(shiny)

ui <- fluidPage(
  h1("h1 Header"), # HTML header
  h2("h2 Header"), # HTML header
  h3("h3 Header"), # HTML header
  h4("h4 Header"), # HTML header
  h5("h5 Header"), # HTML header
  h6("h6 Header"), # HTML header
  p("This is a HTML paragraph."), # HTML paragraph
  em("This text is in italics."), # HTML italics
  p("Hello world!")
)

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

shinyApp(ui, server)
```



User Interface (UI)

UI | Input Widgets

- Add input widgets to UI:
- Input has inputID
- Input has a label
- **InputId needs to be unique!**



https://shiny.rstudio.com/gallery/widget-gallery.html

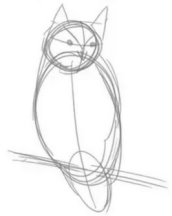
UI | Input Widgets

- Add input widgets to UI:
- Input has inputID
- Input has a label
- **InputId needs to be unique!**

```
ui <- fluidPage(
  textInput(
    inputId =  "strInput",
    label =  "Text input",
    value = ""
  ),
  numericInput(
    inputId = "numInput",
    label = "Numeric input",
    value = 0,
    step = 1
  ),
  sliderInput(
    inputId = "sliderInput",
    label = "Slider input",
    min = 0,
    max = 5,
    step = 1,
    value = 0
  )
)
```

UI | Outputs

- Outputs are generated in the server function and send back to the UI
- Text
  - textOutput for normal text
  - verbatimTextOutput for console output (like code)
- Tables
  - tableOutput for static tables (small summaries)
  - dataTableOutput for dynamic tables
- Plots
  - plotOutput for plots in base R and ggplot
  - plotlyOutput for plots produced with plotly



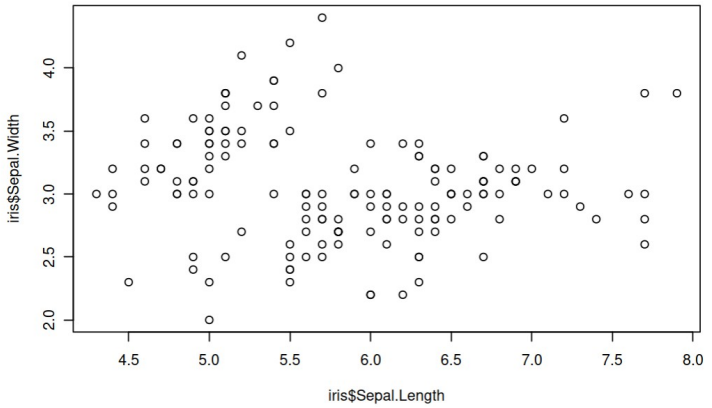
UI | Outputs

- In UI you define your Output (textOutput, tableOutput, plotOutput)
- In server you define the corresponding render function (renderText, renderTable, renderPlot)

```
library(shiny)

ui <- fluidPage(
  fluidRow(
    column(4,
           tableOutput("tblOutput"),
           ),
    column(4,
           textOutput("strOutput"),
    ),
    column(4,
           plotOutput("pltOutput")
    )
  )
)

server <- function(input, output, session) {
  output$strOutput <- renderText({"I am a text send from the server."})
  output$tblOutput <- renderTable({head(iris)}) # iris is default loaded data
  output$pltOutput <- renderPlot({plot(iris$Sepal.Length, iris$Sepal.Width)})
}

shinyApp(ui, server)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.10	3.50	1.40	0.20	setosa
4.90	3.00	1.40	0.20	setosa
4.70	3.20	1.30	0.20	setosa
4.60	3.10	1.50	0.20	setosa
5.00	3.60	1.40	0.20	setosa
5.40	3.90	1.70	0.40	setosa

I am a text send from the server.



Server

Server

- UI contains same HTML for every user
- Every user needs to get an independent version of the app (**User A** moving a slider shouldn't be visible for **User B**)
- Shiny invokes the server() function for every new session
- The 3 inputs (*input, output, session*) are created by Shiny

```
library(shiny)

ui <- fluidPage(
  # front end interface
)

server <- function(input, output, session) {
  # back end logic
}

shinyApp(ui, server)
```

Server | Input

- input Is a list-like object that contains all the input data sent from the browser, named according to the input ID
- You can access the value of that input with **input$count in a reactive context (**render**)**
- **input** objects are read-only → if you try to modify the input you'll get an error
- Everything displayed in the browser is the "single source of truth"

  → imagine if the input slider says **count** is 100, but in the server it would be a different value!

```
ui <- fluidPage(
  numericInput("count", label = "Number of values", value = 100)
)
```

```
server <- function(input, output, session) {
  input$count <- 10
}

shinyApp(ui, server)
#> Error: Can't modify read-only reactive value 'count'
```

Server | Output

- **output** is also a list-like object used to send **output**
- To add something to **output**, you have to be in a **reactive context** created by a function like `renderText({})` or `reactive()`
- The `render` function does two things:

  1) Sets up reactive context that tracks what inputs are used

  2) Converts R code into HTML for display

```
ui <- fluidPage(
  numericInput("count", "Number of values", value = 100),
  textOutput("trackCount")
)

server <- function(input, output, session) {
  output$trackCount <- renderText({
    paste0("Your count is ", input$count)
  })
}

shinyApp(ui, server)
```

Server | Output

- **output** is also picky how you use it
- You get an error message when you forget the render function

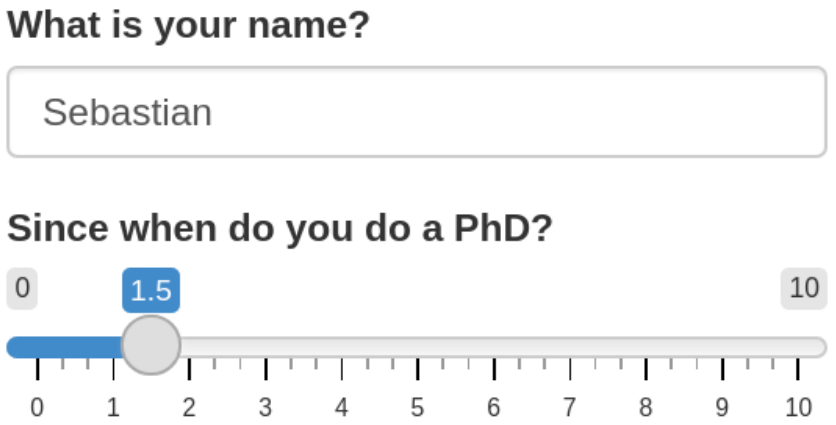
```
server <- function(input, output, session) {
  output$greeting <- "Hello human"
}
shinyApp(ui, server)
#> Error: Unexpected character object for output$greeting
#> ℹ Did you forget to use a render function?
```

- You get an error message when you attempt to read from an output → you have to create a **reactive context** first!

```
server <- function(input, output, session) {
  message("The greeting is ", output$greeting)
}
shinyApp(ui, server)
#> Error: Reading from shinyoutput object is not allowed.
```

Server | Reactive programming example

```
library(shiny)

ui <- fluidPage(
        textInput("name", "What is your name?",
        placeholder = "Please enter name..."),
           sliderInput("phdAge", "Since when do you do a PhD?",
                       min = 0, value = 0, max = 10, step = 0.5),
           h3("Greetings"),
           textOutput("greeting"),
           h3("PhD years to cat years converter"),
           textOutput("catYears")
)

server <- function(input, output, session) {
  output$greeting <- renderText({
    paste0("Well, hello there ", input$name, "!")
  })

  output$catYears <- renderText({
    catYears <- round(input$phdAge * 7, 2)
    paste0(input$phdAge, " human-years spend on PhD is equivalent to ",
    catYears,
    " cat-years.")
  })
}
```



Server | Reactive programming

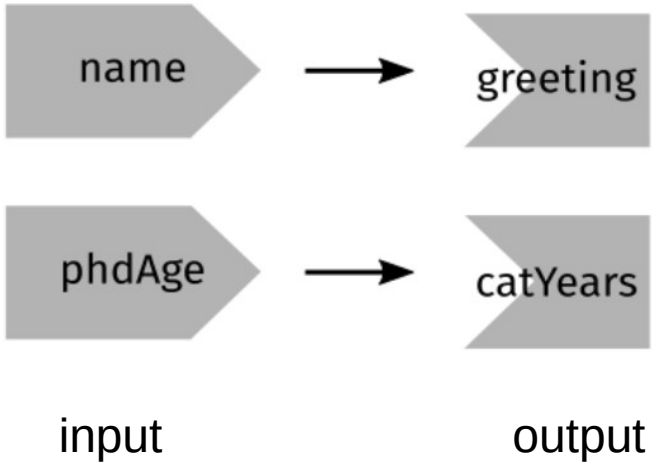
- Might read this as: paste together “Well, hello there“ and **input$name** and send it to **output$greeting**
- But how does Shiny know, when to update **output$greeting** and **output$catYears**?

```
output$greeting <- renderText({
  paste0("Well, hello there ", input$name, "!")
})

output$catYears <- renderText({
  catYears <- round(input$phdAge * 7, 2)
  paste0("If your PhD would be a cat, it would be ", catYears, " years old.")
})
```

Server | Reactive programming

- Might read this as: paste together “Well, hello there“ and **input$name** and send it to **output$greeting**
- But how does Shiny know, when to update **output$greeting** and **output$catYears**?

```
output$greeting <- renderText({
  paste0("Well, hello there ", input$name, "!")
})

output$catYears <- renderText({
  catYears <- round(input$phdAge * 7, 2)
  paste0("If your PhD would be a cat, it would be ", catYears, " years old.")
})
```



Server | Reactive programming

- Might read this as: paste together “Well, hello there“ and **input$name** and send it to **output$greeting**
- But how does Shiny know, when to update **output$greeting** and **output$catYears**?
- Code doesn’t *tell* Shiny to create these strings
- It informs how it *could create* these strings if *it needs to*
- It is up to Shiny when this code should run

```
output$greeting <- renderText({
  paste0("Well, hello there ", input$name, "!")
})

output$catYears <- renderText({
  catYears <- round(input$phdAge * 7, 2)
  paste0("If your PhD would be a cat, it would be ", catYears, " years old.")
})
```

Take home message:

You don’t tell Shiny: *bring me a beer*

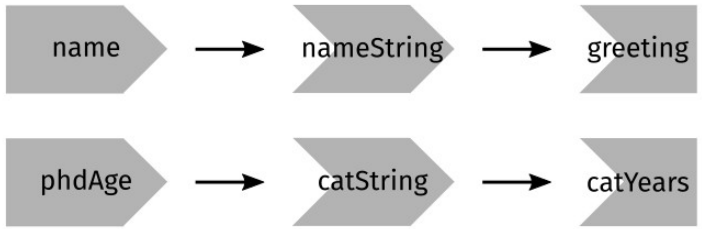
You tell Shiny: *you better make sure there is a beer in the fridge when I look inside it*

Think of your app as providing Shiny with recipes, not giving it commands

Server | Reactive expressions

- As your app gets bigger you end up with many **reactive dependencies**
- **Reactive expressions** allow you to reduce duplication in your code by adding **additional nodes** in your reactive graph
- There are like inputs: can be used in outputs
- There are like outputs: depend on inputs and automatically know when to update

```
nameString <- reactive({
  paste0("Well, hello there ", input$name, "!")
  })

catString <- reactive({
  catYears <- round(input$phdAge * 7, 2)
  paste0("If your PhD would be a cat, it would be ",
                                    catYears, " years old.")
  })
output$greeting <- renderText({
  nameString() # <- have to call reactive expression with brackets
})

output$catYears <- renderText({
  catYearsString()
})
```



Server | Excercises

- Given above ui:

  What is wrong in each server function?

```
ui <- fluidPage(
  textInput("name", "What is your name?", ""),
  h3("Greetings"),
  textOutput("greeting")
)
```

```
server1 <- function(input, output, server) {
  input$greeting <- renderText(paste0("Hello ", name))
}

server2 <- function(input, output, server) {
  greeting <- paste0("Hello ", input$name)
  output$greeting <- renderText(greeting)
}

server3 <- function(input, output, server) {
  output$greting <- paste0("Hello", input$name)
}
```

Server | Excercises solutions

- Given above ui:

  What is wrong in each server function?

- server1: input$name instead of name and output$greeting instead of input$greeting
- server2: cannot access input$name without reactive context → wrap paste0 in reac. expr. with reactive and call it with greetings() in renderText
- server3: same as server2 +

  typo: output$greting

```r
ui <- fluidPage(
  textInput("name", "What is your name?", ""),
  h3("Greetings"),
  textOutput("greeting")
)
```

```r
server1 <- function(input, output, server) {
  input$greeting <- renderText(paste0("Hello ", name))
}
```

```r
server2 <- function(input, output, server) {
  greeting <- paste0("Hello ", input$name)
  output$greeting <- renderText(greeting)
}
```

```r
server3 <- function(input, output, server) {
  output$greting <- paste0("Hello", input$name)
}
```

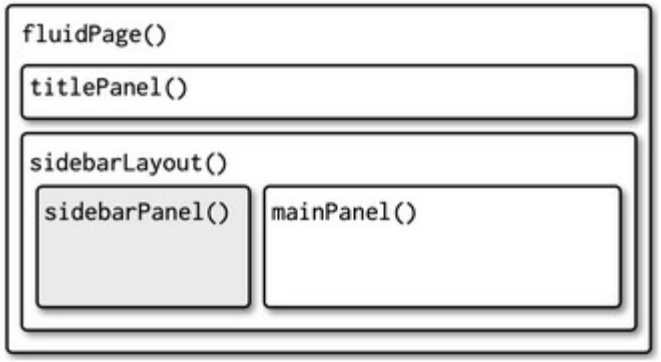
Layouts and themes

MAX PERUTZ LABS VIENNA

Side bar layout

- So far we crammed the UI into a single page layout with fluidPage
- Hierarchical organization
- More complex layouts organize content in columns
- E.g sidebarLayout with sidebarPanel and mainPanel

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



Side bar layout example

```
ui <- fluidPage(
  titlePanel("Simulate a normal distribution"),
  sidebarLayout(
    sidebarPanel(
      h3("Simulate a normal distribution"),
      numericInput("number", "number of data points (n)", value = 50,
                   min = 10, max = 1000),
      numericInput("mean", "Mean", value = 0,
                   min = -1e8, max = 1e8),
      numericInput("sd", "Standard deviation", value = 1,
                   min = 0, max = 100),
      actionButton("submit", "Simulate")
      ),
    mainPanel(
      plotOutput("normalDistPlot")
    )
  )
)

server <- function(input, output, session) {
  normalData <- eventReactive(input$submit, { # only executes when button is pressed
    rnorm(n = input$number, mean = input$mean, sd = input$sd)
  })
  output$normalDistPlot <- renderPlot({
    plot(density(normalData()))
  })
}
```

Simulate a normal distribution

Simulate a normal distribution

number of data points (n)

50

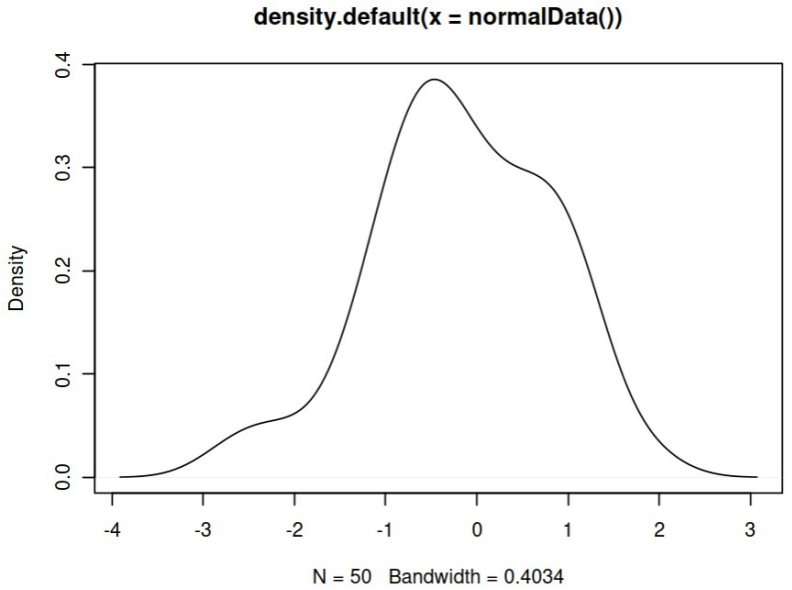
Mean

0

Standard deviation

1

Simulate

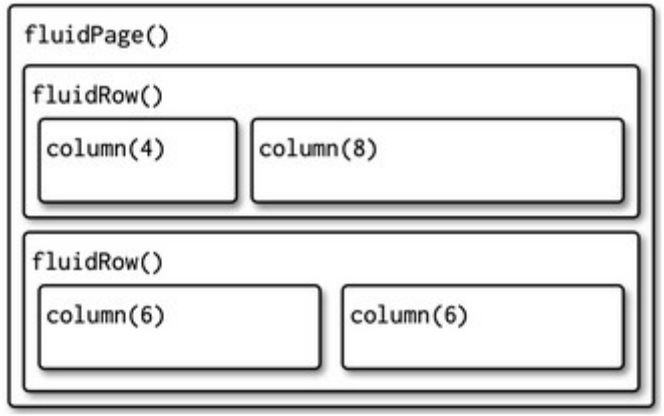


MAX PERUTZ LABS VIENNA

Multi row layouts

- sidebarLayout is built on multiple rows fluidPage
- You can specify your desired number of columns
- Each column takes a width argument
- Each fluidRow has a max. width of 12

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



MAX PERUTZ LABS VIENNA

Multi page layouts

- After all, you would like to display your content on multiple pages
- navBarPage provides you a tab bar
- Each tab is in a tabPanel
- navbarMenu allows you to nest tabPanels

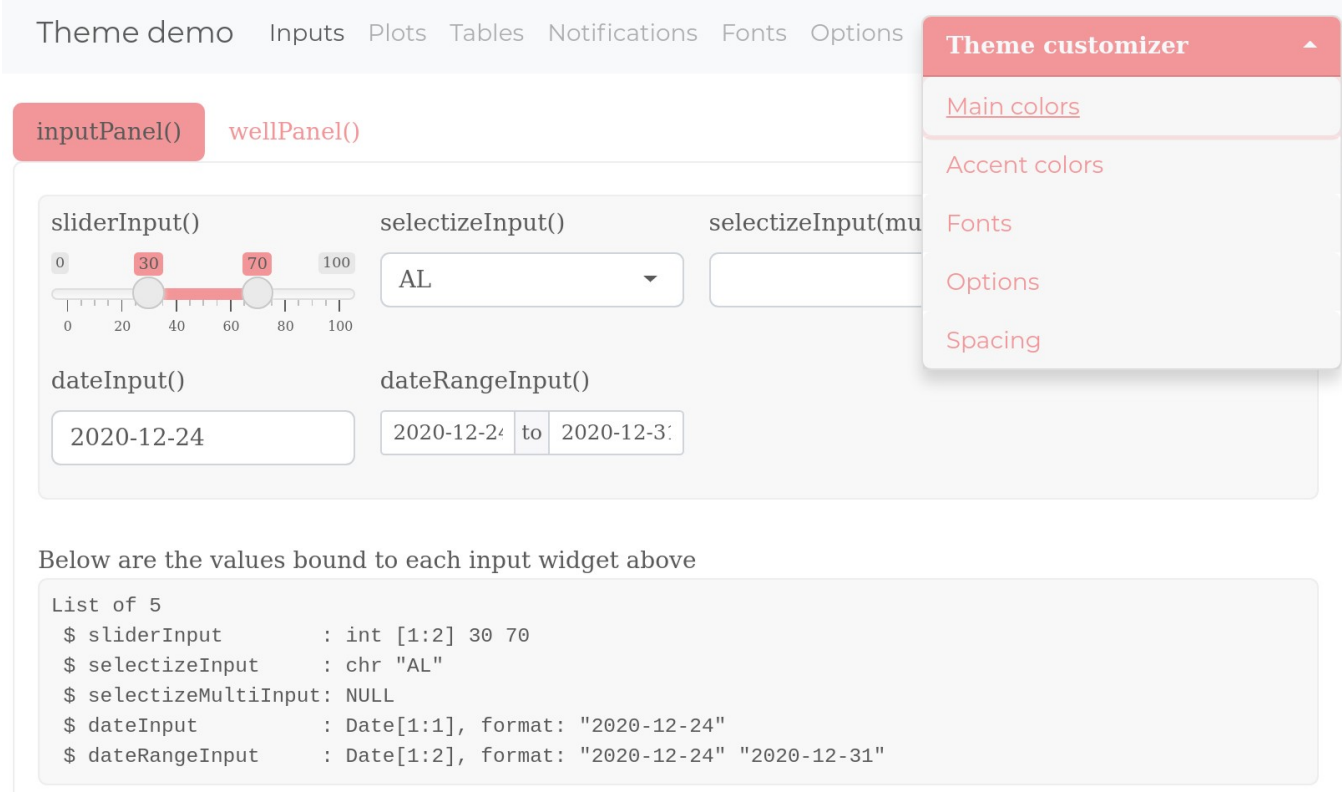
```
ui <- navbarPage(
  "Page title",
  tabPanel("panel 1", ...),
  tabPanel("panel 2", ...),
  tabPanel("panel 3", ...),
  navbarMenu("subpanels",
             tabPanel("panel 4a", ...),
             tabPanel("panel 4b", ...),
             tabPanel("panel 4c", ...)
  )
)
```

MAX PERUTZ LABS VIENNA

Themes

- The bslib package gives many options of freely available themes you and a great way to construct your own themes
- The bs_theme_preview function will open a Shiny app displaying a theme (and it let's you customize it!)
- Free themes: https://bootswatch.com/minty/
- Google fonts: https://fonts.google.com/

```r
install.packages("bslib")
library(bslib)

theme <- bslib::bs_theme(
  bg = "#0b3d91",
  fg = "white",
  base_font = "Source Sans Pro"
)
bslib::bs_theme_preview(theme)
```



MAX PERUTZ LABS VIENNA

Summary

- Add all your inputs and outputs to UI
- Define your outputs within a **reactive context** with the render functions in the server, creating reactive dependencies on the inputs
- Feel free to contact me if you have any questions!

```
library(shiny)

ui <- fluidPage(
  textInput("name", "What's your name?", placeholder = "..."),
  textOutput("greeting")
  )

server <- function(input, output, session) {
  output$greeting <-
    renderText({
      paste0("Well, hello there ", input$name, "!")
    })
}

shinyApp(ui, server)
```

MAX PERUTZ LABS VIENNA

Useful packages and ressources

- plotly (interactive plotting, easy to use with ggplot)
- shinyjs (common JavaScript operations in Shiny apps)
- bslib (bootstrap themes)

Check out the links below for a comprehensive list of ressources:

Example apps: https://github.com/tbaccata/hdydi_shiny

Packages: https://github.com/nanxstats/awesome-shiny-extensions

Tutorial: https://mastering-shiny.org/