Shiny: From Beginner to `I Know How to Google it'

Appsilon
Douglas Mesquita

Belo Horizonte
2025-02-03 - 2025-02-06

Appsilon

Appsilon

About me

- Master`s and PhD in Statistics (UFMG)
- R/Shiny developer for 3 years
- Statistician/Data scientist 10+ years
- Open source contributor

github.com/DouglasMesquita

linkedin.com/in/douglas-mesquita

www.require-r.com

Appsilon

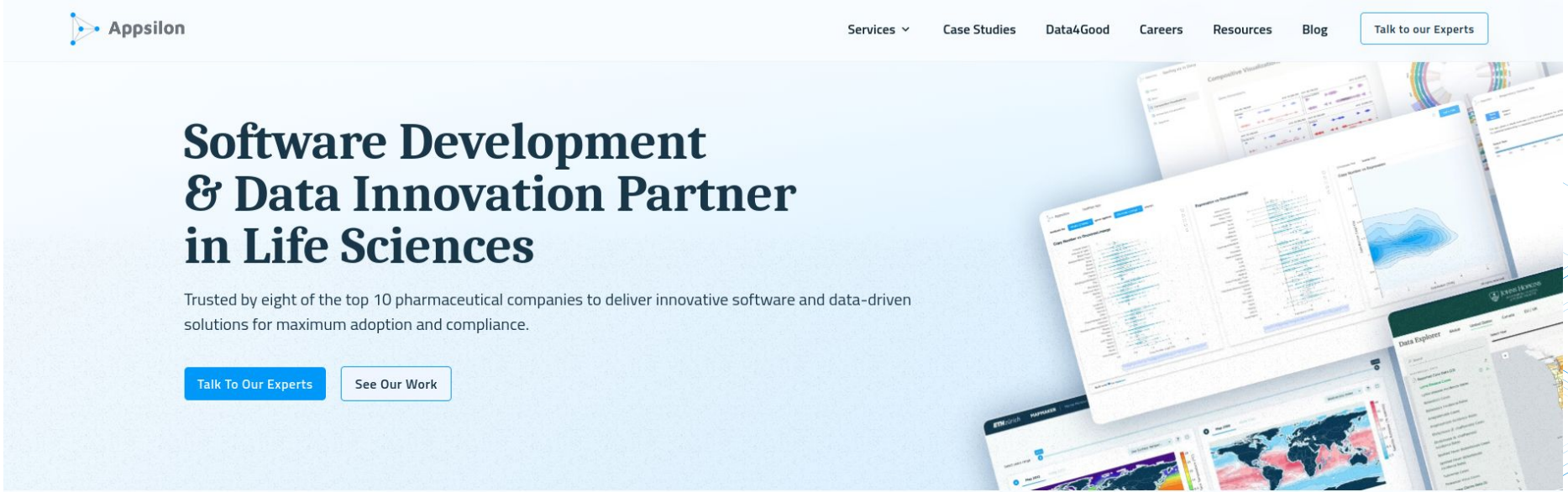
About me



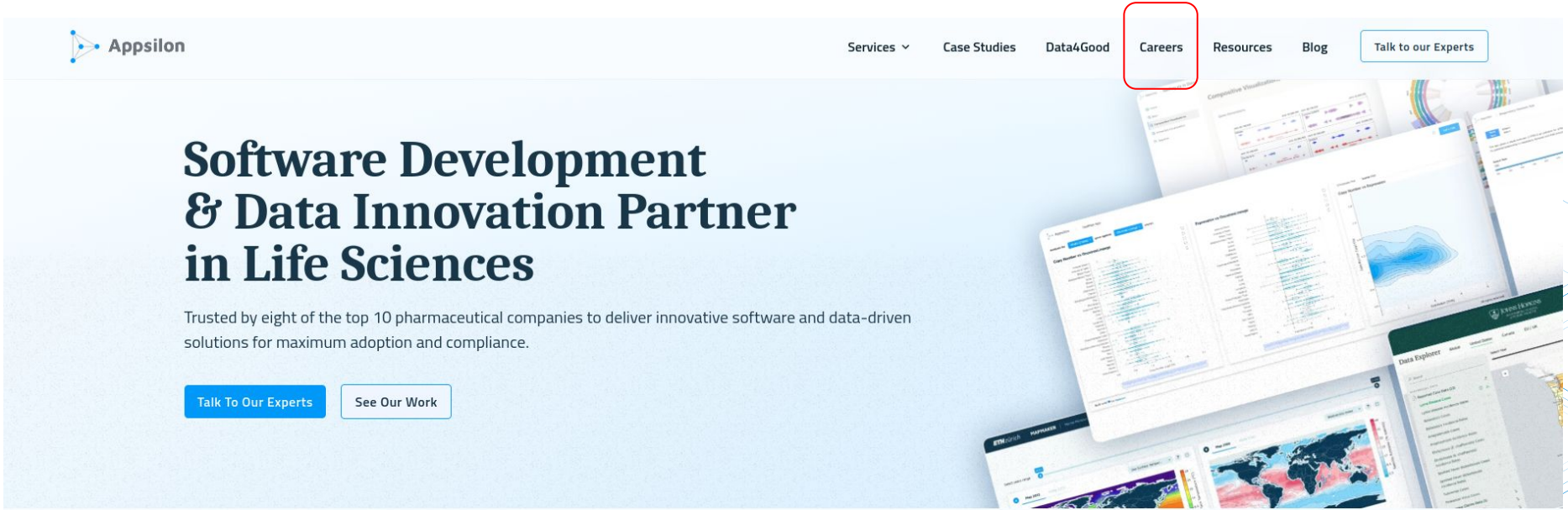


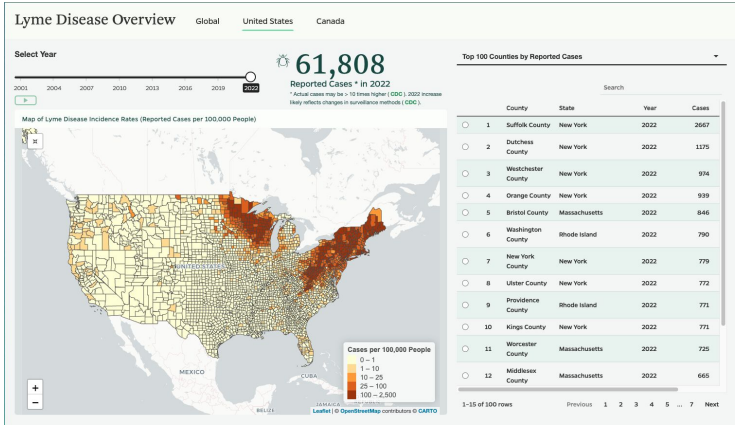
🇧🇷🇲🇿 🇵🇹 🇪🇸 🇫🇷 🇬🇧 🇺🇸 🇵🇱 🇨🇿 🇦🇹 🇭🇺 🇸🇰 🇫🇮 🇪🇪 🇱🇻 🇱🇹 🇮🇹 🇳🇱 🇨🇭 🇩🇪

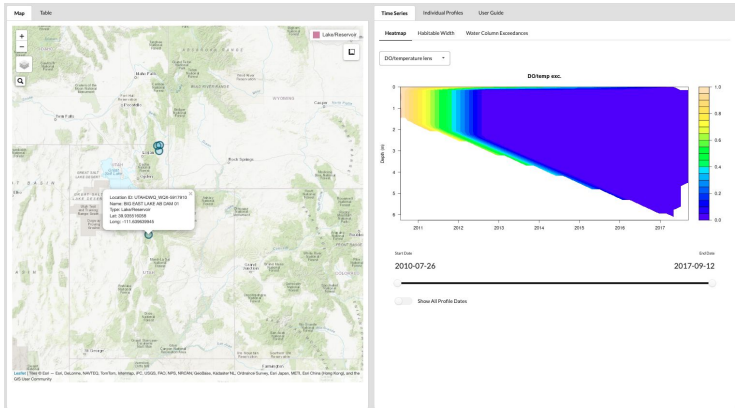
Appsilon


Appsilon
Services
Case Studies
Data4Good
Careers
Resources
Blog
Talk to our Experts
Software Development
& Data Innovation Partner
in Life Sciences
Trusted by eight of the top 10 pharmaceutical companies to deliver innovative software and data-driven solutions for maximum adoption and compliance.
Talk To Our Experts
See Our Work

Appsilon


Appsilon
Services
Case Studies
Data4Good
Careers
Resources
Blog
Talk to our Experts
Software Development
& Data Innovation Partner
in Life Sciences
Trusted by eight of the top 10 pharmaceutical companies to deliver innovative software and data-driven solutions for maximum adoption and compliance.
Talk To Our Experts
See Our Work





🌍 Join Appsilon as an R/Shiny Developer! 🚀

- 🔹 Solve real-world problems – build interactive data science applications for global organizations.
- 🔹 Open-source & impact – contribute to tools used by thousands of analysts worldwide.
- 🔹 Modern tech stack – R, Shiny, Python, AWS, Docker, and more!
- 🔹 Global & remote – work from anywhere with a team of top engineers and data scientists.
- 🔹 Growth & innovation – work on cutting-edge projects in AI, ML, and data visualization.

👉 Ready to level up? Check out our job offer!

R Developer with Life Science Background

Base Salary

(Monthly for a B2B contract, + VAT)

Up to $5650

up to 25.200 PLN

Total Compensation Value

Up to $7100

Up to 31.650 PLN

Apply now

R/Shiny Developer

Base Salary

(Monthly for a B2B contract, + VAT)

Up to $5650

Up to 25.200 PLN

Total Compensation Value

Up to $7100

Up to 31.650 PLN

Apply now

Level up your expertise in R/Shiny

How to Start a Career as an R Shiny Developer

Reading time: 10 min | shiny | career | r | community | r community

By: **Dario Radečić** October 26, 2021

How to Develop an R Shiny Dashboard In 10 Minutes or Less

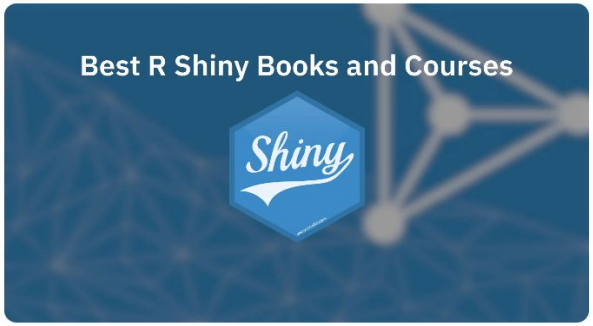
Reading time: 5 min | shiny | shiny dashboards | r | community | tutorials

By: **Dario Radečić** January 25, 2022

Top 7 Best R Shiny Books and Courses That Are Completely Free

Reading time: 5 min | community | shiny | r

By: **Dario Radečić** January 6, 2022

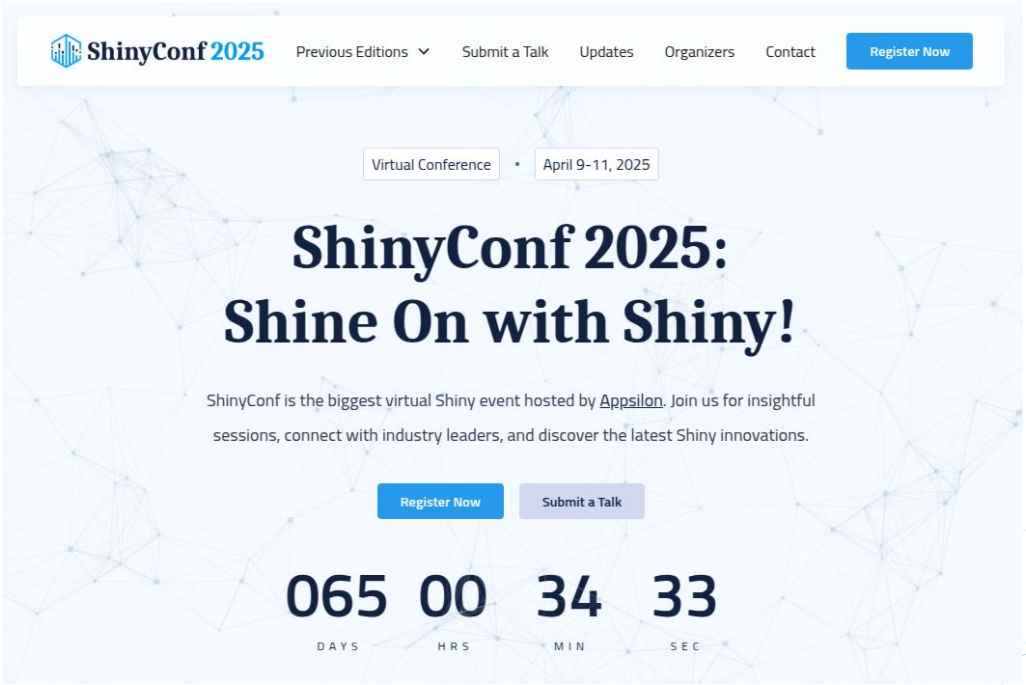


Appsilon

Check our Careers Page

Appsilon

shiny::conf



https://www.shinyconf.com/

Agenda

Appsilon
Douglas Mesquita

Belo Horizonte
2025-02-03 - 2025-02-06

Appsilon

Appsilon

Agenda

- Motivation
- First Steps in Shiny
- Reactivity
- Debugging
- UI/UX in Shiny
- Using Modules and Box
- Large Applications
- What else in shiny?
- Final exercise

Motivation

Appsilon
Douglas Mesquita

Belo Horizonte
2025-02-03 - 2025-02-06

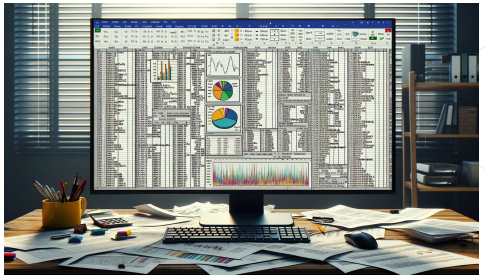
Appsilon

Appsilon

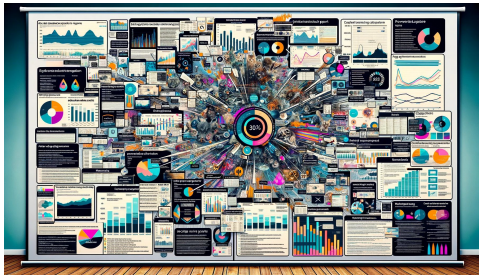
Why Shiny?

Commonly Delivered Products

Reports



Spreadsheets

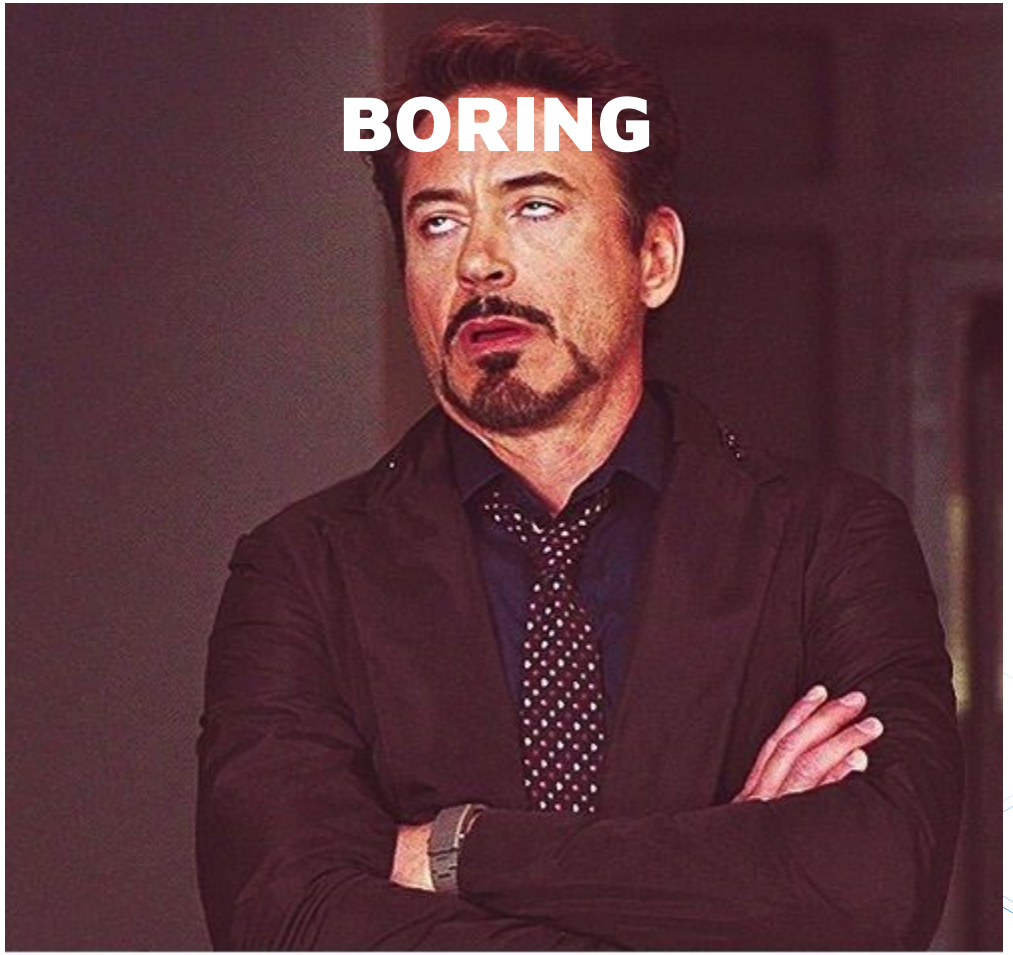


Presentations

Several files

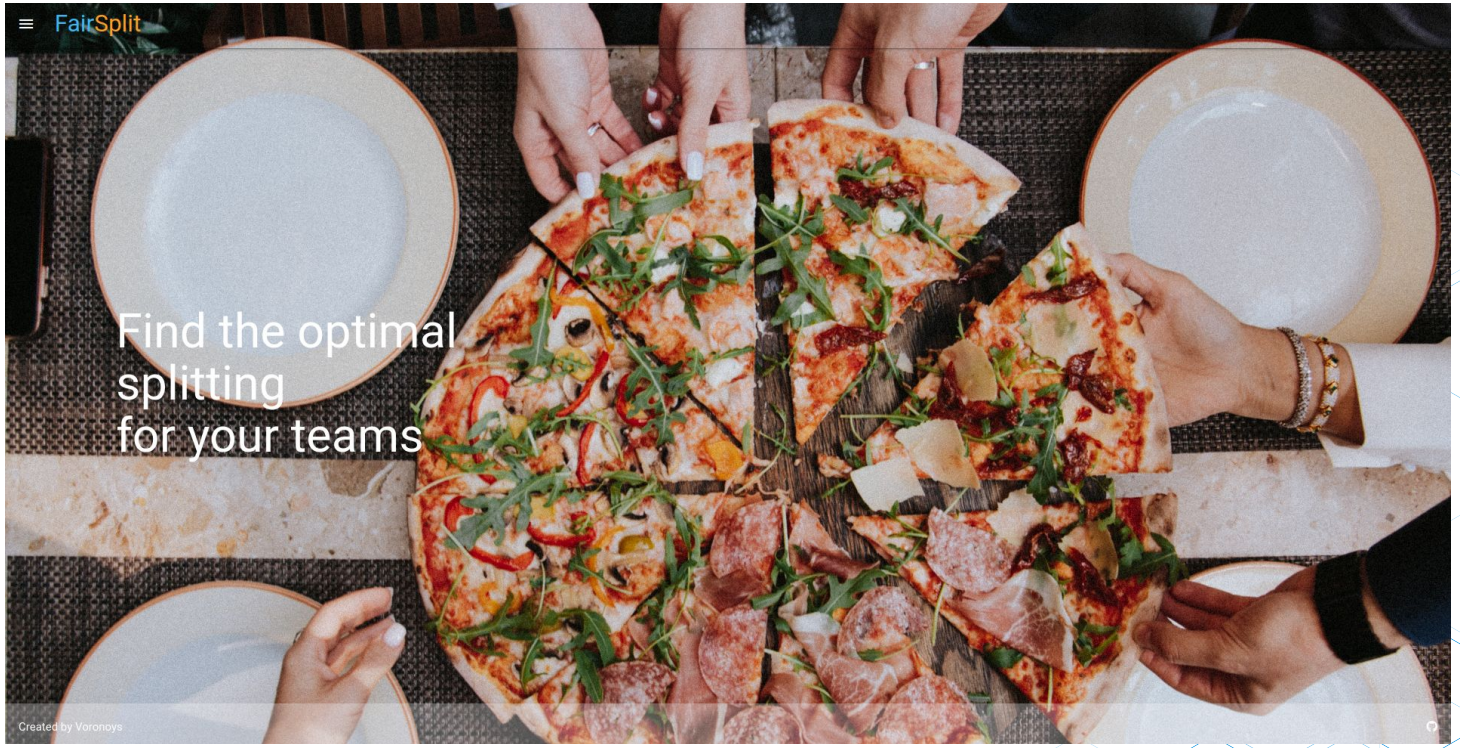
https://lgsilvaesilva.github.io/shiny101/materials/1-single-docs/1-welcome-to-quarto/slides.html#/produtos-tradicionais

Appsilon



Appsilon

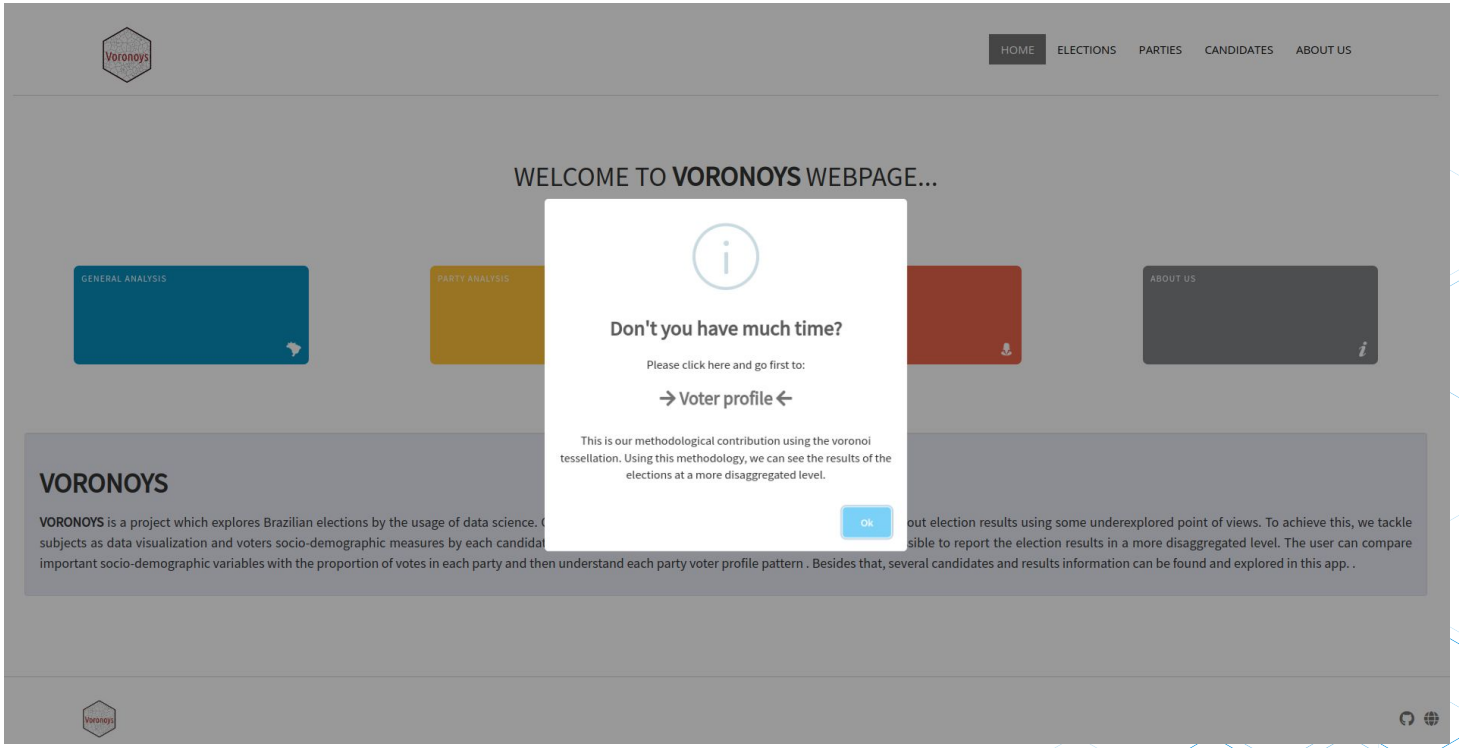
What if?



https://voronoys.shinyapps.io/fairsplit/

Appsilon

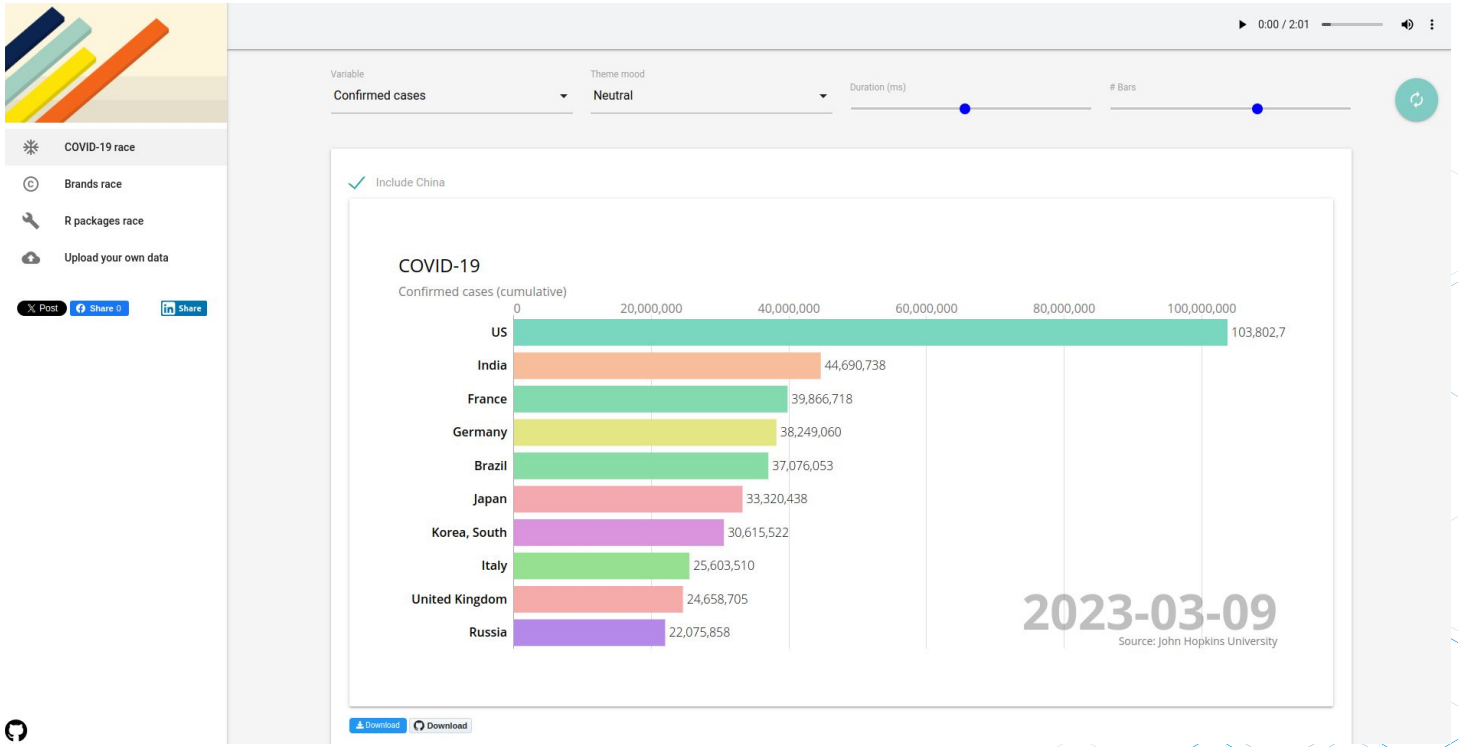
What if?



https://voronoys.shinyapps.io/voronoys/

Appsilon

What if?



https://voronoys.shinyapps.io/barchartraceR2D3/

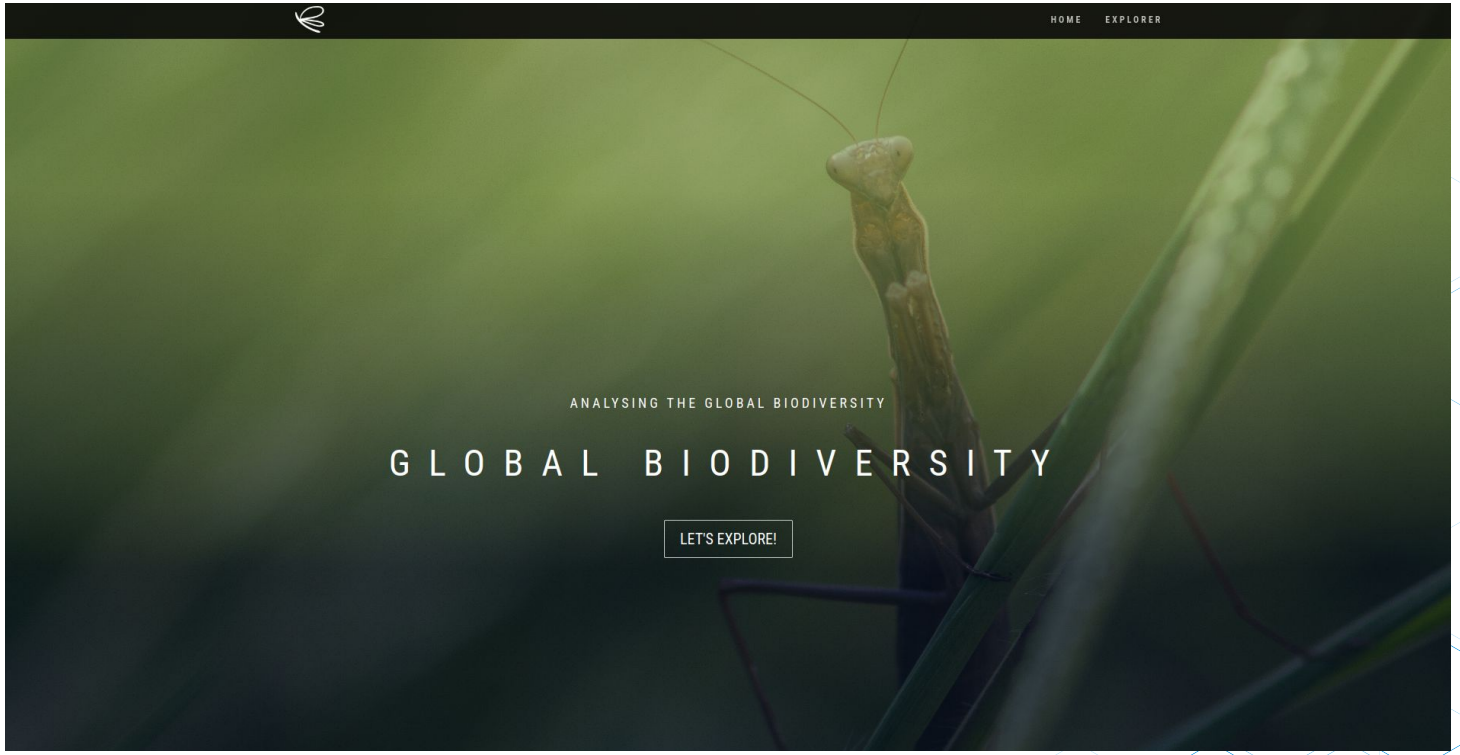
Appsilon

What if?

https://require-r.shinyapps.io/marineApp/

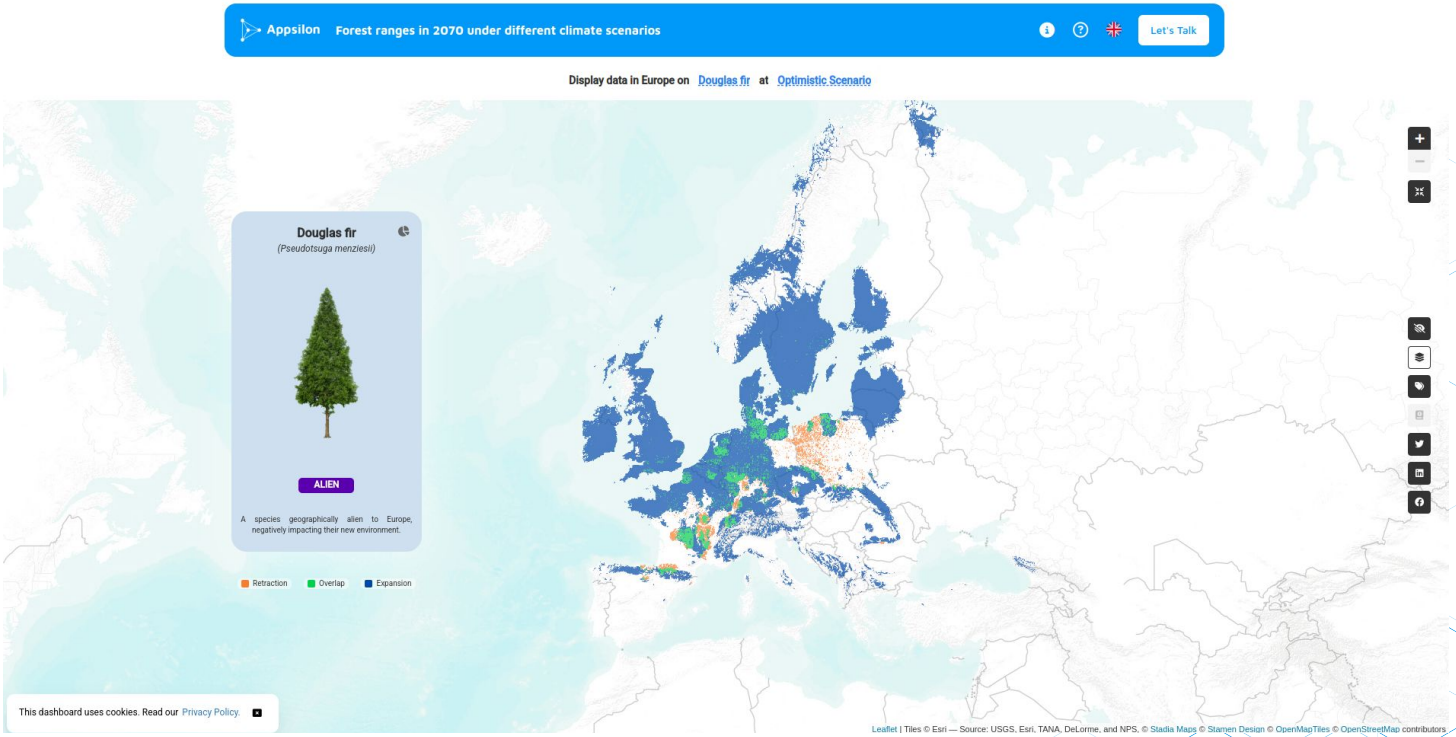
Appsilon

What if?



https://lgsilvaesilva.shinyapps.io/biodiversity2/

What if?

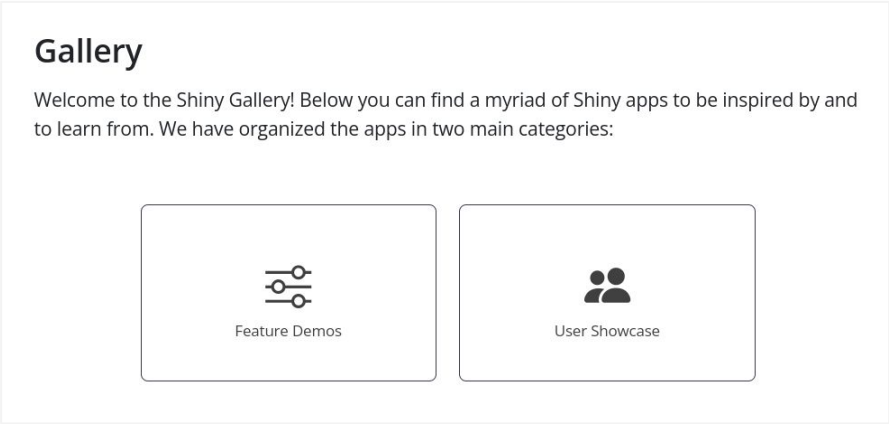


https://connect.appsilon.com/future-forests/

What if?

shiny.posit.co/r/gallery

demo.appsilon.com



https://connect.appsilon.com/future-forests/

Hands-on

Appsilon

Cheeses dataset

Read cheeses.csv and get used to the data

First Steps in Shiny

Appsilon
Douglas Mesquita

Belo Horizonte
2025-02-03 - 2025-02-06

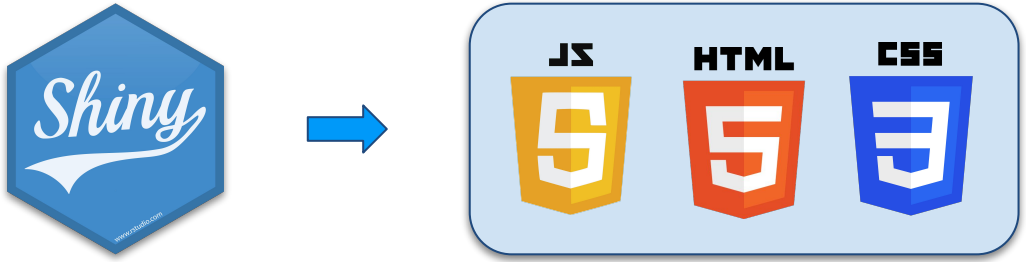
Appsilon

What is Shiny?

Shiny is a framework for building web applications using R. Designed with data scientists in mind, it allows you to create highly sophisticated apps **without requiring any knowledge of HTML, CSS, or JavaScript**.

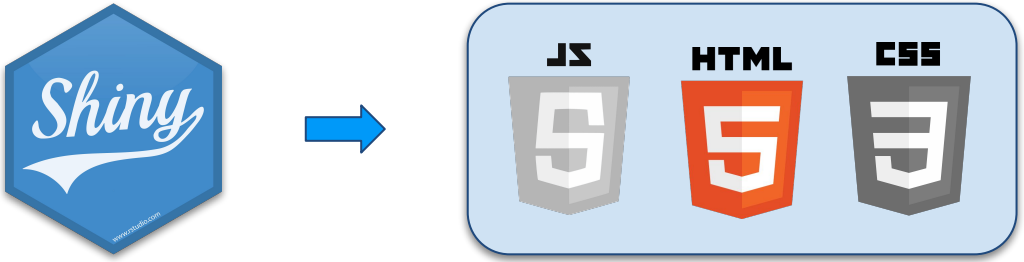
Appsilon

What is Shiny?



Appsilon

What is Shiny?



Appsilon

What is Shiny?

Hello world!

Appsilon

First steps

```
1.  library(shiny)
2.
3.  ui <- fluidPage(
4.   "Hello World!"
5.  )
6.
7.  server <- function(input, output, session) {}
8.
9.  shinyApp(ui, server)
```

Appsilon

First steps

1. **library(shiny)** to load the shiny package

```
1.  library(shiny)
2.
3.  ui <- fluidPage(
4.   "Hello World!"
5.  )
6.
7.  server <- function(input, output, session) {}
8.
9.  shinyApp(ui, server)
```

Appsilon

First steps

```
1.  library(shiny)
2.
3.  ui <- fluidPage(
4.   "Hello World!"
5.  )
6.
7.  server <- function(input, output, session) {}
8.
9.  shinyApp(ui, server)
```

1. **library(shiny)** to load the shiny package

2. **ui** defines the user interface, the HTML webpage that humans interact with

Appsilon

First steps

```
1.  library(shiny)
2.
3.  ui <- fluidPage(
4.   "Hello World!"
5.  )
6.
7.  server <- function(input, output, session) {}
8.
9.  shinyApp(ui, server)
```

1. **library(shiny)** to load the shiny package
2. **ui** defines the user interface, the HTML webpage that humans interact with
3. **server** specifies the behaviour of our app by defining a server function

runApp('examples/01_first_steps/01_hello_world/01_hello_world')

Appsilon

First steps

```
1.  library(shiny)
2.
3.  ui <- fluidPage(
4.   "Hello World!"
5.  )
6.
7.  server <- function(input, output, session) {}
8.
9.  shinyApp(ui, server)
```

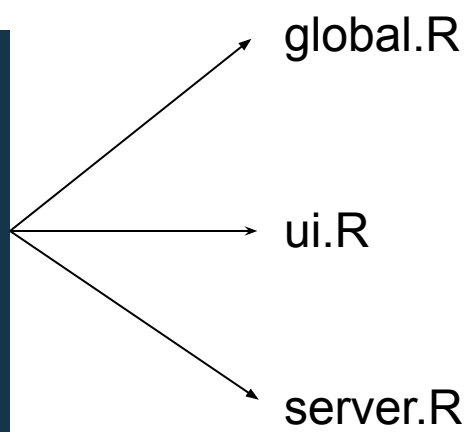
1. **library(shiny)** to load the shiny package
2. **ui** defines the user interface, the HTML webpage that humans interact with
3. **server** specifies the behaviour of our app by defining a server function
4. **shinyApp(ui, server)** to construct and start a Shiny application from UI and server

runApp('examples/01_first_steps/01_hello_world/multiple_files')

Appsilon

First steps

```
1.  library(shiny)
2.
3.  ui <- fluidPage(
4.   "Hello World!"
5.  )
6.
7.  server <- function(input, output, session) {}
8.
9.  shinyApp(ui, server)
```

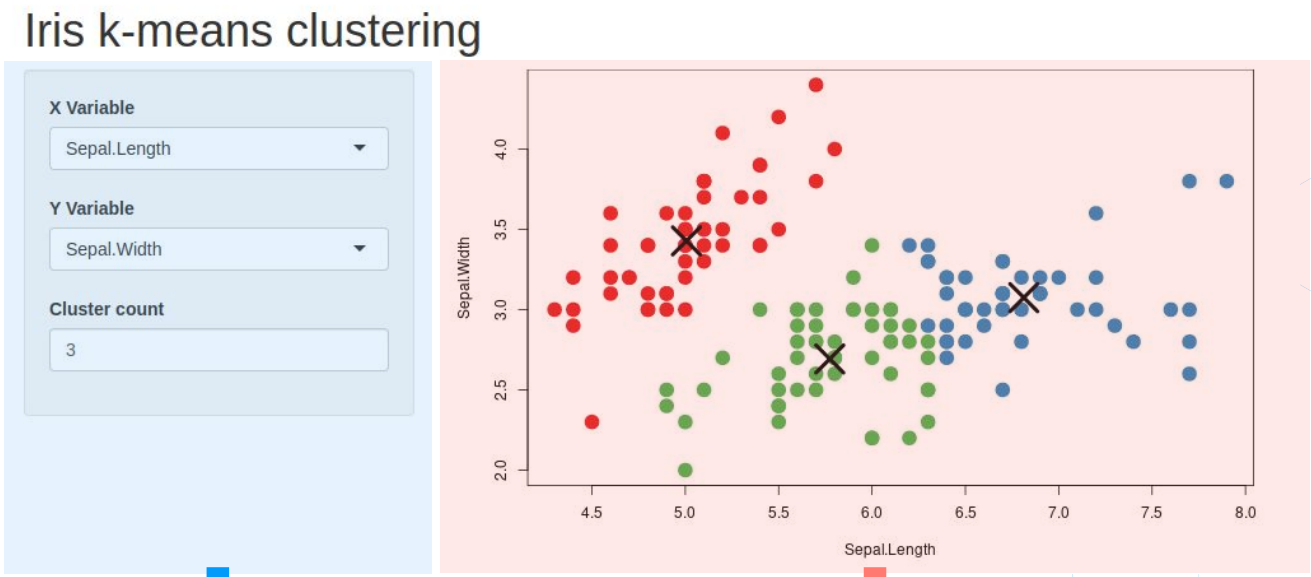


UI - User Interface

Appsilon

Appsilon

User Interface



Appsilon

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 (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

https://shiny.posit.co/r/gallery/widgets/widget-gallery/

Appsilon

Inputs

```
 1.  library(shiny)
 2.
 3.  source("examples/utils.R")
 4.  cheeses <- read.csv("data/cheeses.csv")
 5.  country_choices <- split_unique(cheeses$country)
 6.
 7.  ui <- fluidPage(
 8.   selectInput(
 9.     inputId = "countries",
10.     label = "Countries",
11.     choices = country_choices,
12.     selected = country_choices[1],
13.     multiple = FALSE
14.   )
15.  )
16.
17.  server <- function(input, output, session) {}
18.
19.  shinyApp(ui, server)
```

Appsilon

Inputs

```
1.  library(shiny)
2.
3.  source("examples/utils.R")
4.  cheeses <- read.csv("data/cheeses.csv")
5.  country_choices <- split_unique(cheeses$country)
6.
7.  ui <- fluidPage(
8.   selectInput(
9.     inputId = "countries",
10.    label = "Countries",
11.    choices = country_choices,
12.    selected = country_choices[1],
13.    multiple = FALSE
14.  )
15. )
16.
17. server <- function(input, output, session) {}
18.
19. shinyApp(ui, server)
```

- **inputId:** Must be unique!!!

Appsilon

Inputs

```
1.  library(shiny)
2.
3.  source("examples/utils.R")
4.  cheeses <- read.csv("data/cheeses.csv")
5.  country_choices <- split_unique(cheeses$country)
6.
7.  ui <- fluidPage(
8.   selectInput(
9.     inputId = "countries",
10.    label = "Countries",
11.    choices = country_choices,
12.    selected = country_choices[1],
13.    multiple = FALSE
14.  )
15. )
16.
17. server <- function(input, output, session) {}
18.
19. shinyApp(ui, server)
```

- **inputId:** Must be unique!!!
- **label:** Text that will be displayed next to the widget

Appsilon

Inputs

```
 1.  library(shiny)
 2.
 3.  source("examples/utils.R")
 4.  cheeses <- read.csv("data/cheeses.csv")
 5.  country_choices <- split_unique(cheeses$country)
 6.
 7.  ui <- fluidPage(
 8.   selectInput(
 9.     inputId = "countries",
10.     label = "Countries",
11.     choices = country_choices,
12.     selected = country_choices[1],
13.     multiple = FALSE
14.   )
15.  )
16.
17.  server <- function(input, output, session) {}
18.
19.  shinyApp(ui, server)
```

- **inputId:** Must be unique!!!
- **label:** Text that will be displayed next to the widget
- **choices:** Vector containing all the available choices

Appsilon

Inputs

```
 1.  library(shiny)
 2.
 3.  source("examples/utils.R")
 4.  cheeses <- read.csv("data/cheeses.csv")
 5.  country_choices <- split_unique(cheeses$country)
 6.
 7.  ui <- fluidPage(
 8.   selectInput(
 9.     inputId = "countries",
10.     label = "Countries",
11.     choices = country_choices,
12.     selected = country_choices[1],
13.     multiple = FALSE
14.   )
15.  )
16.
17.  server <- function(input, output, session) {}
18.
19.  shinyApp(ui, server)
```

- **inputId**: Must be unique!!!
- **label**: Text that will be displayed next to the widget
- **choices**: Vector containing all the available choices
- **selected**: The initial selected element

examples/01_first_steps/03_inputs/01_selectInput.R

Appsilon

Inputs

```
 1.  library(shiny)
 2.
 3.  source("examples/utils.R")
 4.  cheeses <- read.csv("data/cheeses.csv")
 5.  country_choices <- split_unique(cheeses$country)
 6.
 7.  ui <- fluidPage(
 8.   selectInput(
 9.     inputId = "countries",
10.     label = "Countries",
11.     choices = country_choices,
12.     selected = country_choices[1],
13.     multiple = FALSE
14.   )
15.  )
16.
17.  server <- function(input, output, session) {}
18.
19.  shinyApp(ui, server)
```

- **inputId**: Must be unique!!!
- **label**: Text that will be displayed next to the widget
- **choices**: Vector containing all the available choices
- **selected**: The initial selected element
- **multiple**: Boolean indicating if the user is allowed to select multiple elements

Appsilon

Inputs

```r
selectInput(
 inputId = "my_select_input",
 label = "Select a color",
 choices = c("red", "green", "blue"),
 selected = "red",
 multiple = TRUE
)
```

```html
<div class="form-group shiny-input-container">
 <label class="control-label" id="my_select_input-label" for="my_select_input">Select a color</label>
 <div>
   <select id="my_select_input" class="shiny-input-select" multiple="multiple"><option value="red" selected>red</option>
<option value="green">green</option>
<option value="blue">blue</option></select>
   <script type="application/json" data-for="my_select_input">{"plugins":["selectize-plugin-a11y"]}</script>
 </div>
</div>
```

Hands-on

Appsilon

Cheeses dataset

1. *Create the following inputs (utils/split_unique may be useful)*
    a. *Type*
    b. *Color*
    c. *Flavor*
    d. *Aroma*
    e. *Vegetarian*
    f. *Vegan*
    g. *Cheese type*

Think about which widget would be better in each case

Appsilon

Outputs

dataTableOutput Interactive table
htmlOutput HTML content
imageOutput images (png, jpeg, ...)
plotOutput Normal R plots
tableOutput Tables (data.frames)
textOutput Text content
uiOutput Other UI components
verbatimTextOutput Text (code)

Appsilon

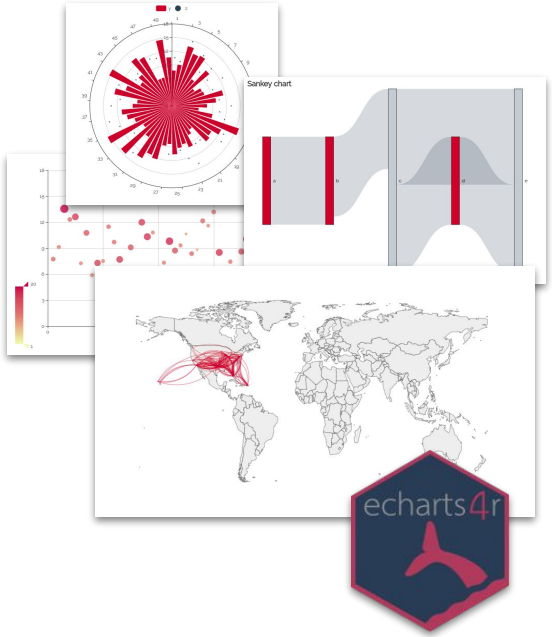
Outputs (displaying)

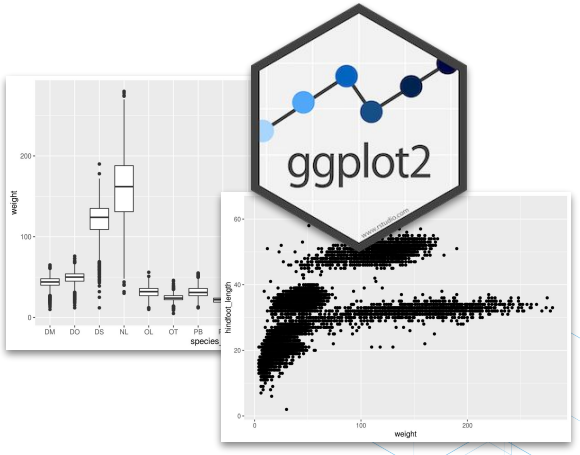
dataTableOutput	Interactive table
htmlOutput	HTML content
imageOutput	images (png, jpeg, ...)
plotOutput	Normal R plots
tableOutput	Tables (data.frames)
textOutput	Text content
uiOutput	Other UI components
verbatimTextOutput	Text (code)

renderDataTable
renderUI
renderImage
renderPlot
renderTable
renderText
renderUI
renderText

Outputs - Additional Widgets

- Charts
  - ggplot2
  - Plotly
  - eCharts4r
- Maps
  - leaflet
- Tables
  - DT
  - reactable
  - rhandsontable





Appsilon

Outputs - plotOutput

```
1.  library(shiny)
2.  library(ggplot2)
3.  library(ggthemes)
4.  library(forcats)
5.
6.  source("examples/utils.R")
7.  cheeses <- read.csv("data/cheeses.csv")
8.
9.  ui <- fluidPage(
10.  plotOutput(outputId = "my_chart")
11. )
12.
13. server <- function(input, output, session) {
14.  output$my_chart <- renderPlot({
15.    ggplot(data = cheeses) +
16.      geom_bar(mapping = aes(x = fct_infreq(color))) +
17.      theme_bw() +
18.      theme(axis.text.x = element_text(angle = 45, hjust = 1)
19.      )
20.  })
21. }
22.
23. shinyApp(ui, server)
```

Appsilon

Outputs - plotOutput

```
1.  library(shiny)
2.  library(ggplot2)
3.  library(ggthemes)
4.  library(forcats)
5.
6.  source("examples/utils.R")
7.  cheeses <- read.csv("data/cheeses.csv")
8.
9.  ui <- fluidPage(
10.  plotOutput(outputId = "my_chart")
11. )
12.
13. server <- function(input, output, session) {
14.  output$my_chart <- renderPlot({
15.    ggplot(data = cheeses) +
16.      geom_bar(mapping = aes(x = fct_infreq(color))) +
17.      theme_bw() +
18.      theme(axis.text.x = element_text(angle = 45, hjust = 1)
19.      )
20.  })
21. }
22.
23. shinyApp(ui, server)
```

- **outputId**: Must be unique!!!

examples/01_first_steps/04_outputs/01_plotOutput.R

Appsilon

Outputs - plotOutput

```
1.  library(shiny)
2.  library(ggplot2)
3.  library(ggthemes)
4.  library(forcats)
5.
6.  source("examples/utils.R")
7.  cheeses <- read.csv("data/cheeses.csv")
8.
9.  ui <- fluidPage(
10.  plotOutput(outputId = "my_chart")
11.  )
12.
13.  server <- function(input, output, session) {
14.   output$my_chart <- renderPlot({
15.     ggplot(data = cheeses) +
16.       geom_bar(mapping = aes(x = fct_infreq(color))) +
17.       theme_bw() +
18.       theme(axis.text.x = element_text(angle = 45, hjust = 1)
19.       )
20.   })
21.  }
22.
23.  shinyApp(ui, server)
```

- **outputId**: Must be unique!!!
- **renderPlot**: This function connects with plotOutput and then the plot is rendered

examples/01_first_steps/04_outputs/02_echarts4rOutput.R

Appsilon

Outputs - echarts4R

```
1.  library(shiny)
2.  library(echarts4r)
3.  library(dplyr)
4.
5.  cheeses <- read.csv("data/cheeses.csv")
6.  ui <- fluidPage(
7.    echarts4rOutput(outputId = "my_chart")
8.  )
9.
10. server <- function(input, output, session) {
11.   output$my_chart <- renderEcharts4r({
12.     cheeses |> count(color) |> arrange(desc(n)) |> na.omit() |>
13.       e_charts(color) |>
14.       e_bar(n) |>
15.       e_x_axis(
16.         name = "Color",
17.         axisLabel = list(rotate = 45)
18.       )
19.   })
20. }
21.
22. shinyApp(ui, server)
```

- **echarts4rOutput:** Declare a echarts4r output
- **renderEcharts4r:** Data handling to generate the chart

examples/01_first_steps/04_outputs/03_leafletOutput.R

Appsilon

Outputs - Leaflet

```
1.  library(shiny)
2.  library(leaflet)
3.  library(sf)
4.
5.  ui <- fluidPage(
6.   leafletOutput(
7.     outputId = "worldmap", width = "100vw", height = "100vh"
8.   )
9.  )
10.
11. # Server
12. server <- function(input, output, session) {
13. worldmap <- st_read("data/worldmap/")
14.  output$worldmap <- renderLeaflet({
15.    leaflet() |>
16.      addTiles() |>
17.      addPolygons(
18.        data = worldmap, color = "transparent", popup = ~name
19.      )
20.  })
21. }
22.
23. shinyApp(ui = ui, server = server)
```

- **leafletOutput**: Declare a leaflet output
- **renderLeaflet**: Data handling to generate the map

Hands-on

Appsilon

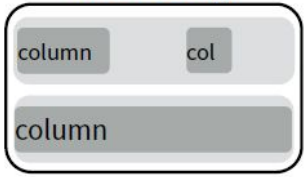
Cheeses dataset

1. *Add a table output to our current app (tableOutput)*
    a. *Customize the column names to make them more user-friendly*
    b. *Remove row names*
    c. *Add adequate spacing between rows and columns for readability*
    d. *How could it be improved?*

Appsilon

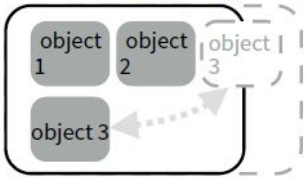
Layouts

fluidRow()



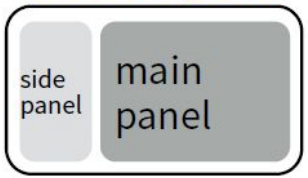

```
ui <- fluidPage(
  fluidRow(column(width = 4),
    column(width = 2,  offset = 3)),
  fluidRow(column(width = 12))
)
```

flowLayout()



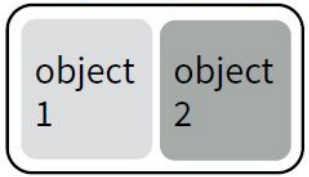

```
ui <- fluidPage(
  flowLayout( # object 1,
              # object 2,
              # object 3
  )
)
```

sidebarLayout()



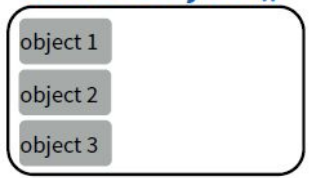

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

splitLayout()




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

verticalLayout()




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

runApp('examples/01_first_steps/05_layouts/01_tabsetPanel.R')

Appsilon

Layouts - tabsetPanel

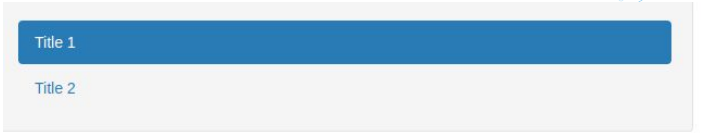
```
1.  ui <- fluidPage(
2.    tabsetPanel(
3.      tabPanel(
4.        title = "Title 1",
5.        "Tab 1"
6.      ),
7.      tabPanel(
8.        title = "Title 2",
9.        "Tab 2"
10.     )
11.   )
12. )
13.
14. server <- function(input, output) {}
15.
16. shinyApp(ui = ui, server = server)
```

runApp('examples/01_first_steps/05_layouts/02_navlistPanel.R')

Appsilon

Layouts - navlistPanel

```
1.  ui <- fluidPage(
2.    navlistPanel(
3.      tabPanel(
4.        title = "Title 1",
5.        "Tab 1"
6.      ),
7.      tabPanel(
8.        title = "Title 2",
9.        "Tab 2"
10.     )
11.   )
12. )
13.
14. server <- function(input, output) {}
15.
16. shinyApp(ui = ui, server = server)
```



runApp('examples/01_first_steps/05_layouts/03_navbarPage.R')

Appsilon

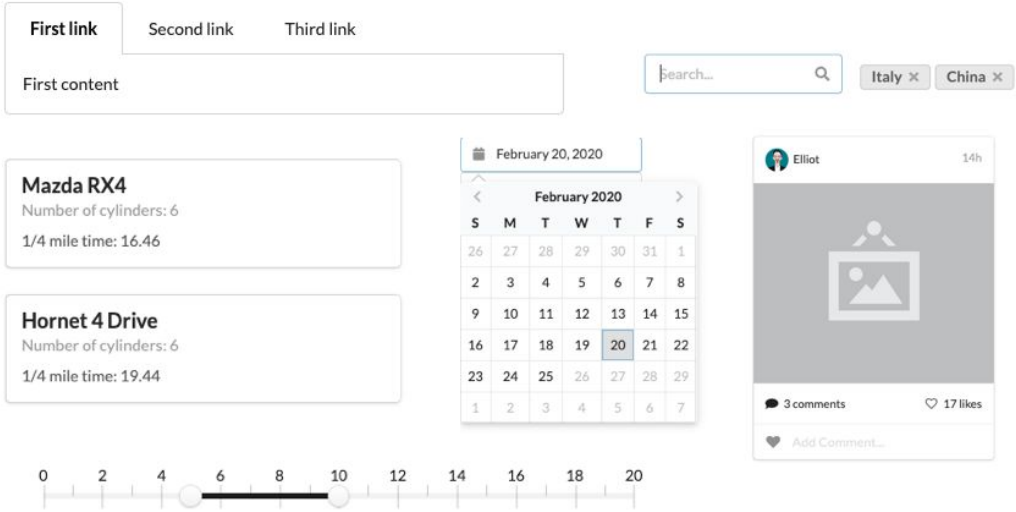
Layouts - navbarPage

```
1.  ui <- navbarPage(
2.    title = "Page",
3.    tabPanel(
4.      title = "Title 1",
5.      "Tab 1"
6.    ),
7.    tabPanel(
8.      title = "Title 2",
9.      "Tab 2"
10.   )
11. )
12.
13. server <- function(input, output) {}
14.
15. shinyApp(ui = ui, server = server)
```

Appsilon

Layouts

- shiny.semantic
- shiny.fluent
- shinybulma
- shinyMobile
- shinymaterial



Appsilon

Full example

examples/01_first_steps/06_full_example.R

Hands-on

Appsilon

Cheeses dataset

1. *Create a navbarPage*
2. *Add a tabsetPanel inside one of the tabs*
3. *Within the tabsetPanel, create two subtabs:*
    a. *One subtab should contain a table*
    b. *Another subtab should contain a map*
4. *In a separate tab (outside the tabsetPanel), add an echarts4R visualization*
5. *Set the echarts4R tab as the default selected tab*

Reactivity

Appsilon
Douglas Mesquita

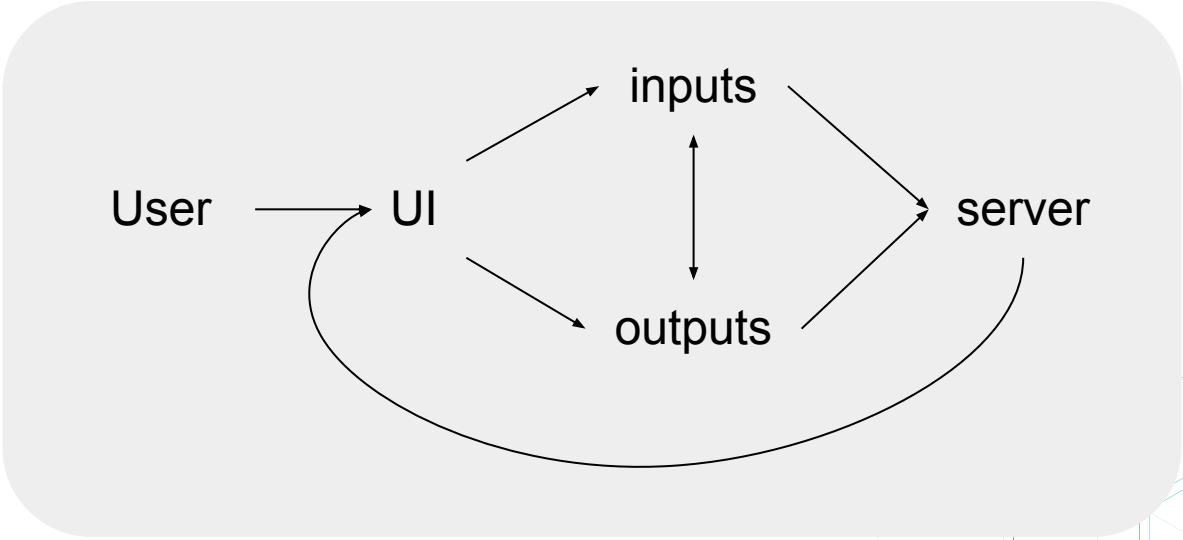
Belo Horizonte
2025-02-03 - 2025-02-06

Appsilon

Appsilon

Reactivity

We would like to update inputs and outputs based on the user interactions with the application



Appsilon

Reactivity

absolutePanel() fixedPanel() bootstrapPage() basicPage() column() conditionalPanel() fillPage() fillRow() fillCol()
fixedPage() fixedRow() fluidPage() fluidRow() helpText() icon() navbarPage() navbarMenu() navlistPanel() sidebarLayout()
sidebarPanel() mainPanel() tabPanel() tabPanelBody() tabsetPanel() titlePanel() inputPanel() flowLayout() splitLayout()
verticalLayout() wellPanel() withMathJax() actionButton() actionLink() checkboxGroupInput() checkboxInput() dateInput()
dateRangeInput() fileInput() numericInput() radioButtons() selectInput() selectizeInput() varSelectInput() varSelectizeInput()
sliderInput() animationOptions() submitButton() textInput() textAreaInput() passwordInput() updateActionButton()
updateActionLink() updateCheckboxGroupInput() updateCheckboxInput() updateDateInput() updateDateRangeInput() updateNumericInput()
updateRadioButtons() updateSelectInput() updateSelectizeInput() updateVarSelectInput() updateVarSelectizeInput()
updateSliderInput() updateTabsetPanel() updateNavbarPage() updateNavlistPanel() insertTab() prependTab() appendTab() removeTab()
showTab() hideTab() updateTextInput() updateTextAreaInput() updateQueryString() getQueryString() getUrlHash() htmlOutput()
uiOutput() imageOutput() plotOutput() outputOptions() textOutput() verbatimTextOutput() downloadButton() downloadLink()
Progress withProgress() setProgress() incProgress() modalDialog() modalButton() urlModal() showModal() removeModal()
showNotification() removeNotification() tags p() h1() h2() h3() h4() h5() h6() a() br() div() span() pre() code() img()
strong() em() hr() tag() HTML() includeHTML() includeText() includeMarkdown() includeCSS() includeScript() singleton()
is.singleton() tagList() tagAppendAttributes() tagHasAttribute() tagGetAttribute() tagAppendChild() tagAppendChildren()
tagSetChildren() tagInsertChildren() validateCssUnit() withTags() htmlTemplate() bootstrapLib() suppressDependencies()
insertUI() removeUI() markdown() renderPlot() renderCachedPlot() renderPrint() renderText() dataTableOutput() renderDataTable()
renderImage() tableOutput() renderTable() renderUI() downloadHandler() createRenderFunction() quoToFunction()
installExprFunction() reactive() is.reactive() observe() observeEvent() eventReactive() reactiveVal() reactiveValues()
bindCache() bindEvent() reactiveValuesToList() is.reactivevalues() isolate() invalidateLater() debounce() throttle() reactlog()
reactlogShow() reactlogReset() reactiveFileReader() reactivePoll() reactiveTimer() getDefaultReactiveDomain()
withReactiveDomain() onReactiveDomainEnded() freezeReactiveVal() freezeReactiveValue() runApp() runGadget() runExample()
runUrl() runGist() runGitHub() stopApp() paneViewer() dialogViewer() browserViewer() isRunning() loadSupport()
bookmarkButton() enableBookmarking() setBookmarkExclude() showBookmarkUrlModal() onBookmark() onBookmarked() onRestore()
onRestored() createWebDependency() addResourcePath() resourcePaths() removeResourcePath() registerInputHandler()
removeInputHandler() markRenderFunction() devmode() in_devmode() with_devmode() devmode_inform() register_devmode_option()
get_devmode_option() shinyAppTemplate() req() isTruthy() validate() need() session getShinyOption() shinyOptions() safeError()
onFlush() onFlushed() onSessionEnded() restoreInput() applyInputHandlers() parseQueryString() getCurrentOutputInfo() plotPNG()
sizeGrowthRatio() exportTestValues() setSerializer() snapshotExclude() snapshotPreprocessInput() snapshotPreprocessOutput()
repeatable() serverInfo() onStop() httpResponse() key_missing() is.key_missing() brushedPoints() nearPoints() brushOpts()
clickOpts() dblclickOpts() hoverOpts() NS() ns.sep moduleServer() callModule() shinyApp() shinyAppDir() shinyAppFile()
maskReactiveContext() runTests() testServer() MockShinySession markRenderFunction() shinyUI() shinyServer() exprToFunction()

Appsilon

Reactivity

Reactivity

```
checkboxGroupInput()
checkboxInput()
fileInput()
varSelectizeInput()
textInput()
varSelectInput()
dateInput()
passwordInput()
selectInput()
selectizeInput()
textAreaInput()
sliderInput()
numericInput()
dateRangeInput()
```

```
reactiveVal()
reactiveValues()
reactive()
reactiveTimer()
eventReactive()
```

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

```
observe()
observeEvent()
```

```
htmlOutput()
dataTableOutput()
tableOutput()
uiOutput()
plotOutput()
textOutput()
imageOutput()
verbatimTextOutput()
```

```
isolate()
invalidateLater()
debounce()
throttle()
bindEvent()
bindCache()
```

Reactivity

Appsilon

*Input()

reactiveVal()
reactiveValues()
reactive()
reactiveTimer()
eventReactive()

render*()

*Output()

observe()
observeEvent()

isolate()
invalidateLater()
debounce()
throttle()
bindEvent()
bindCache()

Appsilon

Reactivity

*Input()

*Output()

render*()

observe()

observeEvent()

reactiveVal()

reactiveValues()

reactive()

eventReactive()

Appsilon

Reactivity

Source

*Input()
reactiveVal()
reactiveValues()

Conductor

render*()
reactive()
eventReactive()

Endpoint

*Output()
observeEvent()
observe()

Appsilon

Reactivity

Sources can affect Endpoints directly

Conductors can affect Endpoints directly

Sources can affect Conductors

Conductors can affect other Conductors

Appsilon

Reactivity

Sources can affect Endpoints directly

Conductors can affect Endpoints directly

Sources can affect Conductors

Conductors can affect other Conductors

You can connect as many Reactive Components as you want

Appsilon

Reactivity

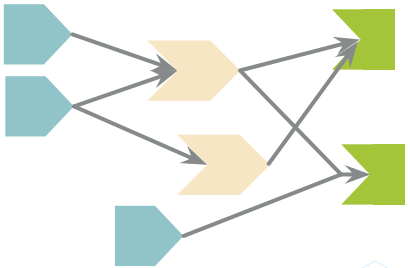
Sources can affect Endpoints directly

Conductors can affect Endpoints directly

Sources can affect Conductors

Conductors can affect other Conductors

You can connect as many Reactive Components as you want



Reactive components can have many-to-many connections

Appsilon

Reactivity

Lets focus on the following functions:

- observe
- observeEvent
- reactive
- eventReactive
- reactiveVal
- reactiveValues
- req
- Isolate
- debounce

Appsilon

Input and output objects

To access the current status of a certain input, use:

```
input$input_id
```

To update a specific output, use:

```
output$output_id <- render*(...)
```

runApp('examples/02_reactivity/01_initial/01_initial_code.R)

Appsilon

Next examples

From now on, I will omit parts of the code that do not change. Also, have in mind our basic example:

```
1.  library(shiny)
2.
3.  source("examples/utils.R")
4.
5.  cheeses <- read.csv("data/cheeses.csv")
6.  country_choices <- split_unique(cheeses$country)
7.
8.  ui <- fluidPage(
9.   selectInput(inputId = "countries", label = "Countries", choices = country_choices),
10.  tableOutput(outputId = "cheeses_table")
11. )
12.
13. server <- function(input, output, session) {
14.  output$cheeses_table <- renderTable({
15.    cheeses
16.  })
17. }
18.
19. shinyApp(ui, server)
```

Appsilon

observe

observe will watch all reactive elements inside an expression and will be triggered every time something changes:

```
1.  library(dplyr)
2.
3.  server <- function(input, output, session) {
4.   observe({
5.     country_sel <- input$countries
6.
7.     output$cheeses_table <- renderTable({
8.       cheeses |>
9.         filter(grepl(x = country, pattern = country_sel))
10.    })
11.  })
12.  }
```

Appsilon

observe

observe will watch all reactive elements inside an expression and will be triggered every time something changes:

```
1.  library(dplyr)
2.
3.  server <- function(input, output, session) {
4.   observe({
5.     country_sel <- input$countries
6.
7.     output$cheeses_table <- renderTable({
8.       cheeses |>
9.         filter(grepl(x = country, pattern = country_sel))
10.    })
11.  })
12.  }
```

Observe will trigger when input$countries change

examples/02_reactivity/02_observe/02_observe.R

Appsilon

observe

observe will watch all reactive elements inside an expression and will be triggered every time something changes:

```
1.  library(dplyr)
2.
3.  server <- function(input, output, session) {
4.   observe({
5.     country_sel <- input$countries
6.
7.     output$cheeses_table <- renderTable({
8.       cheeses |>
9.         filter(grepl(x = country, pattern = country_sel))
10.    })
11.  })
12.  }
```

Output will be updated (filtering the table)

Appsilon

observeEvent

observeEvent will watch specific reactive elements and will be triggered only when at least one of them changes

examples/02_reactivity/02_observe/02_observeEvent.R

Appsilon

observeEvent

```
1.  cheeses <- read.csv("data/cheeses.csv")
2.  country_choices <- split_unique(cheeses$country)
3.  type_choices <- split_unique(cheeses$type)
4.
5.  ui <- fluidPage(
6.   selectInput(inputId = "countries", label = "Countries", choices = country_choices),
7.   selectInput(inputId = "type", label = "Type", choices = type_choices),
8.   tableOutput(outputId = "cheeses_table")
9.  )
10.
11. server <- function(input, output, session) {
12.   observeEvent(input$countries, {
13.     country_sel <- input$countries
14.     type_sel <- input$type
15.
16.     output$cheeses_table <- renderTable({
17.       cheeses |>
18.         filter(
19.           grepl(x = country, pattern = country_sel),
20.           grepl(x = type, pattern = type_sel)
21.         )
22.     })
23.   })
24. }
```

Table is updated only when country changes

examples/02_reactivity/02_observe/02_observeEvent_multiple.R

Appsilon

observeEvent

```
1.  server <- function(input, output, session) {
2.    observeEvent({
3.      input$countries
4.      input$type
5.    }, {
6.      country_sel <- input$countries
7.      type_sel <- input$type
8.
9.      output$cheeses_table <- renderTable({
10.       cheeses |>
11.         filter(
12.           grepl(x = country, pattern = country_sel),
13.           grepl(x = type, pattern = type_sel)
14.         )
15.     })
16.   })
17. }
```

Now table is updated every time one of the two inputs changes

Appsilon

reactive

reactive will be triggered only when explicitly called

```
server <- function(input, output, session) {
 filtered_data <- reactive({
   country_sel <- input$countries
   type_sel <- input$type

   cheeses |>
     filter(
       grepl(x = country, pattern = country_sel),
       grepl(x = type, pattern = type_sel)
     )
 })

 output$cheeses_table <- renderTable({
   filtered_data()
 })
}
```

Appsilon

reactive

reactive will be triggered only when explicitly called

```
1.  server <- function(input, output, session) {
2.    filtered_data <- reactive({
3.      country_sel <- input$countries
4.      type_sel <- input$type
5.
6.      cheeses |>
7.        filter(
8.          grepl(x = country, pattern = country_sel),
9.          grepl(x = type, pattern = type_sel)
10.       )
11.   })
12.
13.   output$cheeses_table <- renderTable({
14.     filtered_data()
15.   })
16. }
```

Give a name in order to call a reactive

examples/02_reactivity/03_reactive/01_reactive.R

Appsilon

reactive

reactive will be triggered only when explicitly called

```
 1.  server <- function(input, output, session) {
 2.    filtered_data <- reactive({
 3.      country_sel <- input$countries
 4.      type_sel <- input$type
 5.
 6.      cheeses |>
 7.        filter(
 8.          grepl(x = country, pattern = country_sel),
 9.          grepl(x = type, pattern = type_sel)
10.        )
11.    })
12.
13.    output$cheeses_table <- renderTable({
14.      filtered_data()
15.    })
16.  }
```

Call the reactive by using *reactive_name()*

examples/02_reactivity/03_reactive/02_eventReactive.R

Appsilon

eventReactive

eventReactive triggers only when called and if the specific events have changed

```
1.  ui <- fluidPage(
2.    selectInput(inputId = "countries", label = "Countries", choices = country_choices),
3.    selectInput(inputId = "type", label = "Type", choices = type_choices),
4.    actionButton(inputId = "apply_filters", label = "Apply Filters"),
5.    tableOutput(outputId = "cheeses_table")
6.  )
7.
8.  server <- function(input, output, session) {
9.   filtered_data <- eventReactive(input$apply_filters, {
10.    country_sel <- input$countries
11.    type_sel <- input$type
12.
13.    <same filter here>
14.   })
15.
16.   output$cheeses_table <- renderTable({
17.    filtered_data()
18.   })
19. }
```

Triggers only when apply_filters is clicked

examples/02_reactivity/03_reactive/03_reactive_vs_function.R

Appsilon

Why to use reactives instead of functions?

A reactive will only be evaluated when the input change. A function will always be executed again.

```
1.  filtered_data <- function(country_sel, type_sel) {
2.    Sys.sleep(5)
3.    cheeses |>
4.      filter(
5.        grepl(x = country, pattern = country_sel),
6.        grepl(x = type, pattern = type_sel)
7.      )
8.  }
9.
10. server <- function(input, output, session) {
11.   observeEvent(input$apply_filters, {
12.     country_sel <- input$countries
13.     type_sel <- input$type
14.
15.     output$cheeses_table <- renderTable({
16.       filtered_data(country_sel, type_sel)
17.     })
18.   })
19. }
```

The function will be executed every time apply_filter is clicked

examples/02_reactivity/03_reactive/03_reactive_vs_function.R

Appsilon

Why to use reactives instead of functions?

A reactive will only be evaluated when the input change. A function will always be executed again.

```
1.  server <- function(input, output, session) {
2.    filtered_data <- reactive({
3.      Sys.sleep(5)
4.      country_sel <- input$countries
5.      type_sel <- input$type
6.
7.      cheeses |>
8.        filter(
9.          grepl(x = country, pattern = country_sel),
10.         grepl(x = type, pattern = type_sel)
11.       )
12.   })
13.
14.   output$cheeses_table <- renderTable({
15.     input$apply_filters
16.     filtered_data()
17.   })
18. }
```

If inputs did not change, the reactive will be executed only once

Appsilon

reactiveVal

What if you want to change manually the values of a reactive? A: use reactiveVal or reactiveValues

```
server <- function(input, output, session) {
 counter <- reactiveVal(0)

 filtered_data <- eventReactive(input$apply_filters, {
   counter(counter() + 1)

   {same as before}
 })

 observeEvent(counter(), {
   if (counter() == 5)
     showNotification(ui = "You already clicked here 5 times :)")
 })

 output$cheeses_table <- renderTable({
   filtered_data()
 })
}
```

Appsilon

reactiveVal

What if you want to change manually the values of a reactive? A: use reactiveVal or reactiveValues

```
1.  server <- function(input, output, session) {
2.    counter <- reactiveVal(0)
3.
4.    filtered_data <- eventReactive(input$apply_filters, {
5.      counter(counter() + 1)
6.
7.      {same as before}
8.    })
9.
10.   observeEvent(counter(), {
11.     if (counter() == 5)
12.       showNotification(ui = "You already clicked here 5 times :)")
13.   })
14.
15.   output$cheeses_table <- renderTable({
16.     filtered_data()
17.   })
18. }
```

Initialize the reactiveVal

Appsilon

reactiveVal

What if you want to change manually the values of a reactive? A: use reactiveVal or reactiveValues

```
server <- function(input, output, session) {
 counter <- reactiveVal(0)

 filtered_data <- eventReactive(input$apply_filters, {
   counter(counter() + 1)

   {same as before}
 })

 observeEvent(counter(), {
   if (counter() == 5)
     showNotification(ui = "You already clicked here 5 times :)")
 })

 output$cheeses_table <- renderTable({
   filtered_data()
 })
}
```

Increment the counter every time apply_filters is clicked

examples/02_reactivity/03_reactive/04_reactiveVal.R

Appsilon

reactiveVal

What if you want to change manually the values of a reactive? A: use reactiveVal or reactiveValues

```
1.  server <- function(input, output, session) {
2.    counter <- reactiveVal(0)
3.
4.    filtered_data <- eventReactive(input$apply_filters, {
5.      counter(counter() + 1)
6.
7.      {same as before}
8.    })
9.
10.   observeEvent(counter(), {
11.     if (counter() == 5)
12.       showNotification(ui = "You already clicked here 5 times :)")
13.   })
14.
15.   output$cheeses_table <- renderTable({
16.     filtered_data()
17.   })
18. }
```

Show a message when the user reach 5 clicks

Appsilon

reactiveValues

Sometimes you need more to store more than one information

```
1.  user_usage <- reactiveValues(
2.   counter = 0,
3.   history = data.frame(
4.     counter = numeric(0),
5.     country = character(0),
6.     type = character(0)
7.   )
8.  )
```

Each entry are going to be reactive

Appsilon

reactiveValues

Sometimes you need more to store more than one information

```
1.  filtered_data <-
    eventReactive(input$apply_filters, {
2.    country_sel <- input$countries
3.    type_sel <- input$type
4.
5.    user_usage$counter <- user_usage$counter + 1
6.
7.    cheeses |>
8.      filter(
9.        grepl(x = country, pattern = country_sel),
10.       grepl(x = type, pattern = type_sel)
11.     )
12. })
```

Easier to update!

Appsilon

reactiveValues

Sometimes you need more to store more than one information

```
observeEvent(c(input$countries, input$type), {
 user_usage$history <- rbind(
   user_usage$history,
   data.frame(
     counter = user_usage$counter,
     country = input$countries,
     type = input$type
   )
 )
})
```

Easier to update!

examples/02_reactivity/03_reactive/05_reactiveValues.R

Appsilon

reactiveValues

Sometimes you need more to store more than one information

```
1.  observeEvent(user_usage$history, {
2.   print(user_usage$history)
3.  })
4.
5.  output$cheeses_table <- renderTable({
6.   filtered_data()
7.  })
```

You can observe a specific element!

Appsilon

req

Useful when you want to avoid running code when some inputs are not ready

```
server <- function(input, output, session) {
 filtered_data <- reactive({
   country_sel <- input$countries
   type_sel <- input$type

   cheeses |>
     filter(
       grepl(x = country, pattern = country_sel),
       grepl(x = type, pattern = type_sel)
     )
 })

 output$cheeses_table <- renderTable({
   req(nrow(filtered_data()) > 0)

   filtered_data()
 })
}
```

Appsilon

req

Useful when you want to avoid running code when some inputs are not ready

```
1.  server <- function(input, output, session) {
2.   filtered_data <- reactive({
3.     country_sel <- input$countries
4.     type_sel <- input$type
5.
6.     cheeses |>
7.       filter(
8.         grepl(x = country, pattern = country_sel),
9.         grepl(x = type, pattern = type_sel)
10.      )
11.   })
12.
13.   output$cheeses_table <- renderTable({
14.     req(nrow(filtered_data()) > 0)
15.
16.     filtered_data()
17.   })
18.  }
```

Table will be displayed only in case data is available

examples/02_reactivity/04_others/02_isolate.R

Appsilon

isolate

Useful when you want to use some reactive element but do not want to trigger any update when doing it

```
1.  filtered_data <- reactive({
2.    country_sel <- input$countries
3.    type_sel <- input$type
4.
5.    i <- isolate({ counter() })
6.    counter(i + 1)
7.    print(i)
8.
9.    cheeses |>
10.     filter(
11.       grepl(x = country, pattern = country_sel),
12.       grepl(x = type, pattern = type_sel)
13.     )
14.  })
```

reactive will not update when counter() is updated

Appsilon

debounce

Useful when you want to wait for some time before executing a task

```
1.  filtered_data_debounced <- debounce(
2.   r = filtered_data,
3.   millis = 2000
4.  )
5.
6.  output$cheeses_table <- renderTable({
7.   req(nrow(filtered_data_debounced()) > 0)
8.
9.   filtered_data_debounced()
10. })
```

2s after filtered data is invalidated, *filtered_data_debounced* will be invalidated

examples/02_reactivity/04_others/03_debounce.R

Appsilon

debounce

Useful when you want to wait for some time before executing a task

```
 1.   filtered_data_debounced <- debounce(
 2.    r = filtered_data,
 3.    millis = 2000
 4.   )
 5.
 6.   output$cheeses_table <- renderTable({
 7.    req(nrow(filtered_data_debounced()) > 0)
 8.
 9.    filtered_data_debounced()
10.   })
```

observe *filtered_data_debounced*

Appsilon

Updating inputs

Very often we want to update inputs based on what the usage of the app

actionButton Action Button	updateActionButton
checkboxGroupInput A group of check boxes	updateCheckboxGroupInput
checkboxInput A single check box	updateCheckboxInput
dateInput A calendar to aid date selection	updateDateInput
dateRangeInput A pair of calendars (date range)	updateDateRangeInput
fileInput A file upload control wizard	
helpText Help text that can be added to an input form	
numericInput A field to enter numbers	updateNumericInput
radioButtons A set of radio buttons	updateRadioButtons
selectInput A box with choices to select from	updateSelectInput
sliderInput A slider bar	updateSliderInput
submitButton A submit button	
textInput A field to enter text	updateTextInput

Appsilon

Updating inputs

Very often we want to update inputs based on what the usage of the app

```
observeEvent(input$countries, {
  type_choices <- cheeses |>
    filter(grepl(x = country, pattern = input$countries)) |>
    pull(type) |>
    split_unique()

  updateSelectInput(
    session = session,
    inputId = "type",
    choices = type_choices
  )
})
```

observe *input$countries*

examples/02_reactivity/04_others/04_updating_inputs.R

Appsilon

Updating inputs

Very often we want to update inputs based on what the usage of the app

```
observeEvent(input$countries, {
 type_choices <- cheeses |>
   filter(grepl(x = country, pattern = input$countries))
|>
   pull(type) |>
   split_unique()

 updateSelectInput(
   session = session,
   inputId = "type",
   choices = type_choices
 )
})
```

update *input$type* with filtered options

Hands-on

Appsilon

Cheeses dataset

1. *Create a reactiveValues object to track how many times each input changes:*
   a. *Type, Color, Flavor, Aroma, Vegetarian, Vegan, Cheese*
2. *Replace the current table output with a DT table:*
   a. *https://rstudio.github.io/DT/shiny.html*
3. *Detect row clicks in the table (input$<id>_rows_selected) and*
   a. *Debounce them by 3s*
   b. *Display a modal with the selected cheese name as a title*
   c. *Create a reactive function to retrieve the cheese's photo (if available) using get_cheese_img from utils.R*
   d. *Show the retrieved image inside the modal*

Debugging Techniques

Appsilon
Douglas Mesquita

Belo Horizonte
2025-02-03 - 2025-02-06

Appsilon

Appsilon

Debugging Techniques

Survey: Fixing Bugs Stealing Time from Development

BY: MIKE VIZARD ON FEBRUARY 16, 2021 — 4 COMMENTS

A global survey of 950 developers published today finds more than a third (38%) of developers spend up to a quarter of their time fixing software bugs, with slightly more than a quarter (26%) spending up to half their time fixing bugs.

Nearly two-thirds of all developers said they would rather do an unpleasant activity than fix errors, including pay bills (26%), go to the dentist (21%) and spend time with in-laws (20%).

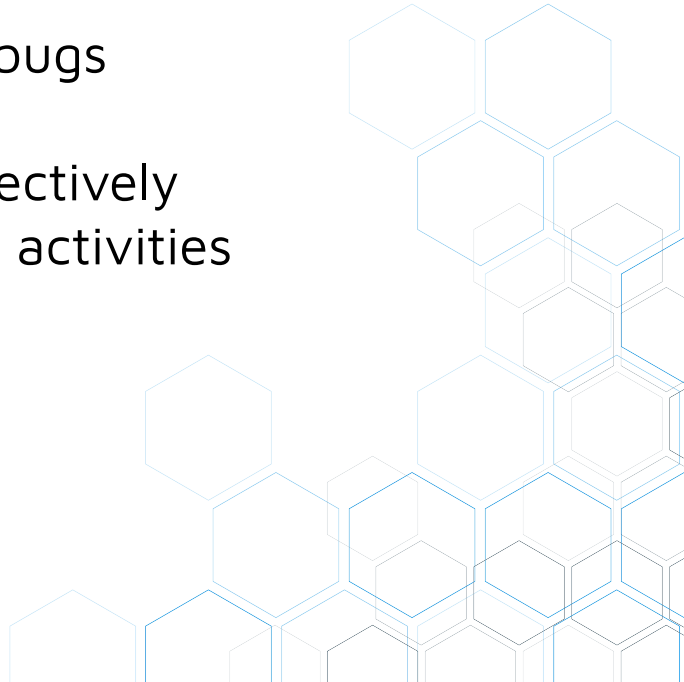
https://www.businesswire.com/news/home/20210216005484/en/Rollbar-Research-Shows-That-Traditional-Error-Monitoring-Is-Missing-the-Mark

Appsilon

Debugging Techniques

Most of us don’t like fixing bugs

Let’s see how we can do it effectively
and save time for more pleasant activities



Appsilon

Debugging Techniques

Interactive

Breakpoints, interactive code evaluation, step by step

Non-Interactive

Logs inspection after a code run

Appsilon

When to debug?

If a bug breaks the app, we need to get to a place just before where it crashes

If a bug doesn’t break the app, but works unexpectedly, we don’t have an obvious destination to find

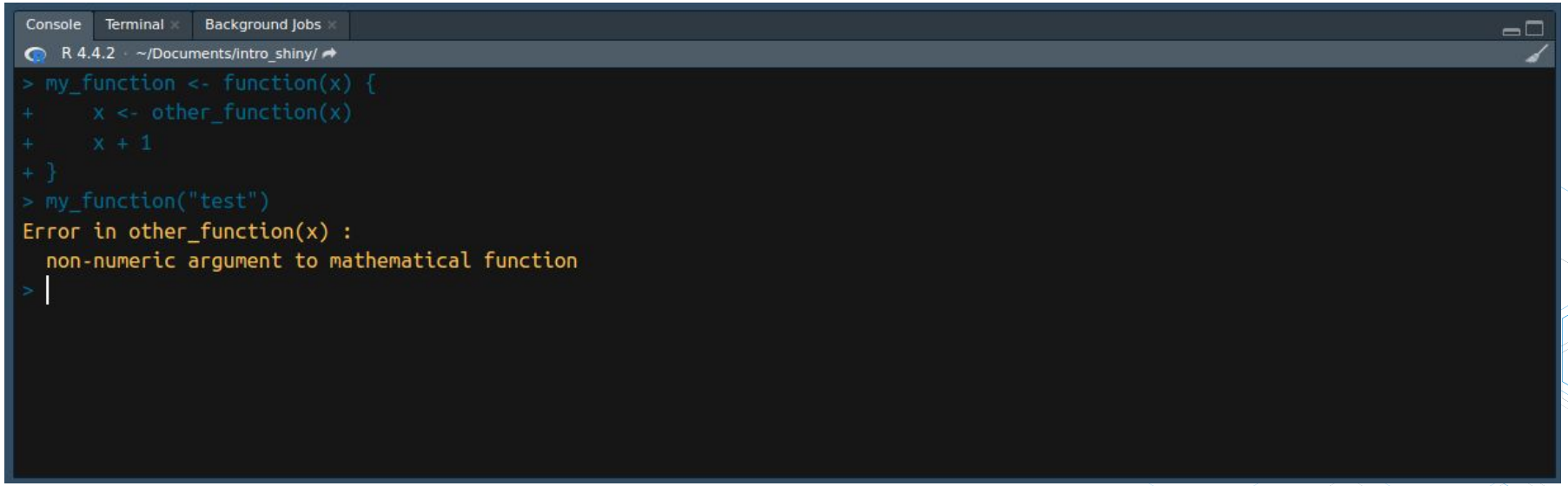
Appsilon

When to debug?

When the code crashes

Appsilon

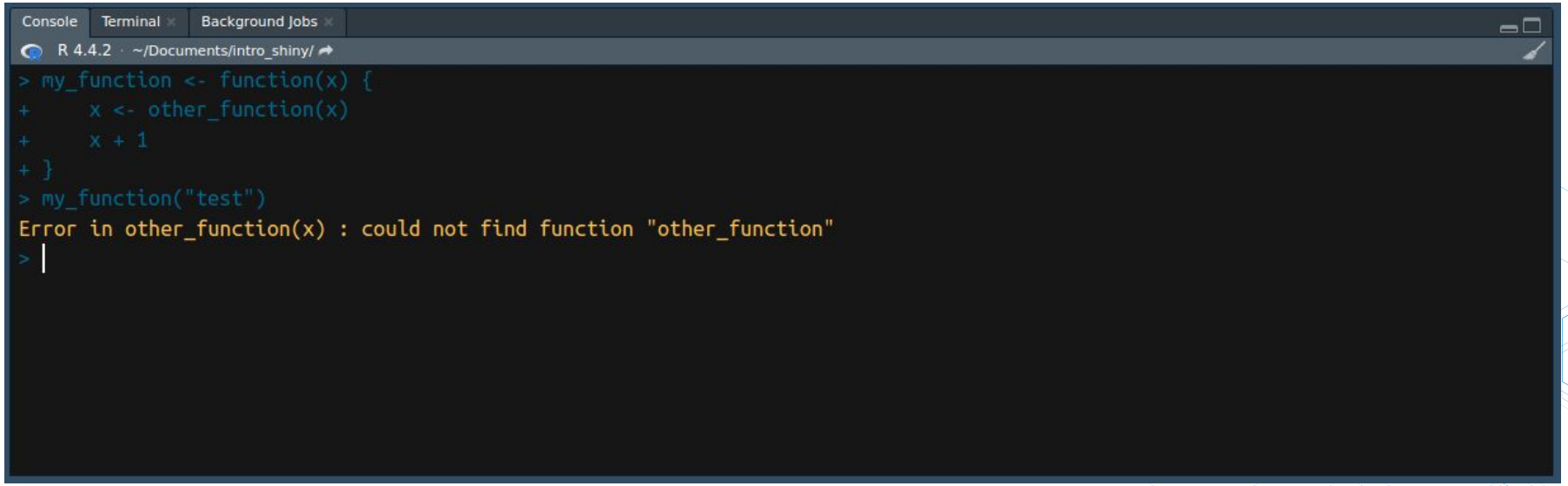
When the code crashes



Some errors are quite easy to spot

Appsilon

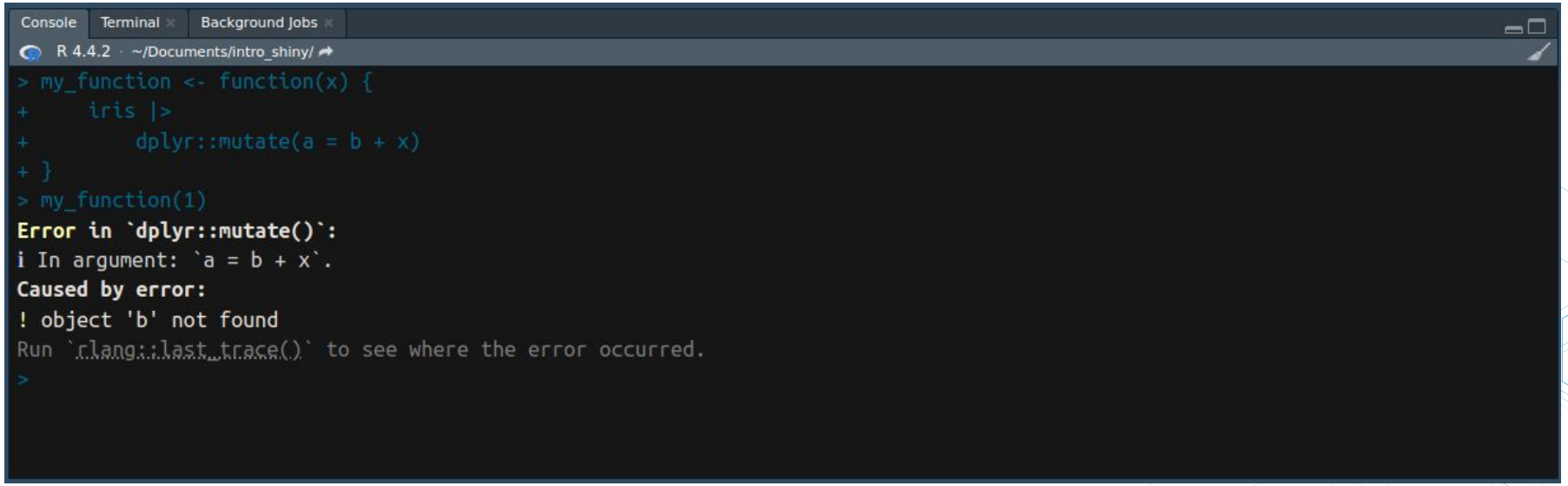
When the code crashes



Some are obvious

Appsilon

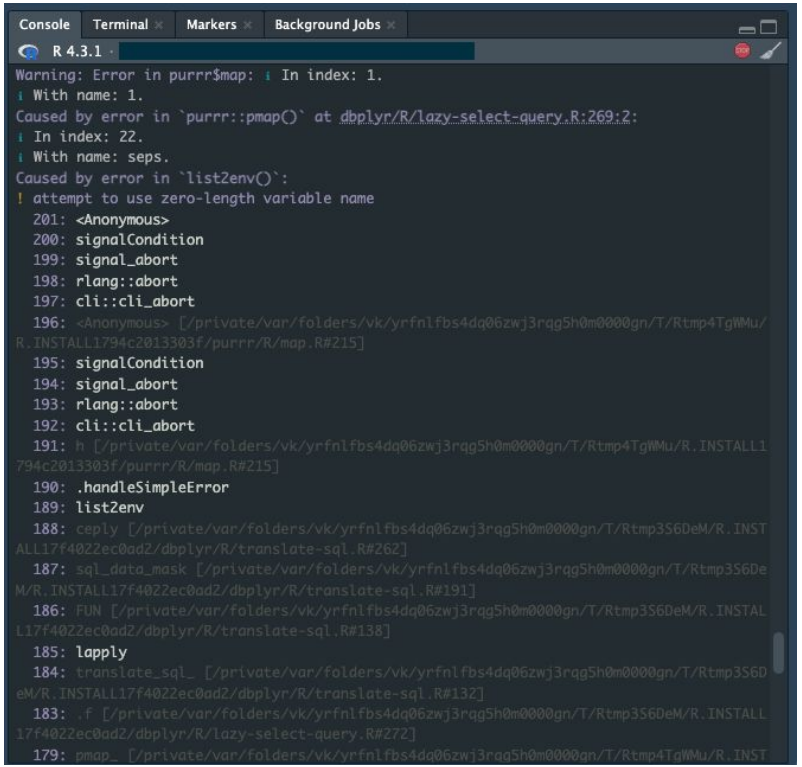
When the code crashes



Some are very informative and nicely formatted

Appsilon

When the code crashes



Unfortunately, often they're just a mess

Appsilon

Where to start

- Ask AI :)
- Search on the internet: Google, Stack Overflow, GitHub
  - It can straight away show you what was wrong
  - With time, you'll start to associate certain types of errors with their causes

- Recall if the error could be caused by recent changes
  Has it happened before? Roll-back changes to confirm

- If it was done by a recent change, the search scope is limited
- If it's not, we have a broader surface to inspect

Appsilon

When to debug?

When the code doesn’t work as expected

Appsilon

Lack of an error message

A plot may look off

Results in a table may not looks as you expect

We can use the same tools and approach as finding the place that throws an error

But we need to know what constitutes the “off” behavior and what is the expected outcome

Appsilon

Bug prevention

R is an interpreted language – most bugs can only be caught after the code is run

Static code analysis with tools like https://lintr.r-lib.org/ can help spot only the most obvious mistakes

Discovering and addressing problems in the code is crucial before releasing a change, via:

- Manual checks
- Automated checks – tests

Appsilon

Defensive programming

A good way to prevent common errors is validating inputs

If inputs are invalid, we'll get an informative message why it happened and where

R doesn't support types, it opens doors to improper usage of functions

We can catch these types of errors with assertions:

- https://mllg.github.io/checkmate/
- https://docs.ropensci.org/assertr/
- Or base R `stopifnot`
- Or `rlang::abort`

examples/02_reactivity/03_debugging/01_checkmate.R

Appsilon

checkmate

```
1.  library(checkmate)
2.  library(dplyr)
3.
4.  my_function <- function(df, at, add) {
5.   assert_data_frame(df)
6.   assert_string(at)
7.   assert_choice(at, colnames(df))
8.   assert_numeric(add)
9.
10.  df |>
11.    mutate(across(all_of(at), ~ .x + add))
12.  }
13.
14.  my_function(list(x = 1), "mpg", "a")
15.  my_function(mtcars, "mpg", "a")
16.  my_function(mtcars, "x", 1)
```

Debugging: The server side

Appsilon

Messages

We can put printing statements that inform about what's happening in the code: `print`, `message`, `cat`, `logger::log_debug` (or other)

This technique can be especially powerful when logging is consistently added to the app, indicating what is happening.

If non-interactive logging doesn't answer what broke the code, it can significantly narrow-down the scope where to debug interactively.

Appsilon

print()

```
1.  server <- function(input, output, session) {
2.   filtered_data <- reactive({
3.     filtered_data <<- cheeses |>
4.       filter(grepl(x = country, pattern = input$countries))
5.     return(filtered_data)
6.   })
7.
8.   output$cheeses_table <- renderTable({
9.       print("step 1")
10.      res <- filtered_data()
11.      print("step 2")
12.      return(res)
13.  })
14.  }
```

Add print statements in order to check where the problem is happening

examples/02_reactivity/03_debugging/02_print.R

Appsilon

print()

```
> shinyApp(ui, server)

Listening on http://127.0.0.1:4917
[1] "step 1"
[1] "step 2"
[1] "step 1"
Warning: Error in filtered_data: could not find function "filtered_data"
  103: renderTable [#10]
  102: func
   89: renderFunc
   88: output$cheeses_table
    3: runApp
    2: print.shiny.appobj
    1: <Anonymous>
```

examples/02_reactivity/03_debugging/03_browser.R

Appsilon

browser()

```
1.  server <- function(input, output, session) {
2.   filtered_data <- reactive({
3.     browser()
4.     filtered_data <<- cheeses |>
5.       filter(grepl(x = country, pattern = input$countries))
6.     return(filtered_data)
7.   })
8.
9.   output$cheeses_table <- renderTable({
10.    browser()
11.    filtered_data()
12.  })
13.  }
```

Console will be enabled when the flow reach this part of the code (everytime)

Appsilon

Alternatives to browser()

- `debug(fun)`:
  - Inserts a browser statement on the first line of the provided function.
  - Use `undebug(fun)` to remove it.
- `debugonce(fun)`:
  - Runs a `debug` on given function only once.

Those functions can be useful when you already have a good idea that there's a bug in a scope of the selected function.

It makes it easy to step into a function within a loop.

examples/02_reactivity/03_debugging/04_fullstack_trace.R

Appsilon

Full stack trace

```
options(shiny.fullstacktrace = TRUE)

server <- function(input, output, session) {
 filtered_data <- reactive({
   filtered_data <<- cheeses |>
     filter(grepl(x = country, pattern =
input$countries))
   return(filtered_data)
 })

 output$cheeses_table <- renderTable({
   filtered_data()
 })
}
```

Error message will be more verbose

Appsilon

Reactive log

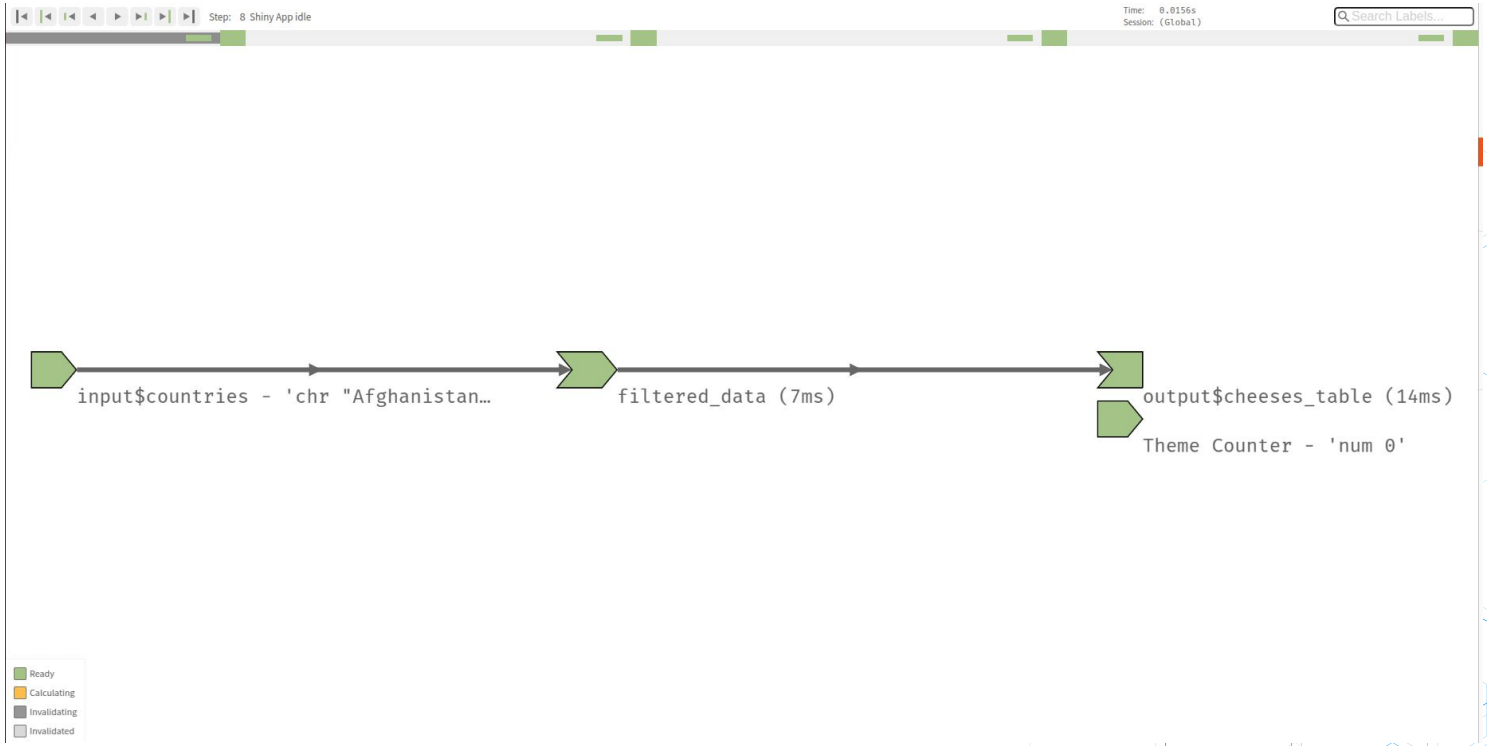
```
1.  options(shiny.reactlog = TRUE)
2.
3.  server <- function(input, output, session) {
4.   filtered_data <- reactive({
5.     filtered_data <- cheeses |>
6.       filter(grepl(x = country, pattern =
    input$countries))
7.     return(filtered_data)
8.   })
9.
10.  output$cheeses_table <- renderTable({
11.    filtered_data()
12.  })
13. }
14.
15. shinyApp(ui, server)
```

Run app and after some time press CTRL + F3

examples/02_reactivity/03_debugging/05_reaclog.R

Appsilon

Reactive log



Debugging: The browser side

Appsilon

Appsilon

debugger

debugger = browser()

- debugger stops JS code similarly to browser(), by letting you dive into the context of where it is called
- It will only trigger if the browser console is open

Appsilon

console.log

console.log(“message”) = print(“message”)

- Prints a message into the browser console.

examples/02_reactivity/03_debugging/08_showLog.R

Appsilon

R based Solutions - {shinyjs}

- {shinyjs} offers an option of logging console.log functions
- Can be done by calling showLog() on your server function
- Only supports console.log calls

https://rdrr.io/cran/shinyjs/man/showLog.html

Hands-on

Appsilon

Cheeses dataset

1. *Enable the Shiny reactlog by setting options(shiny.reactlog = TRUE) in our current example. Run the app and open the reactive graph using CTRL + F3. Analyze the graph dependencies to:*
    a. *Identify which reactive expressions trigger others*
    b. *Detect any unnecessary recalculations, if present*
2. *Do the same for*
    a. *shiny::runGitHub("rstudio/shiny-examples", subdir="063-superzip-example")*

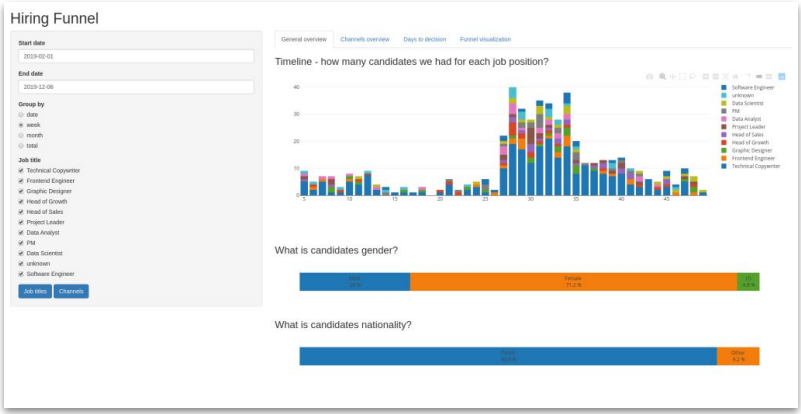
UI/UX in Shiny

Appsilon
Douglas Mesquita

Belo Horizonte
2025-02-03 - 2025-02-06

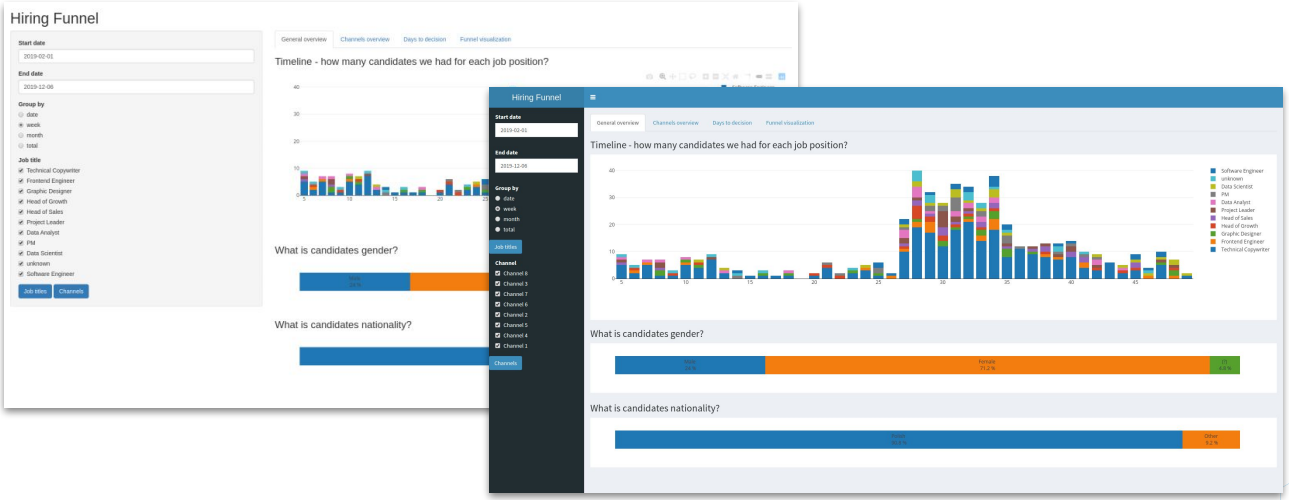
Appsilon

Goals



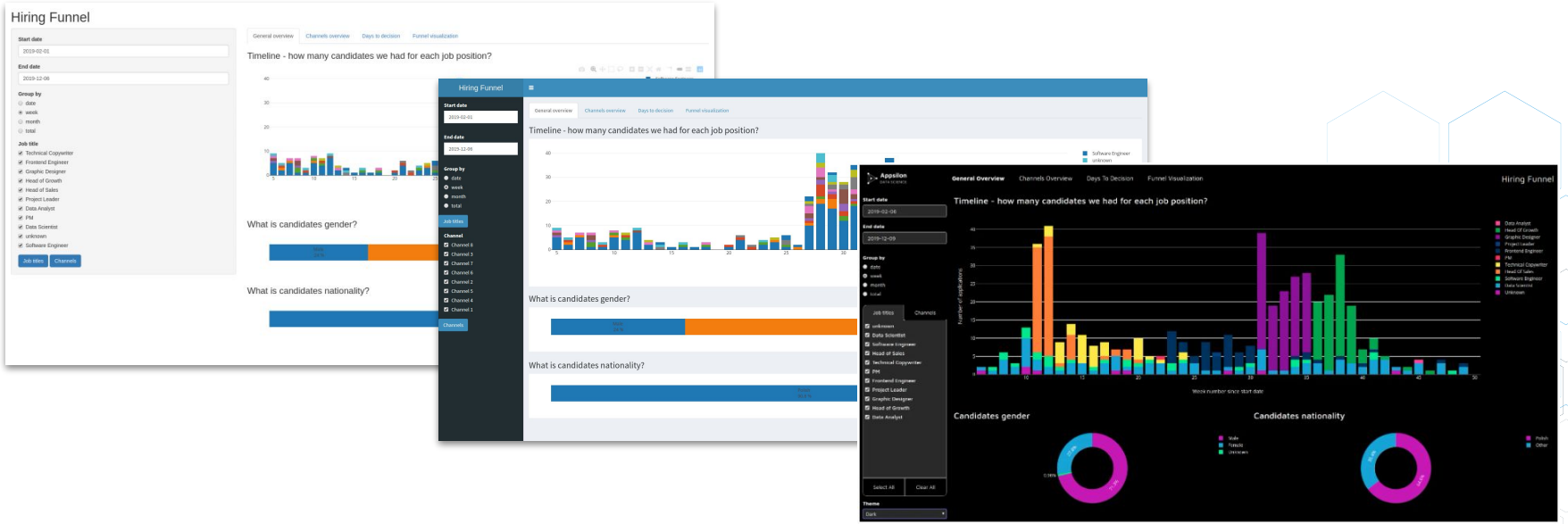
https://appsilon.com/journey-from-basic-prototype-to-production-ready-shiny-dashboard/

Goals



https://appsilon.com/journey-from-basic-prototype-to-production-ready-shiny-dashboard/

Goals



https://appsilon.com/journey-from-basic-prototype-to-production-ready-shiny-dashboard/

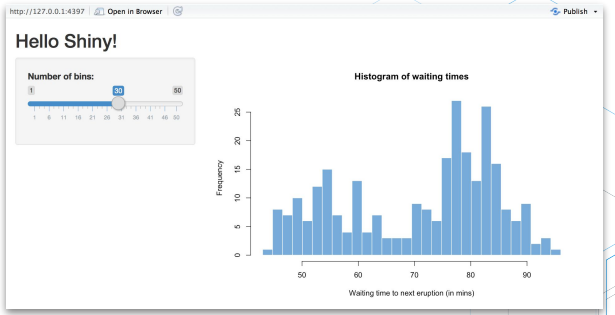
Why standing out matters

Appsilon

Appsilon

Why standing out matters

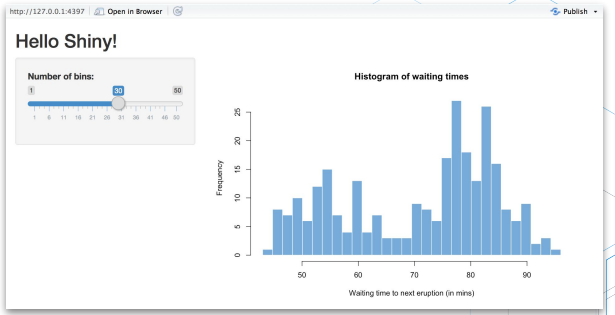
- The way your **application looks and responds** is your **first point of contact with users**.
  -
- **Users don't know** (and often don't care) about what is under the hood of your application or how its written.
  -
- Users need to feel **engaged** and **comfortable** when they visit your application. Frustration can be the difference between **success and failure**.



Appsilon

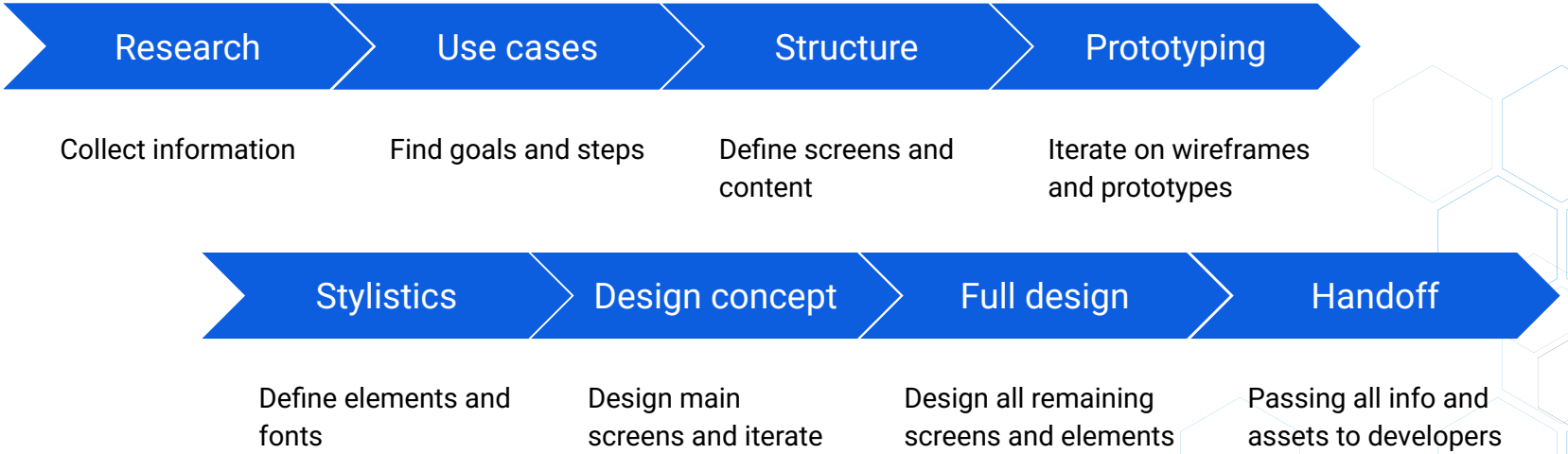
Why standing out matters

- The way your **application looks and responds** is your **first point of contact with users**.
  -
- **Users don't know** (and often don't care) about what is under the hood of your application or how its written.
  -
- Users need to feel **engaged** and **comfortable** when they visit your application. Frustration can be the difference between **success and failure**.
  -
- **<u>Commonly referred to as UI/UX</u>**



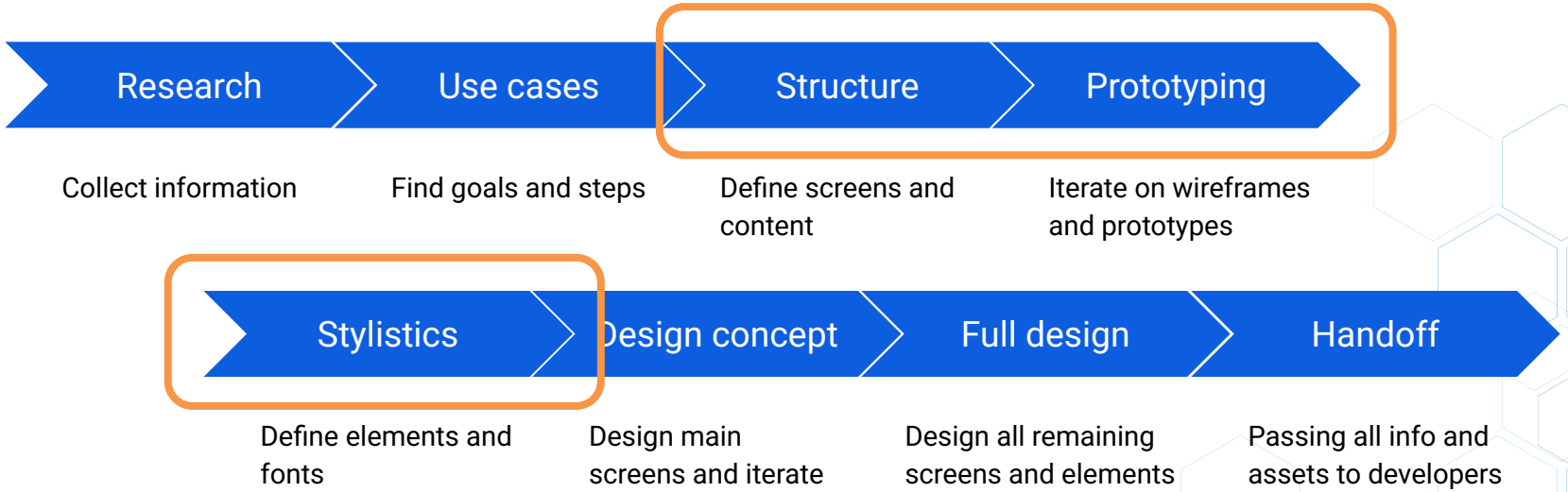
Appsilon

UI/UX process



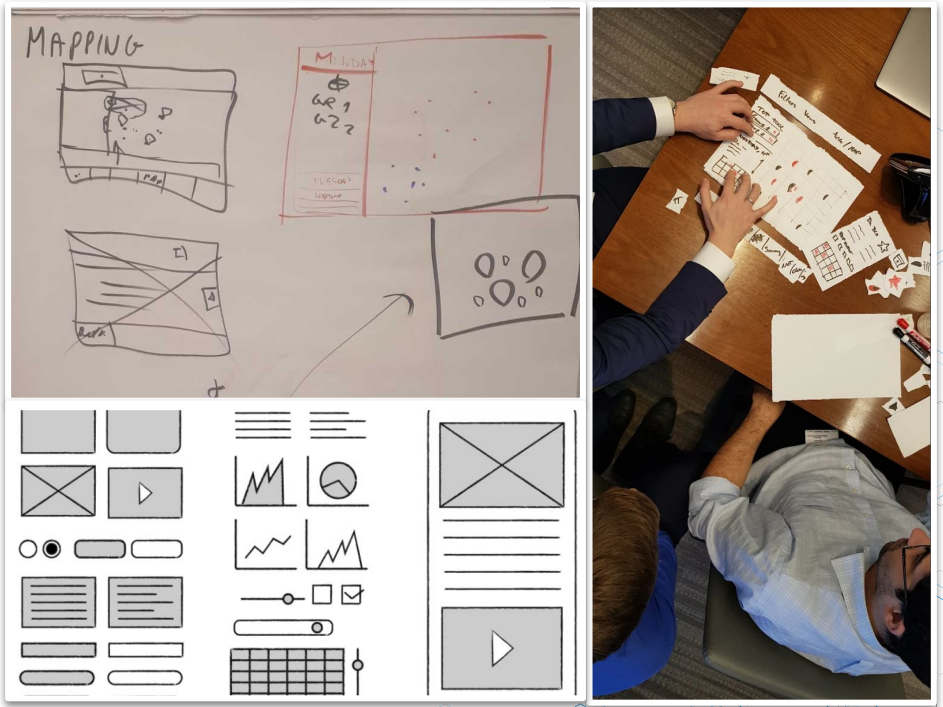
Appsilon

UI/UX process



Prototyping

- Figma
- Powerpoint
- Pen and paper
- Low-fi prototypes



Appsilon

Rules to live by

- Keep it **simple,** stupid (KISS)
- Create **consistency** and use common UI elements - Don't Repeat Yourself (DRY)
- Be purposeful in your **layout**
- Use **color, texture, and typography** to create **hierarchy** and clarity
- Provide **feedback** to the user about what is happening
- Think about the **defaults**

Appsilon

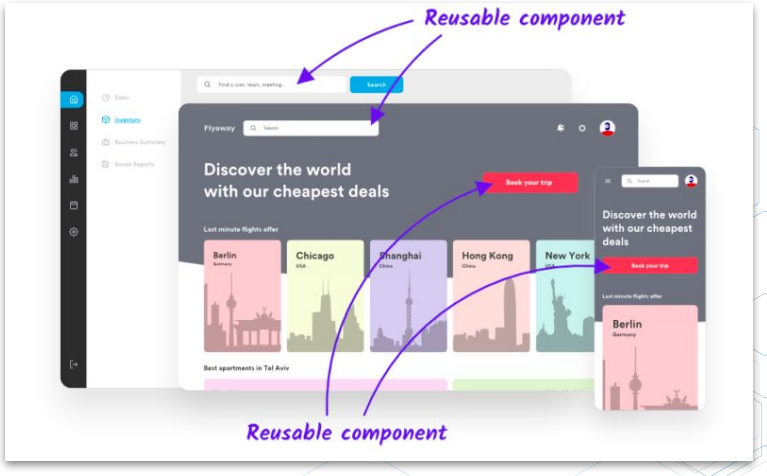
Rules to live by

- Keep the interface **simple** (KISS)
  - Avoid unnecessary elements
  - Be clear in your language

The best interfaces are almost invisible to the user.

Rules to live by

- Be consistent
  - Use common UI elements
  - Reuse components when possible
  - Create patterns in your structure and language



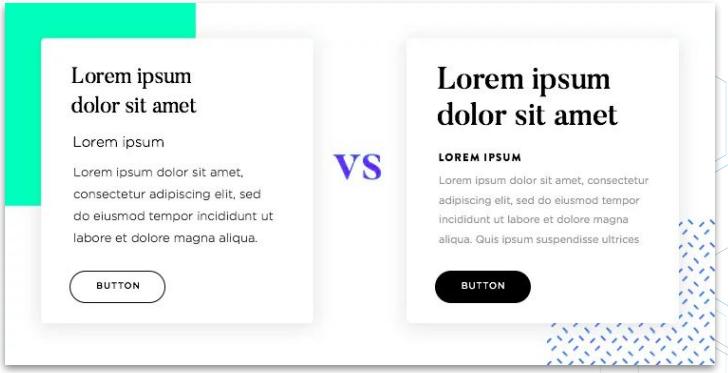
Rules to live by

- Be purposeful in your layout
  - Be spatially mindful
  - Create structure based on importance
  - Draw attention to important information
  - Aid scanning and readability

Appsilon

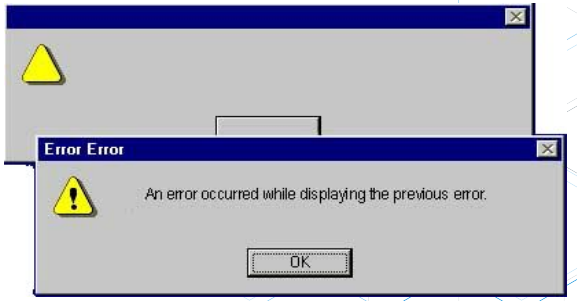
Rules to live by

- Use color, texture and typography strategically
  - Leverage color, light, contrast, and texture
  - Use them to direct attention toward / away from items
  - **Different fonts send different messages**
  - Size, fonts and text arrangement can improve scannability, legibility and readability



Rules to live by

- Provide feedback to the user about what is happening
    - **Always** inform your users of location, actions, changes in state, or errors
    - Use elements to communicate status
    - Use next steps to reduce frustration for your user



Appsilon

Rules to live by

- Think about the defaults
  - Anticipate the goals people bring to your app
  - Create defaults that reduce the burden on the user
  - When possible have some fields pre-chosen or filled out
  - Don’t use defaults for input fields that require user attention

Appsilon

Leverage the community

https://cran.r-project.org/web/packages/available_packages_by_name.html

https://github.com/grabear/awesome-rshiny
https://github.com/nanxstats/awesome-shiny-extensions

Hands-on

Appsilon

Cheeses dataset

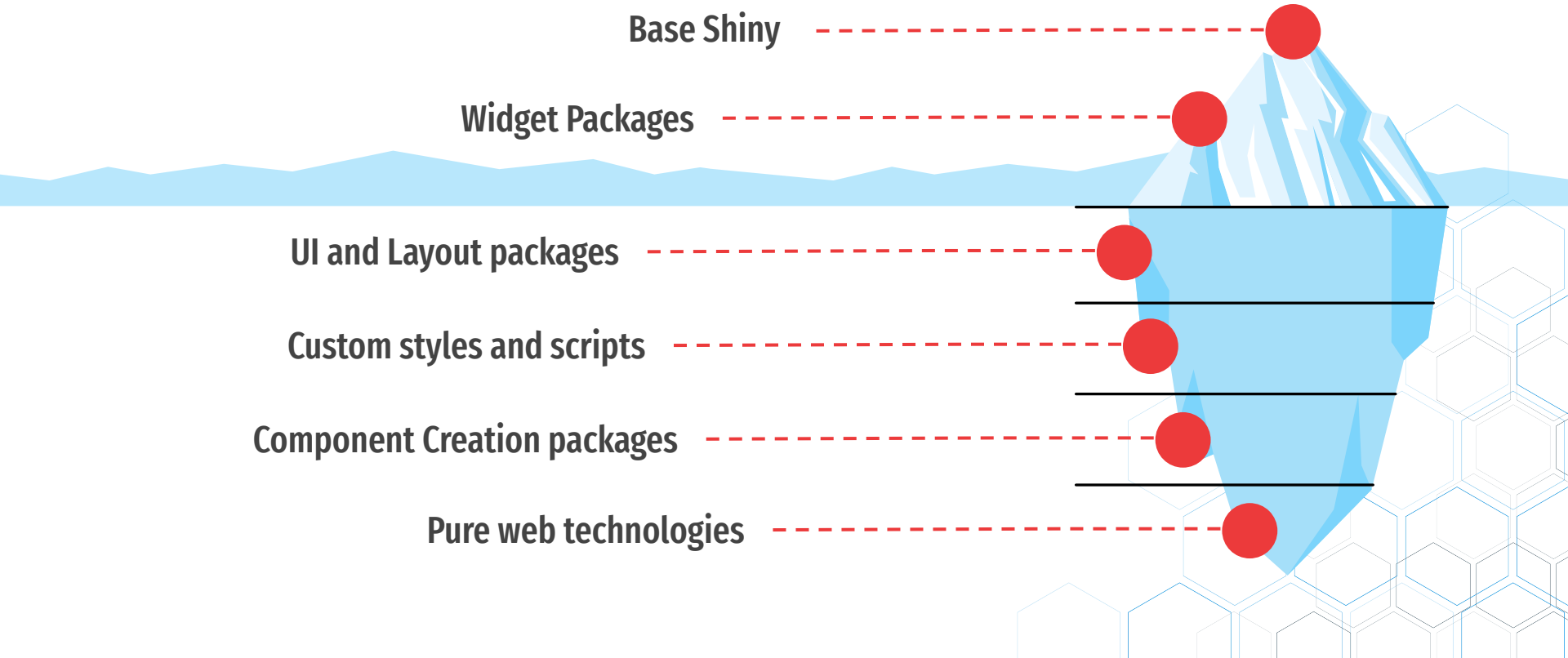
1. Grab paper and pen
2. Sketch your ideal app layout
    a. How will users navigate and interact with the app?
    b. What graphs, tables, or other UI elements best represent the data?
    c. What functionalities should be available (e.g., filters, downloads)?
3. Write your name on the back of the paper
4. Hand it to me :)

What can you do as a Shiny developer?

Appsilon

Appsilon

The depth of custom UI/UX



Appsilon

What can you do as a Shiny developer?

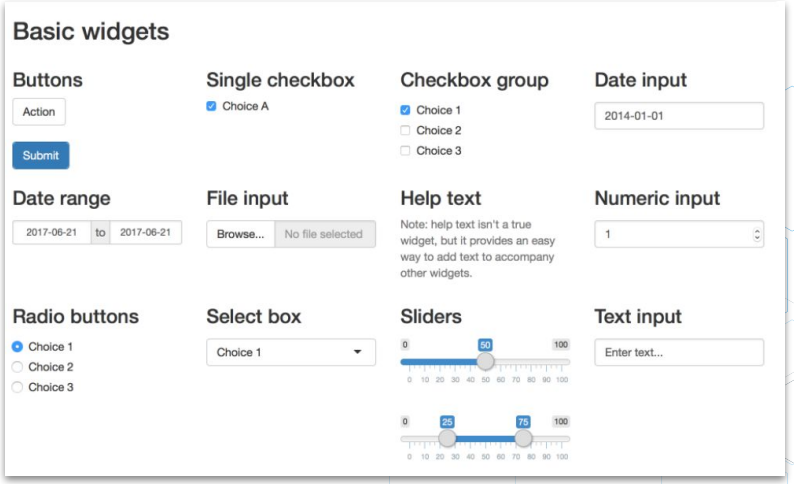
- How it looks
- How its structured
- How it behaves

How it looks

Appsilon

How it looks - Base shiny

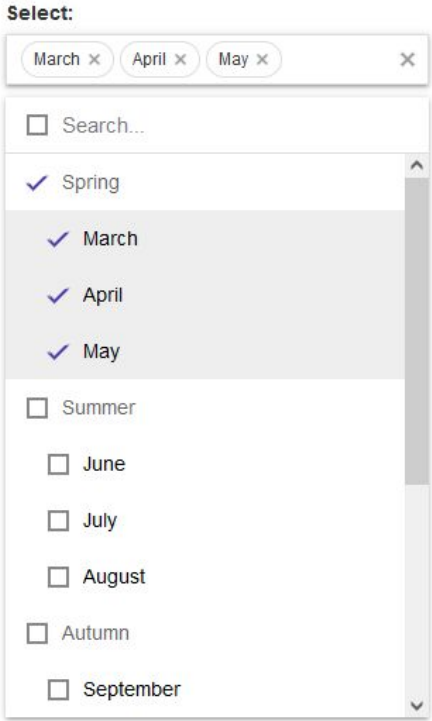
- **Get familiar** with base widgets
- Explore the function arguments to **understand all the options** they offer
- Remember that a **bit of CSS** can go a long way, even for basic widgets

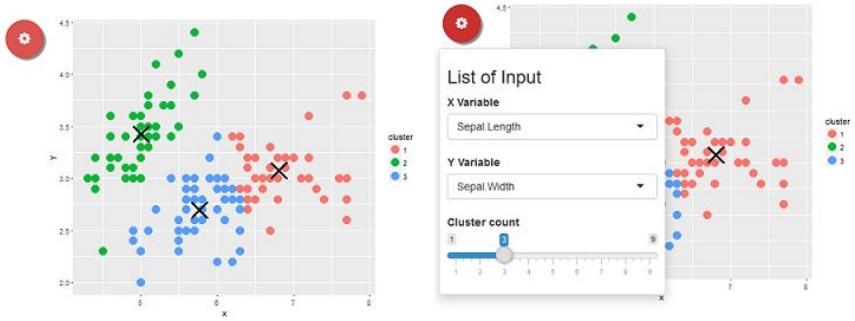


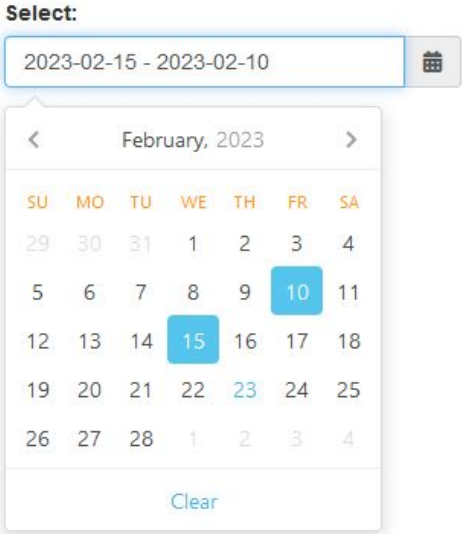
https://shiny.rstudio.com/gallery/widget-gallery.html

Appsilon

How it looks - shinyWidgets





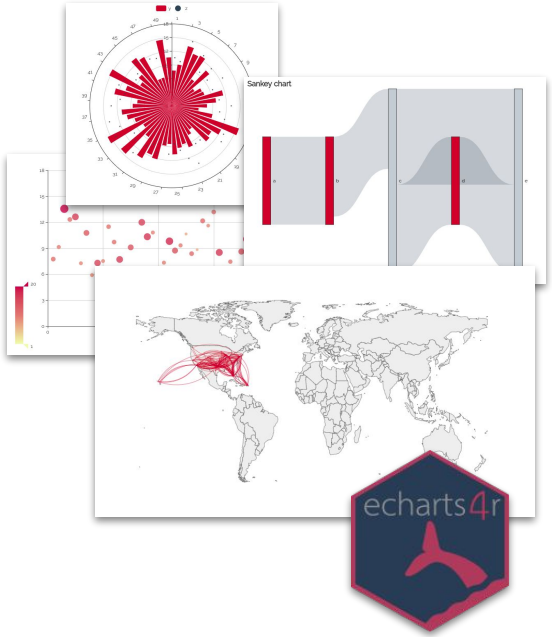


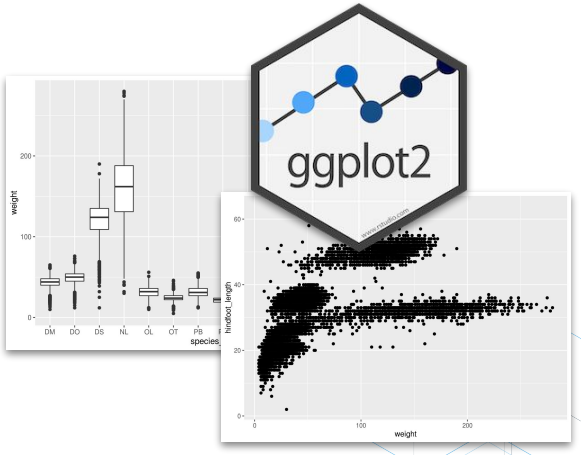
https://shinyapps.dreamrs.fr/shinyWidgets/

Appsilon

How it looks - Additional widgets

- Charts
  - ggplot2
  - Plotly
  - eCharts4r
- Maps
  - leaflet
- Tables
  - DT
  - reactable
  - rhandsontable

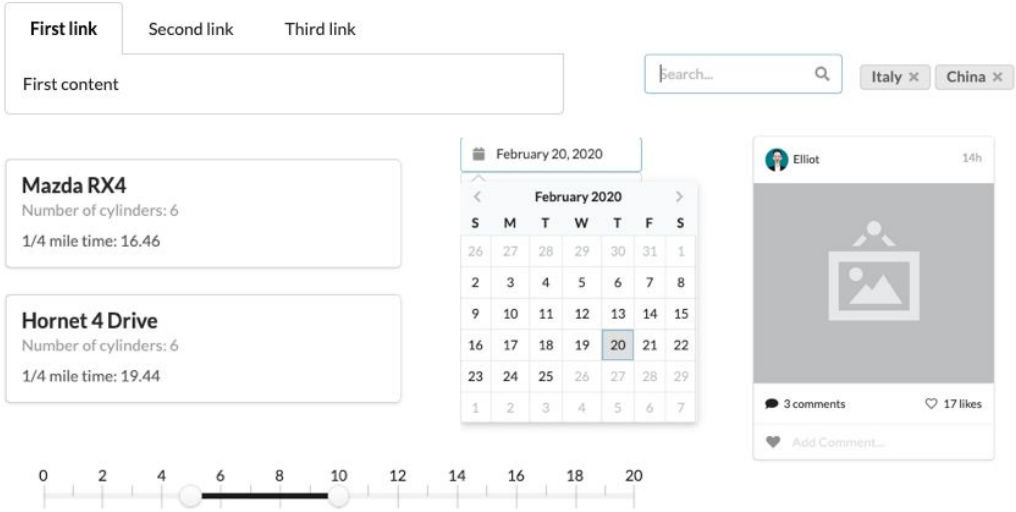




Appsilon

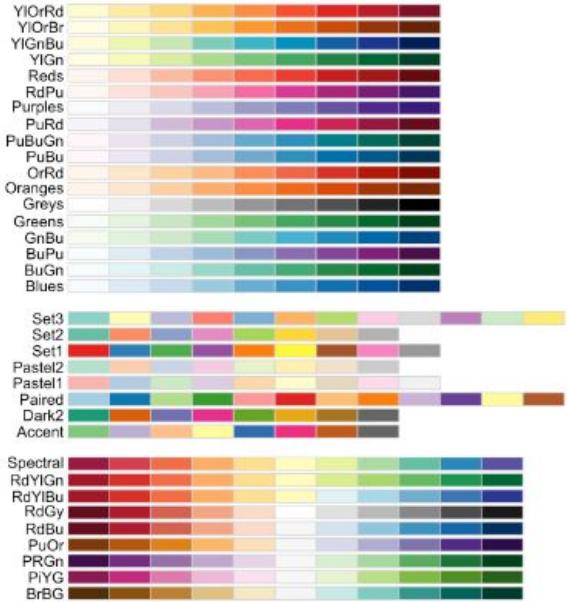
Layouts

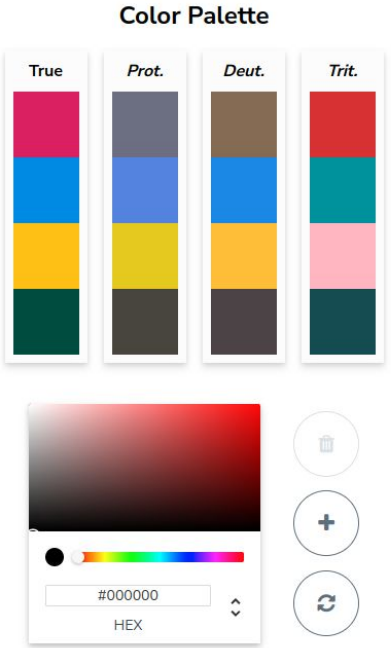
- shiny.semantic
- shiny.fluent
- shinybulma
- shinyMobile
- shinymaterial



Appsilon

How it looks - Colors

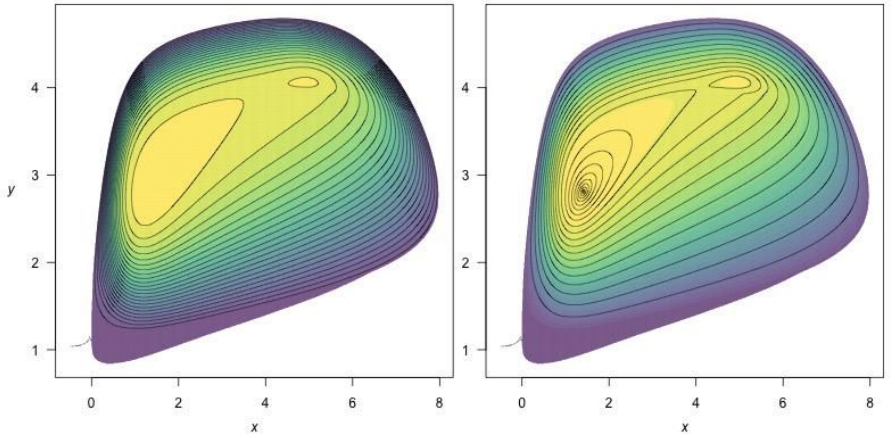




https://r-graph-gallery.com/38-rcolorbrewers-palettes.html
http://hclwizard.org:3000/cvdemulator/

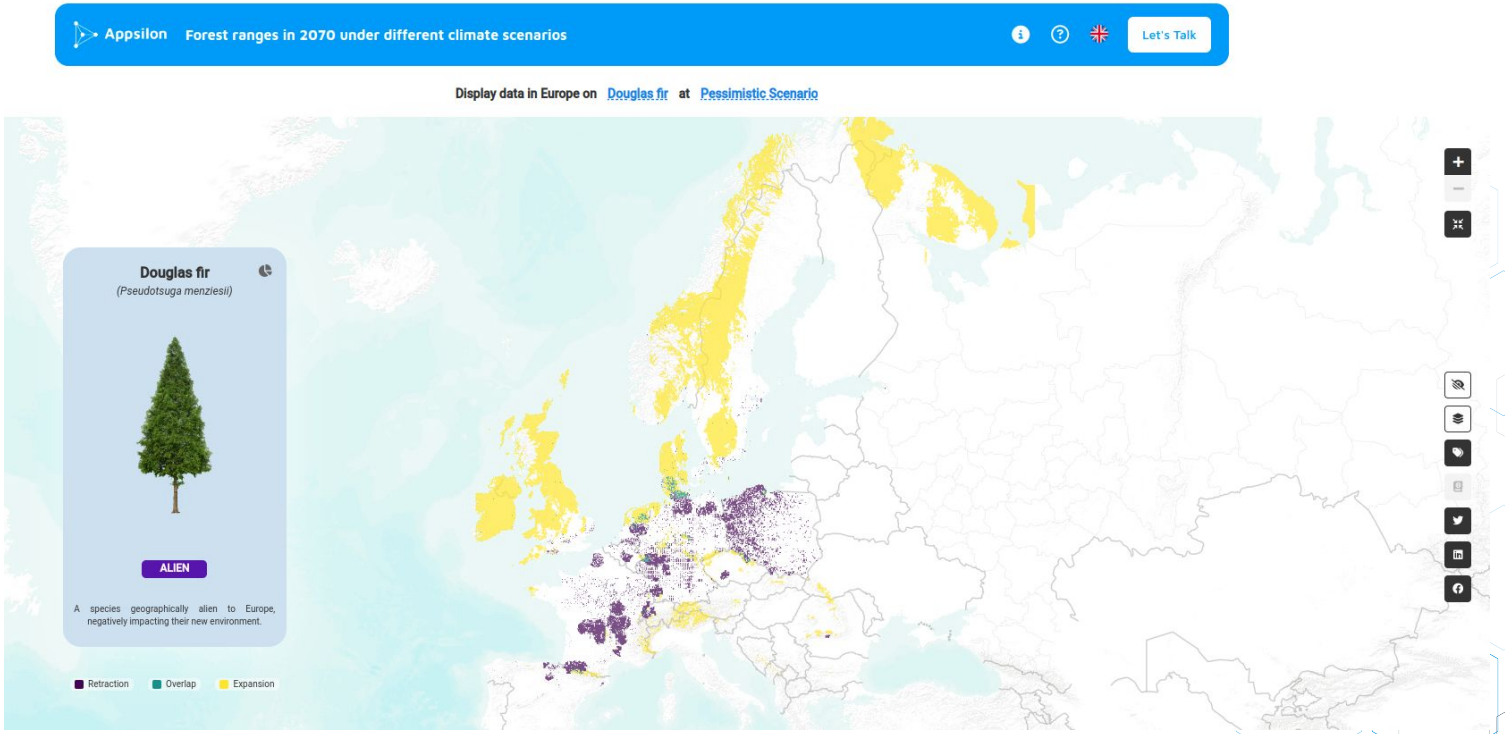
Appsilon

How it looks - Colors



https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html

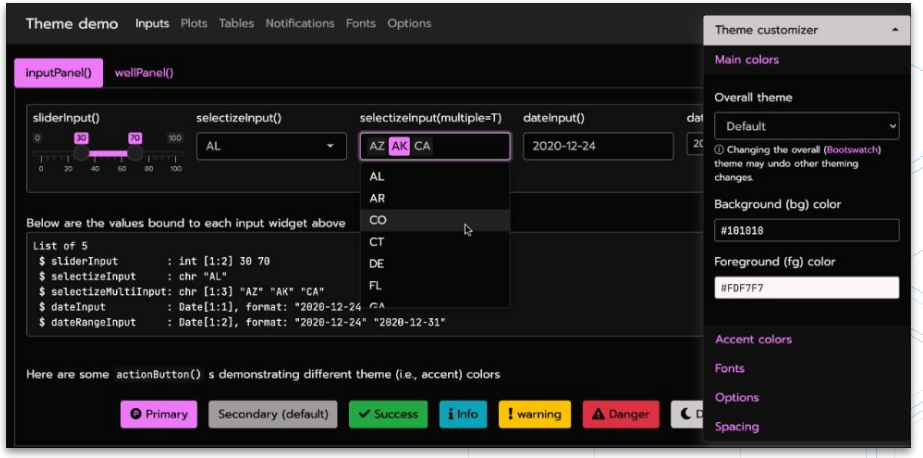
How it looks - Colors



https://connect.appsilon.com/future-forests/

How it looks - Theme packages

- **bslib** - Tools for theming Shiny and R Markdown
- **fresh** - Create fresh themes for use in shiny & shinydashboard applications and flexdashboard documents.
- **Rnightly** - An R wrapper of the JavaScript library Nightly.



examples/04_ui_ux/01_how_it_looks/02_bs_theme.R

Appsilon

How it looks - bslib

- Custom theming of Shiny apps and R Markdown documents.
- Use of modern versions of Bootstrap and Bootswatch

```
ui <- fluidPage(
  theme = bs_theme(
    bootswatch = "darkly",
    base_font = font_google("Inter"),
    navbar_bg = "#25443B"
  ),
  selectInput(
    inputId = "countries",
    label = "Countries",
    choices = country_choices
  ),
  tableOutput(outputId = "cheeses_table")
)
```

Appsilon

How it looks - bslib

	Value
	#fff
	#f8f9fa
	#e9ecef
gray-300	#dee2e6
gray-400	#ced4da
gray-500	#adb5bd
gray-600	#6c757d
gray-700	#495057
gray-800	#343a40
gray-900	#212529
black	#000
blue	#0d6efd
indigo	#6610f2
purple	#6f42c1

Showing 1 to 15 of 932 entries

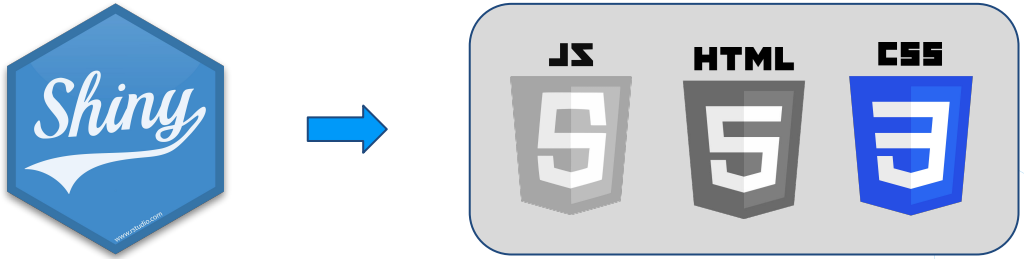
➔ Compatible (through thematic)

- (gg)-plotly
- ggiraph (not in CRAN)
- ggplot, lattice, base plot
- DT
- kableExtra
- shinyWidgets (partly)
- flexdashboard, pkgdown, and bookdown

Appsilon

What is CSS (Cascading Style Sheets)?

- Every web page uses CSS in some way
- Shiny is no exception, by default it uses the bootstrap library for most of its styling



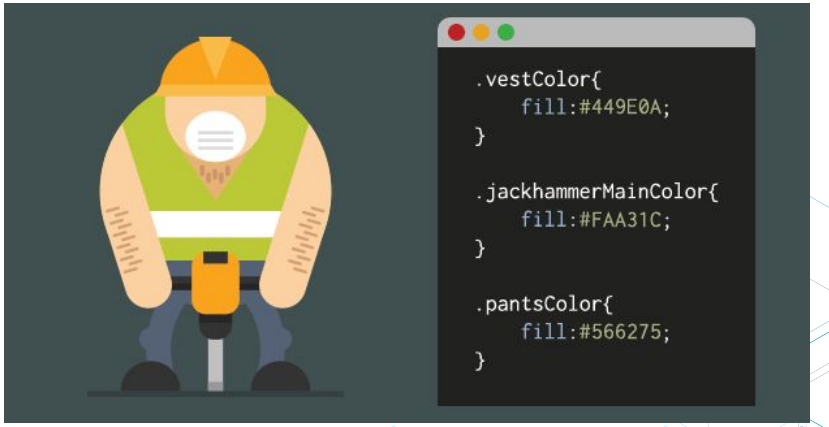
Appsilon

How it looks - Custom styles

- Describes how HTML elements are to be displayed

- It can control the styling of multiple pages and components all at once (CSS saves a lot of work!)

- Instructions are called **statements**. Statements do two things:
  - Identify the elements in an HTML document that it affects
  - Gives the browser rules on how to draw these elements



Appsilon

How it looks - Custom styles

- CSS statements have 2 parts:
  - **Selector**
  - **Declaration**
    - Property
    - Value

```
1.  .btn-default {
2.      color: #ffffff;
3.      background-color : #ff0000;
4.  }
```

Appsilon

How it looks - Custom styles

- There are 3 main ways you can add CSS styling to your code:
  - Add styling directly to tags
  - Add CSS to your header
  - Add style sheets with the www directory

examples/04_ui_ux/01_how_it_looks/04_css_basic.R

Appsilon

Adding CSS to Shiny

1. Add styling directly to HTML tags

```
ui <- fluidPage(
 actionButton(
   inputId = "my_btn",
   label = "Click me",
   style = "color: #ffffff; background-color: #ff0000;"
 )
)
```

✘ Easy to lose track of in large projects
✘ Inability to reuse same rules in different elements
✘ Hard to keep consistency throughout the project
✘ Hard to implement large changes

Appsilon

Adding CSS to Shiny

2. Add CSS to your HTML header

```
ui <- fluidPage(
 tags$head(
   tags$style(
     ".btn-default {
       color: #ffffff;
       background-color: #ff0000;
     }"
   )
 ),
 actionButton(
   inputId = "my_btn", label = "Click me"
 )
)
```

✔ Allows code to be reused by using selectors

✘ Not very organized

Appsilon

Adding CSS to Shiny

3. Add style sheets with the www directory

```
ui <- fluidPage(
 tags$head(
   tags$link(rel = "stylesheet", type = "text/css", href = "styles.css")
 ),
 actionButton(
   inputId = "my_btn",
   label = "Click me"
 )
)
```

✔ Code can be reused
✔ Allows caching
✔ Allows multiple files to be included and allow some structure
✘ CSS can get very complex in large projects

Appsilon

Adding CSS to Shiny

3. Add style sheets with the www directory

```
ui <- fluidPage(
 tags$head(
   includeCSS(path = "www/styles.css")
 ),
 actionButton(
   inputId = "my_btn",
   label = "Click me"
 )
)
```

✔ Code can be reused
✔ Allows caching
✔ Allows multiple files to be included and allow some structure
✘ CSS can get very complex in large projects

examples/04_ui_ux/01_how_it_looks/06_css_file.R

Appsilon

Adding CSS to Shiny

3. Add style sheets with the www directory

```
ui <- fluidPage(
 tags$head(
   includeCSS(path = "www/styles.css")
 ),
 actionButton(
   inputId = "my_btn",
   label = "Click me"
 )
)
```

www/styles.css

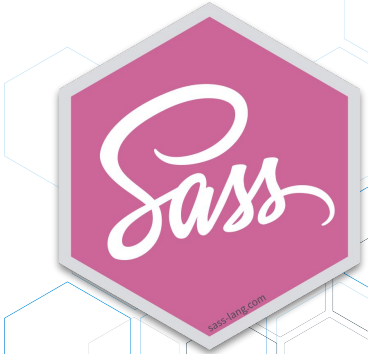
```
.btn-default {
  color: #ffffff;
  background-color: #ff0000;
}
```

- ✔ Code can be reused
- ✔ Allows caching
- ✔ Allows multiple files to be included and allow some structure
- ✘ CSS can get very complex in large projects

Appsilon

How it looks - Custom styles

- SASS (Systematically Awesome Style Sheets) is a pre-processor of CSS (SASS code always compiles into CSS).
- r/sass implements a CSS preprocessor, letting R developers use SASS to generate dynamic style sheets.
- sass() can take a **SASS string or file** and returns a **string of compiled CSS**



Hands-on

Appsilon

Cheeses dataset

1. Change the background color of the entire app.
2. Customize the background color of all inputs:
    - When idle
    - When hovered (mouse over)
    - When active (clicked or focused)
3. Modify the row colors in a DT table:
    - Set a custom color for odd rows
    - Set a different color for even rows
    - Ensure the selected row stands out

How its structured

Appsilon

Appsilon

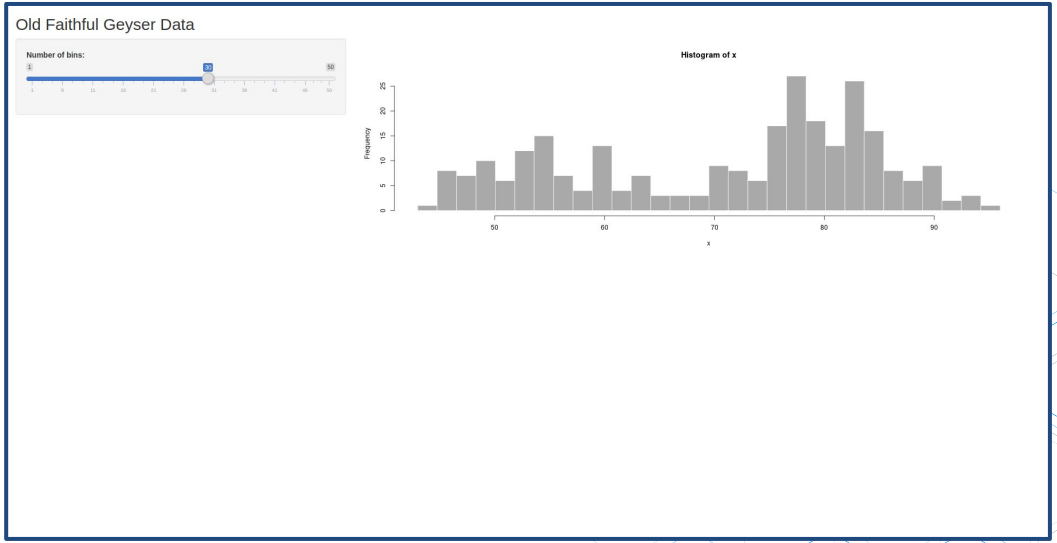
How it is structured

- Base structure of how your application is organized
- Areas where you can add your components (menus, inputs, images, plots)

Appsilon

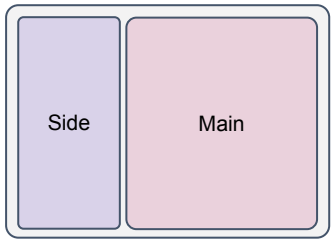
How it is structured

- Base structure of how your application is organized
- Areas where you can add your components (menus, inputs, images, plots)

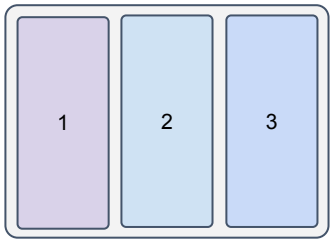


How it is structured - Base shiny

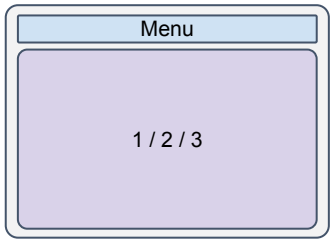
fluidPage()
sidebarLayout()



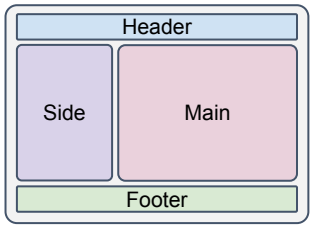
fluidPage()
splitLayout()



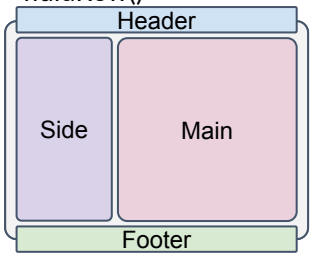
navbarPage()
tabPanel()



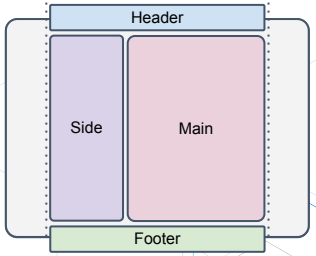
fullPage()
fluidRow()



fluidPage()
fluidRow()



fixedPage()
fixedRow()

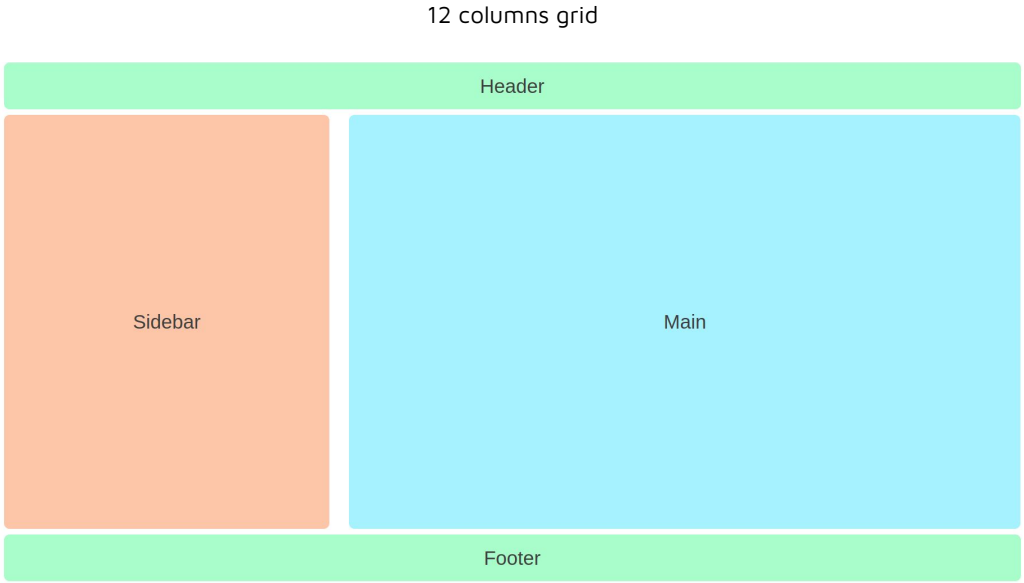


https://shiny.rstudio.com/articles/layout-guide.html

Appsilon

How it is structured - Base shiny (Bootstrap)

```
ui <- fluidPage(
 fluidRow(
   column(
     width = 12,
     element("#a8feca", "10vh", "Header")
   )
 ),
 fluidRow(
   column(
     width = 4,
     element("#fdc6a7", "80vh", "Sidebar")
   ),
   column(
     width = 8,
     element("#a8f2fe", "80vh", "Main")
   )
 ),
 fluidRow(
   column(
     width = 12,
     element("#a8feca", "10vh", "Footer")
   )
 )
)
```



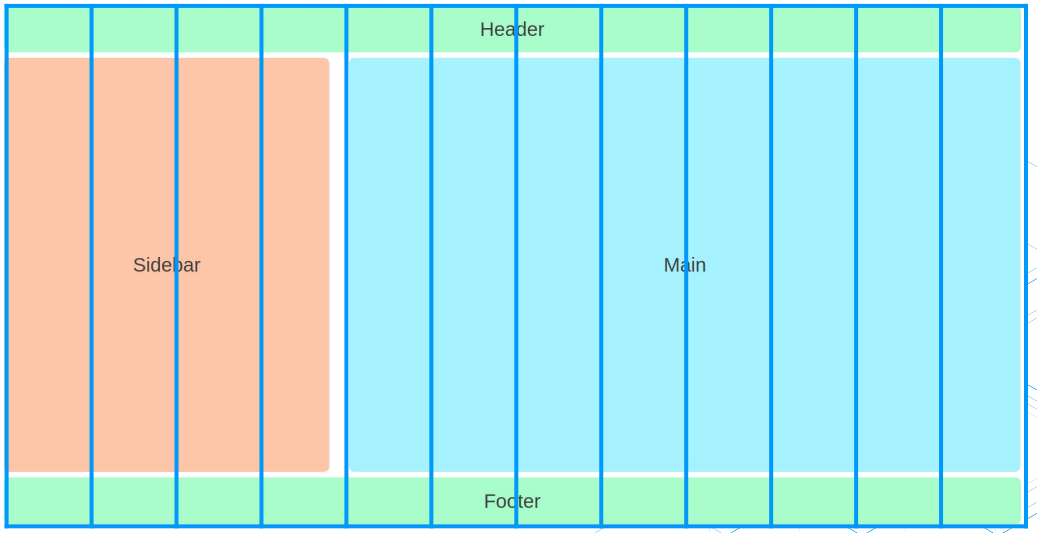
examples/04_ui_ux/01_how_it_is_structured/01_page_layout.R

Appsilon

How it is structured - Base shiny (Bootstrap)

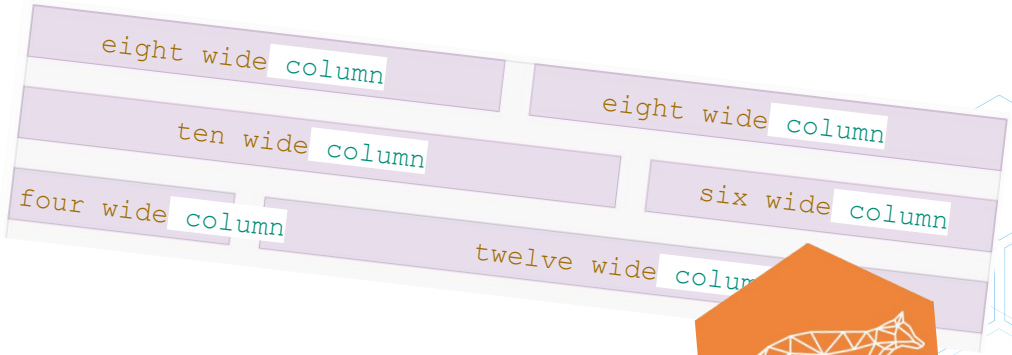
```r
ui <- fluidPage(
 fluidRow(
   column(
     width = 12,
     element("#a8feca", "10vh", "Header")
   )
 ),
 fluidRow(
   column(
     width = 4,
     element("#fdc6a7", "80vh", "Sidebar")
   ),
   column(
     width = 8,
     element("#a8f2fe", "80vh", "Main")
   )
 ),
 fluidRow(
   column(
     width = 12,
     element("#a8feca", "10vh", "Footer")
   )
 )
)
```

12 columns grid



https://en.wikipedia.org/wiki/Highly_composite_number

How it is structured - Layout packages



Appsilon

How it is structured - Layout packages

- **shinydashboard** - Shiny dashboarding framework based on AdminLTE 2.
- **shinydashboardPlus** - Additional AdminLTE 2 components for shinydashboard.
- **gentelellaShiny** - Bootstrap 3 Gentelella theme for Shiny dashboards.
- **semantic.dashboard** - Semantic UI for Shiny dashboards.
- **bs4Dash** - Bootstrap 4 Shiny dashboards using AdminLTE 3.
- **argonDash** - Bootstrap 4 Argon template for Shiny dashboards.
- **tablerDash** - Tabler dashboard template for Shiny with Bootstrap 4.

Hands-on

Appsilon

Cheeses dataset

Implement the layout you sketched in your first hands-on exercise. If any widgets are missing or not yet available, add placeholders instead

How it behaves

Appsilon

Appsilon

How it is behaves - Additional packages

- **shinyjs** - Perform common JavaScript operations in Shiny apps.
- **shinyCanvas** - Create and customize an interactive canvas using the D3 JavaScript library and the htmlwidgets package.
- **shinyscroll** - Scroll to an element in Shiny.
- **pagemapR** - Create a minimap for Shiny apps and R Markdown documents.
- **keys** - Assign and listen to keyboard shortcuts in Shiny using the Mousetrap Javascript library.

Appsilon

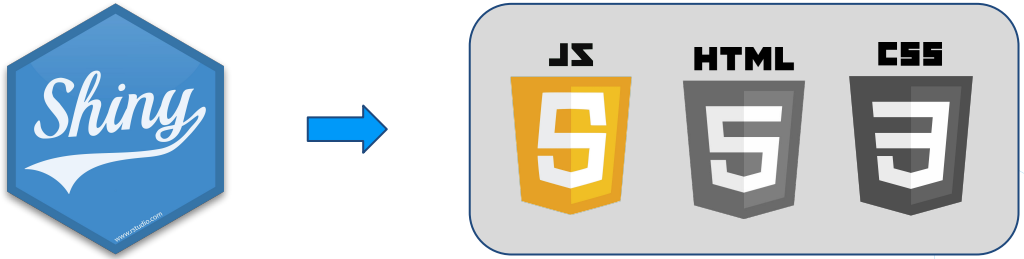
What is JavaScript?

- JavaScript is the Programming Language for the Web.
- It can update and change both HTML and CSS.
- JavaScript can calculate, manipulate and validate data

Appsilon

What is JavaScript?

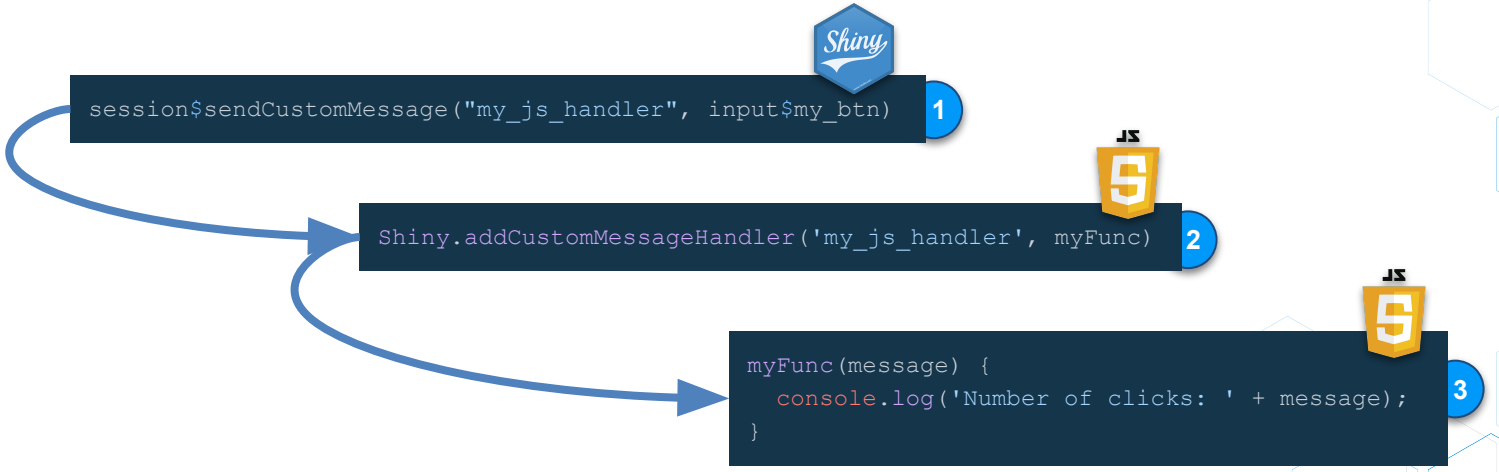
- Shiny contains a full JavaScript framework that works together with the R session to achieve any desired behavior
- This is what allows shiny to observe events and also trigger any changes in the browser
- In the browser, **all** Shiny behavior is ruled by javascript



Appsilon

How its behaves - Custom messages

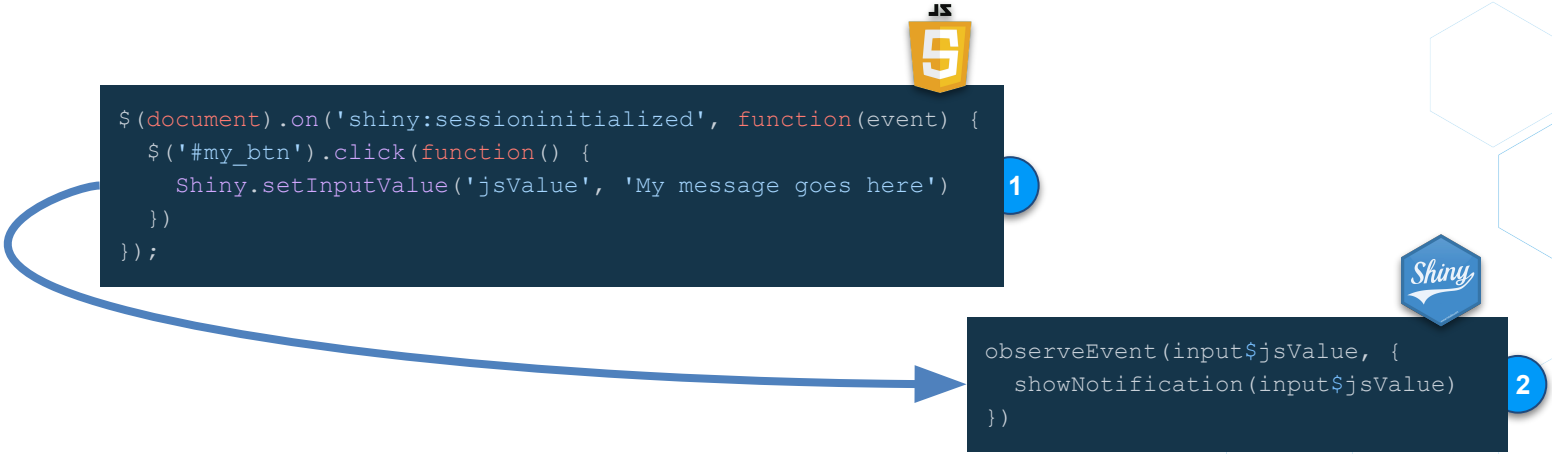
- Trigger JavaScript functions from R, e.g. hide an element when you observe a specific event



Appsilon

How its behaves - Custom messages

- Let Shiny listen to events that happen in the browser



https://shiny.posit.co/r/articles/build/js-events/

Appsilon

How its behaves - Custom JS

- Select a page element
  - **$("{css selector}")**
- Apply an event function
- Event functions:
  - **click()**
  - **dbclick()**
  - **mouseenter()**
  - **mouseleave()**

```
> '5' - 3
< 2          // weak typing + type coercion = headache
> '5' + 3
< '53'       // we love constancy, don't we?
> '5' - '4'
< 1          // string - string = number. what?
> '5' + + '5'
< '55'       // okay, let it be
> 'foo' + + 'foo'
< 'fooNaN'   // brilliant!
> '5' + - '2'
< '5-2'      // great!
> '5' + - + - - + - - + + - + - + - + - + - - - '-2'
< '52'       // let's say, this is how it should be
> var x = 3
> '5' - x + x
< 5          // it makes sense
> '5' + x - x
< 50         // fuck math!
```

https://www.w3schools.com/jquery/

examples/04_ui_ux/01_how_it_behaves/03_what_triggered.R

Appsilon

How its behaves

```
1.  library(shiny)
2.
3.  ui <- fluidPage(
4.   tags$head(
5.     tags$script(
6.       "$(document).on('shiny:inputchanged', function(event) {
7.           if (event.name != 'changed') {
8.             Shiny.setInputValue('changed', event.name);
9.           }
10.        });"
11.    )
12.   ),
13.   actionButton(inputId = "test1", label = "button 1"),
14.   actionButton(inputId = "test2", label = "button 2")
15. )
16.
17. server <- function(input, output, session) {
18.   observeEvent(input$changed, {
19.     showNotification(input$changed)
20.   })
21. }
22.
23. shinyApp(ui, server)
```

Appsilon

How its behaves - Custom JS

- Use these when you have custom behaviour already programmed in javascript, or want to offload some computation to the browser directly
- Using custom messages also allows you to extend the update functions behavior for actions that suit your project
- Keep in mind that this requires some knowledge of javascript to get started

```
1.  $(document).on('shiny:inputchanged', function(event) {
2.    if (event.name != 'changed') {
3.      Shiny.setInputValue('changed', event.name);
4.    }
5.  })
```

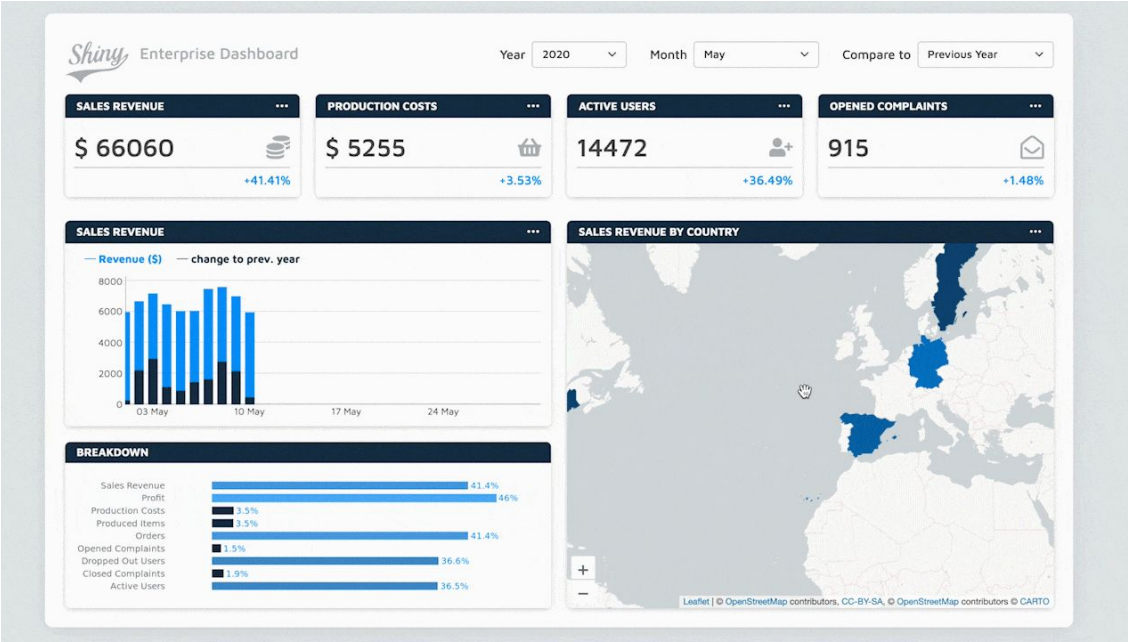
```
tags$script(src =
"main.js")
```

Examples

Appsilon

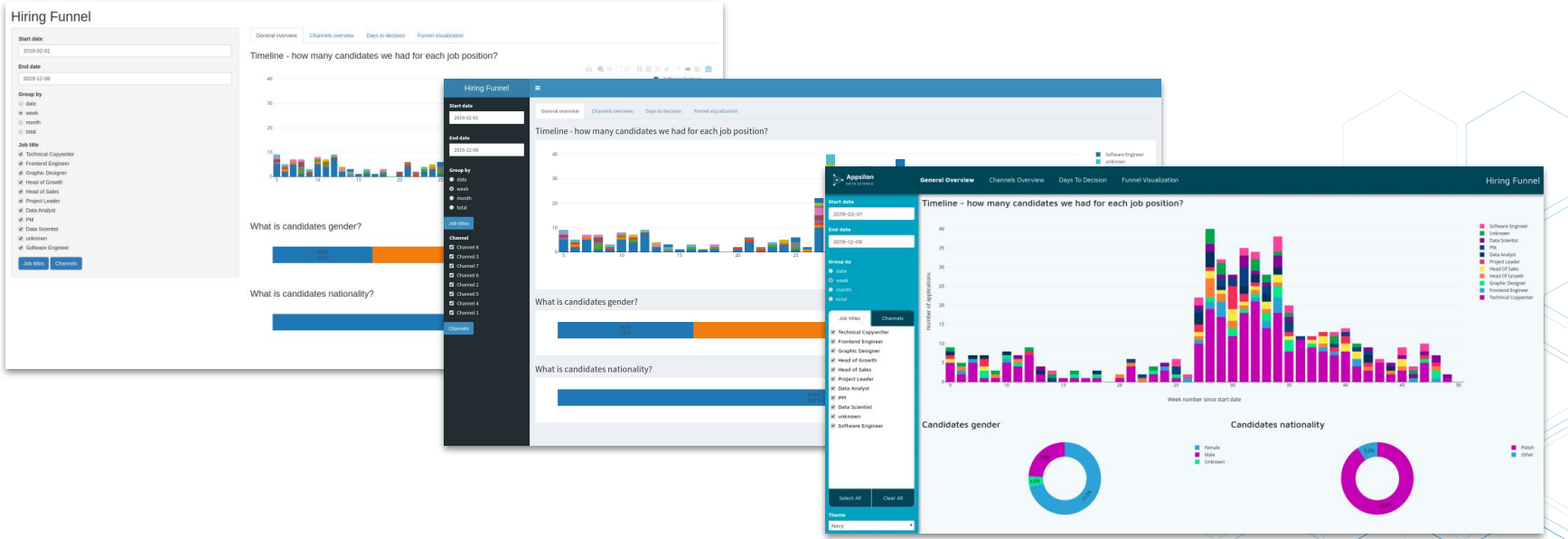
Appsilon

Examples



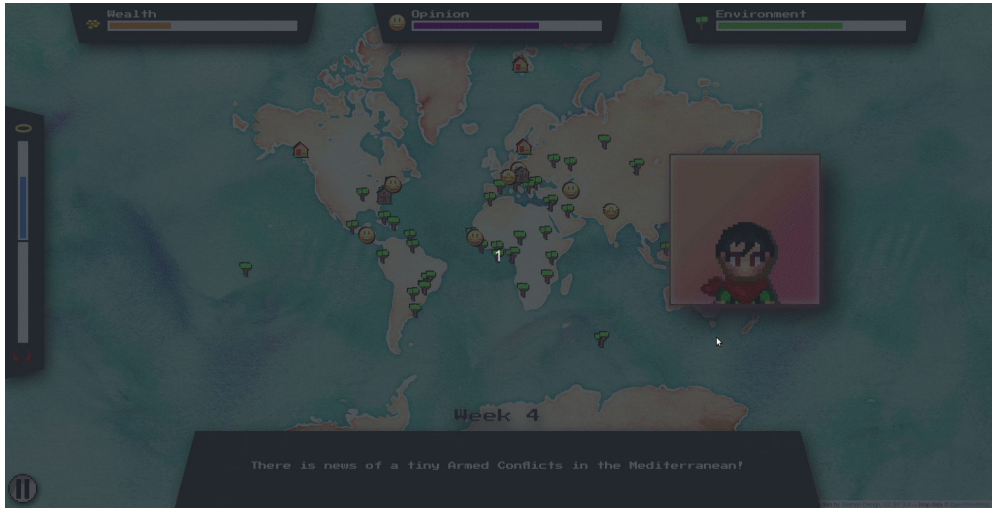
- Appsilon Shiny Dashboard (shiny.semantic)

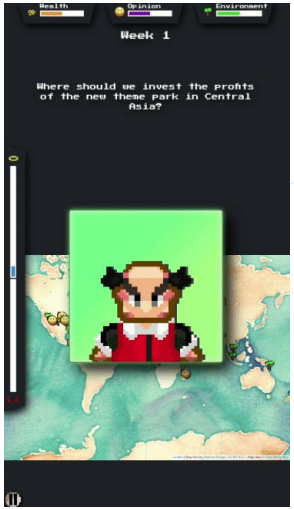
Examples



- Hiring funnel UI update (Shiny)
  https://appsilon.com/journey-from-basic-prototype-to-production-ready-shiny-dashboard/

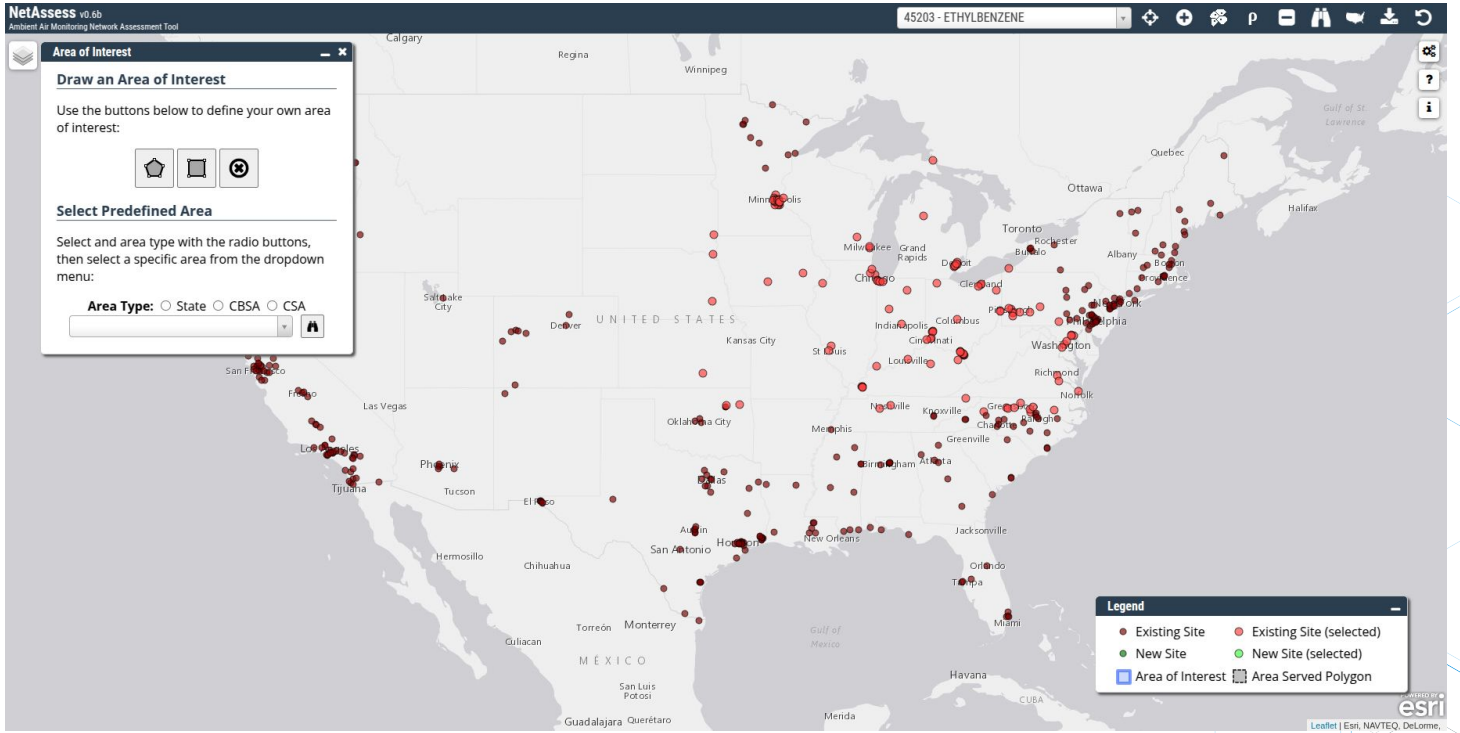
Examples





- Shiny Decisions
  - Reuses existing CSS framework (nes.css)
  - Custom built layouts and behavior
    https://appsilon.com/is-it-possible-to-build-a-video-game-in-r-shiny/

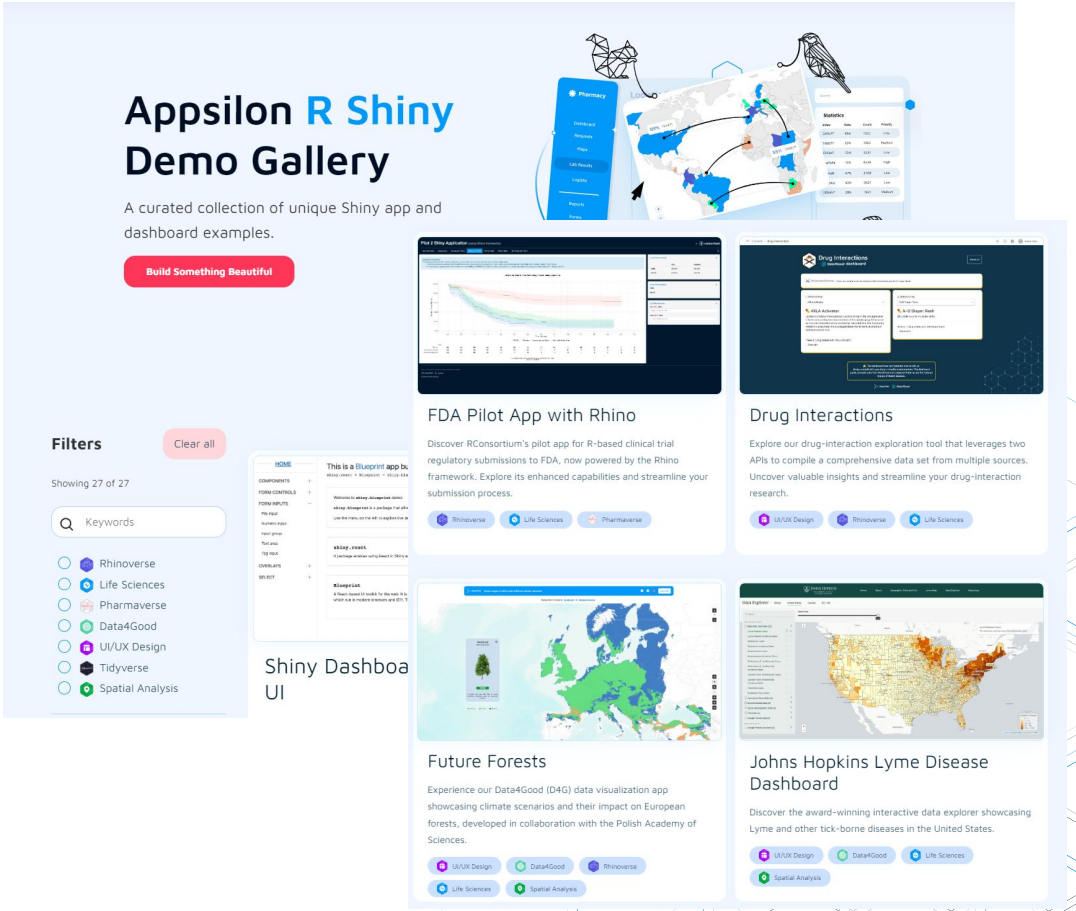
Examples



https://ladco.shinyapps.io/NetAssessApp/

Examples

https://demo.appsilon.com/



Hands-on

Appsilon

Cheeses dataset

1. Add a CSS file to your project and create reusable CSS classes for consistent styling across the app
2. Replace default Shiny inputs with equivalent components from {shinyWidgets} for a more polished look
3. Adjust the charts and map to align with your app’s design
4. Display a map with polygons representing available countries
5. Detect user clicks on a country
6. When a user clicks on a country, show a modal dialog listing the available cheeses for that country.

Using Modules and Box

Appsilon
Douglas Mesquita

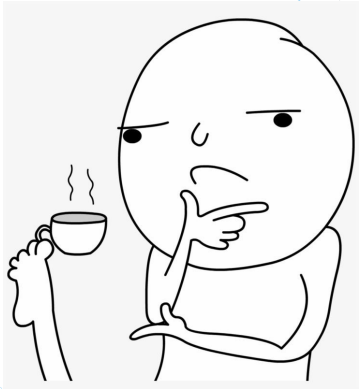
Belo Horizonte
2025-02-03 - 2025-02-06

Appsilon

Modules

What is easier?

- Running the whole app to reproduce an error?
- Running a part of the app that causes problems?



Modules

Modularity is about separating pieces of code from the rest.

Modular code allows us to test pieces of code in isolation, under precise boundary conditions.

It can drastically speed up the debugging efforts.

Appsilon

Namespacing

```
1.  my_ui <- function(id) {
2.    ns <- NS(id)
3.    tagList(
4.      selectInput(
5.        inputId = ns("my_input"),
6.        label = "letters",
7.        choices = letters
8.      ),
9.      uiOutput(outputId = ns("my_selection"))
10.   )
11.  }
```

```
1.  my_server <- function(id) {
2.    moduleServer(
3.      id,
4.      function(input, output, session) {
5.        output$my_selection <- renderUI({
6.          h1(input$my_input)
7.        })
8.      }
9.    )
10.  }
```

Both UI and server are functions of id

Appsilon

Namespacing

```
1.  my_ui <- function(id) {
2.    ns <- NS(id)
3.    tagList(
4.      selectInput(
5.        inputId = ns("my_input"),
6.        label = "letters",
7.        choices = letters
8.      ),
9.      uiOutput(outputId = ns("my selection"))
10.   )
11.  }
```

```
1.  my_server <- function(id) {
2.    moduleServer(
3.      id,
4.      function(input, output, session) {
5.        output$my_selection <- renderUI({
6.          h1(input$my_input)
7.        })
8.      }
9.    )
10.  }
```

NS helps to controls the namespacing: *first_id-second_id-third_id*

Appsilon

Namespacing

```
my_ui <- function(id) {
 ns <- NS(id)
 tagList(
   selectInput(
     inputId = ns("my_input"),
     label = "letters",
     choices = letters
   ),
   uiOutput(outputId = ns("my_selection"))
 )
}
```

```
my_server <- function(id) {
 moduleServer(
   id,
   function(input, output, session) {
     output$my_selection <- renderUI({
       h1(input$my_input)
     })
   }
 )
}
```

Based on the matched id, *moduleServer* creates the reactivity. Notice that we do not use the ns() in the server side

examples/05_modules/01_namespacing.R

Appsilon

Modules

```
1.  ui <- fluidPage(
2.    my_ui("my_id")
3.  )
4.
5.  server <- function(input, output, session) {
6.    my_server("my_id")
7.  }
8.
9.  shinyApp(ui, server)
```

Match the ids and use ui and server normally

examples/05_modules/02_using_modules_multiple_times.R

Appsilon

Using the same module multiple times

```
1.  ui <- fluidPage(
2.    my_ui("my_id1"),
3.    my_ui("my_id2")
4.  )
5.
6.  server <- function(input, output, session) {
7.    my_server("my_id1")
8.    my_server("my_id2")
9.  }
10.
11. shinyApp(ui, server)
```

Take care with ids while reusing modules

Appsilon

Modules

```
1.  my_server <- function(id, excluded_letters = NULL) {
2.   moduleServer(
3.     id,
4.     function(input, output, session) {
5.       if (is.reactive(excluded_letters)) {
6.         observeEvent(excluded_letters(), {
7.           choices <- letters[letters != excluded_letters()]
8.           updateSelectInput(
9.             session = session,
10.            inputId = "my_input",
11.            choices = choices
12.          )
13.        })
14.      }
15.
16.      output$my_selection <- renderUI({
17.        h1(input$my_input)
18.      })
19.
20.      return(reactive({ input$my_input }))
21.    }
22.  )
23. }
```

New parameter to exclude letters

Appsilon

Modules

```
1.  my_server <- function(id, excluded_letters = NULL) {
2.   moduleServer(
3.     id,
4.     function(input, output, session) {
5.       if (is.reactive(excluded_letters)) {
6.         observeEvent(excluded_letters(), {
7.           choices <- letters[letters != excluded_letters()]
8.           updateSelectInput(
9.             session = session,
10.            inputId = "my_input",
11.            choices = choices
12.          )
13.        })
14.      }
15.
16.      output$my_selection <- renderUI({
17.        h1(input$my_input)
18.      })
19.
20.      return(reactive({ input$my_input }))
21.    }
22.  )
23.  }
```

Return the selected letter as a reactive

Appsilon

Modules

```
1.  my_server <- function(id, excluded_letters = NULL) {
2.   moduleServer(
3.     id,
4.     function(input, output, session) {
5.       if (is.reactive(excluded_letters)) {
6.         observeEvent(excluded_letters(), {
7.           choices <- letters[letters != excluded_letters()]
8.           updateSelectInput(
9.             session = session,
10.            inputId = "my_input",
11.            choices = choices
12.          )
13.        })
14.      }
15.
16.      output$my_selection <- renderUI({
17.        h1(input$my_input)
18.      })
19.
20.      return(reactive({ input$my_input }))
21.    }
22.  )
23.  }
```

Remove excluded letter from the list of letters in case excluded_letters is a reactive

Appsilon

Modules

```
1.  ui <- fluidPage(
2.   my_ui("my_id1"),
3.   my_ui("my_id2")
4.  )
5.
6.  server <- function(input, output, session) {
7.   selected_letter <- my_server("my_id1")
8.   my_server("my_id2", selected_letter)
9.  }
10.
11. shinyApp(ui, server)
```

Return the selected letter in the first time we use the module

examples/05_modules/03_returning_reactives.R

Appsilon

Modules

```
1.  ui <- fluidPage(
2.   my_ui("my_id1"),
3.   my_ui("my_id2")
4.  )
5.
6.  server <- function(input, output, session) {
7.   selected_letter <- my_server("my_id1")
8.   my_server("my_id2", selected_letter)
9.  }
10.
11. shinyApp(ui, server)
```

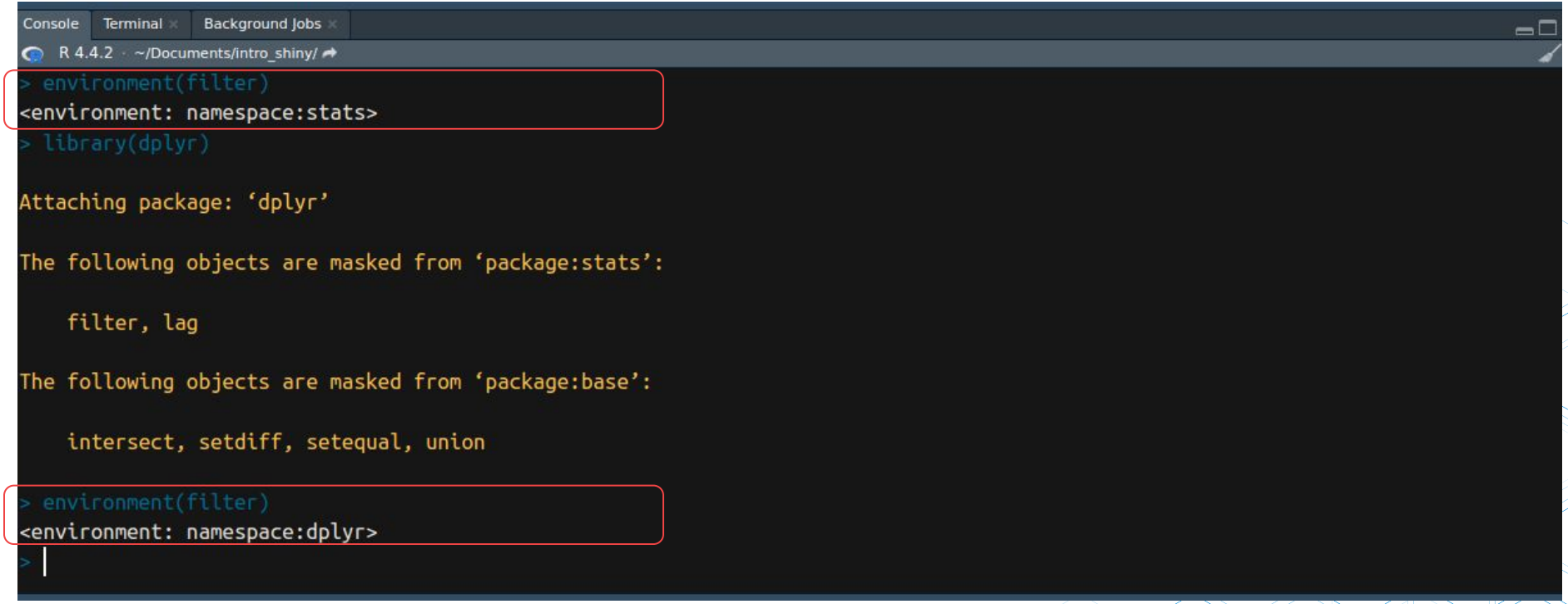
Exclude the selected letter from the second list

Appsilon

Using box

Appsilon

Using box



Using box

```
library(rhino)

# UI
library(shiny.router)
library(shiny.semantic)
library(shinyWidgets)
library(toastui)
library(reactable)
library(DT)
library(shinyTime)
library(timevis)
library(waiter)
library(htmltools)

# Utils
library(R6)
library(shinyjs)
library(lubridate)
library(yaml)
library(dplyr)
library(readr)
library(uuid)
library(here)
library(tidyr)
library(shinyvalidate)
library(snakecase)
library(covr)
library(shinytest2)
library(zip)
library(blastula)
```

```
# Storage
library(DBI)
library(pool)
library(odbc)
library(dbplyr)
library(paws.storage)
library(paws.security.identity)

# Deployment
library(rsconnect)

# Testing
library(testthat)
library(mockery)

# API
library(plumber)
library(httr2)

# Testing
library(testthat)
library(checkmate)

# Validator
library(textreuse)
library(stopwords)
library(stringdist)
```

In real life…

It is almost impossible to load packages without any clash between functions

Also, the number of functions in each package are huge

Appsilon

Using box

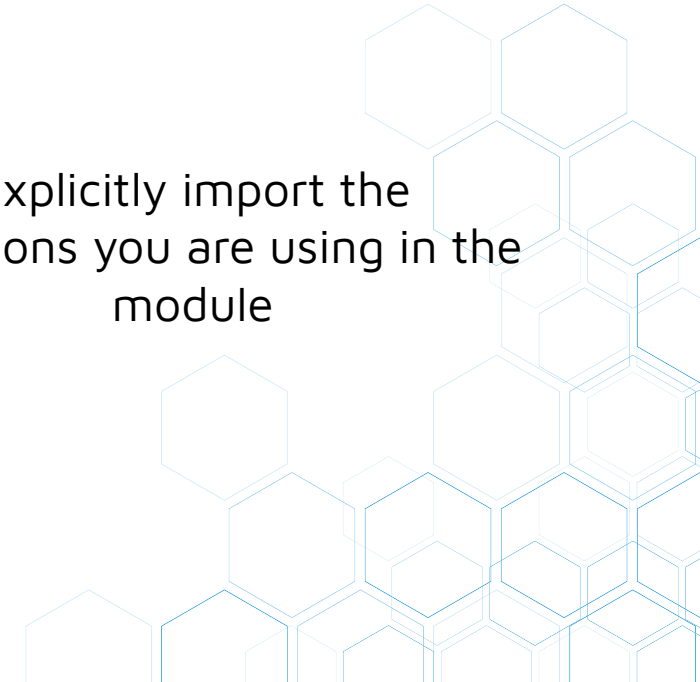
- **Simplifies Module Management**: Enables modular programming by letting you create and reuse self-contained modules
- **Improves Code Organization:** Helps break code into smaller, manageable units for better readability and maintenance
- **Dependency Management**: Automatically resolves and loads required dependencies for modules
- **Avoids Name Conflicts**: Uses explicit module imports to prevent function/variable name clashes.
- **Testing and Debugging**: Facilitates isolated testing of modules, making debugging more efficient.

Appsilon

Using box

```
box::use(
 shiny[
   h1,
   moduleServer,
   NS,
   renderUI,
   selectInput,
   tagList,
   uiOutput,
 ],
)
```

Explicitly import the functions you are using in the module



Appsilon

Using box

```
1.  #' @export
2.  my_ui <- function(id) {
3.   ns <- NS(id)
4.   tagList(
5.     selectInput(
6.       inputId = ns("my_input"),
7.       label = "letters",
8.       choices = letters
9.     ),
10.    uiOutput(outputId = ns("my_selection"))
11.  )
12.  }
13.
14.  #' @export
15.  my_server <- function(id) {
16.   moduleServer(
17.     id,
18.     function(input, output, session) {
19.       output$my_selection <- renderUI({
20.         h1(input$my_input)
21.       })
22.     }
23.  )
24.  }
```

Export the functions you want to reuse

examples/05_modules/03_returning_reactives.R

Appsilon

Using box

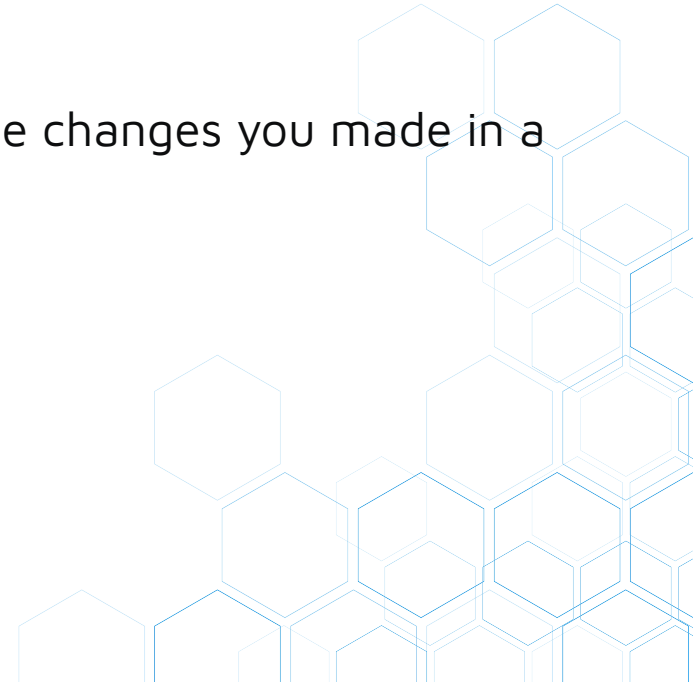
```
box::use(
  shiny[fluidPage, shinyApp],
  ./my_module[my_ui, my_server],
)
```

Import your own modules

Appsilon

Using box - IMPORTANT!

Use box::purge_cache() in order to initially see the changes you made in a module



Hands-on

Appsilon

Cheeses dataset

1. *Refactor the application using box*
2. *Update functions in utils.R to align with {box} conventions*
3. *Modularize the application by splitting it into the following modules:*
    a. *Map and click observer*
    b. *Table and click observer*
    c. *Modal window and its content*
    d. *Embedding images*

Large Applications

Appsilon
Douglas Mesquita

Belo Horizonte
2025-02-03 - 2025-02-06

Appsilon

Package management

Appsilon

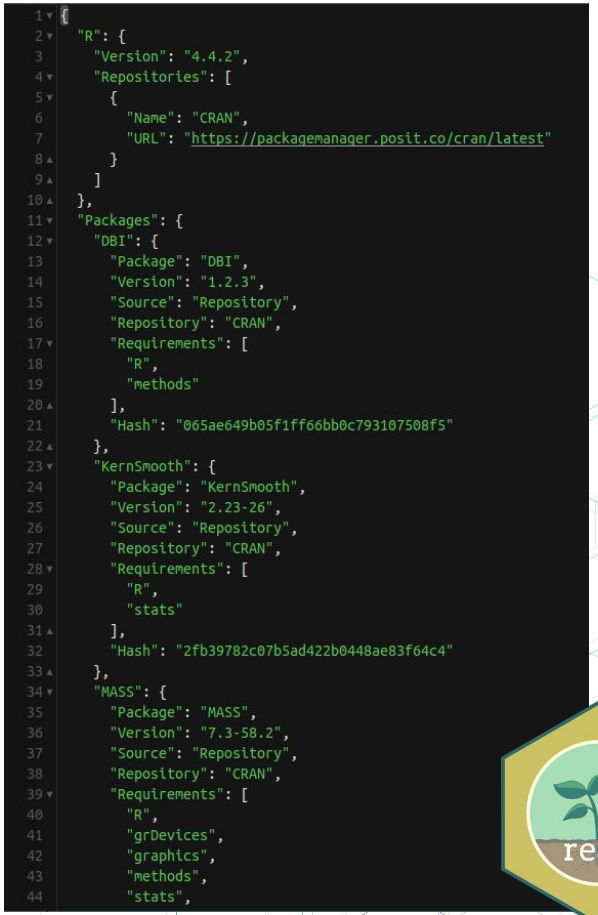
Appsilon

What is renv?

- R package on CRAN
- “Dependency management system for R”
- Goal: make your projects isolated, portable, reproducible

In concrete terms:

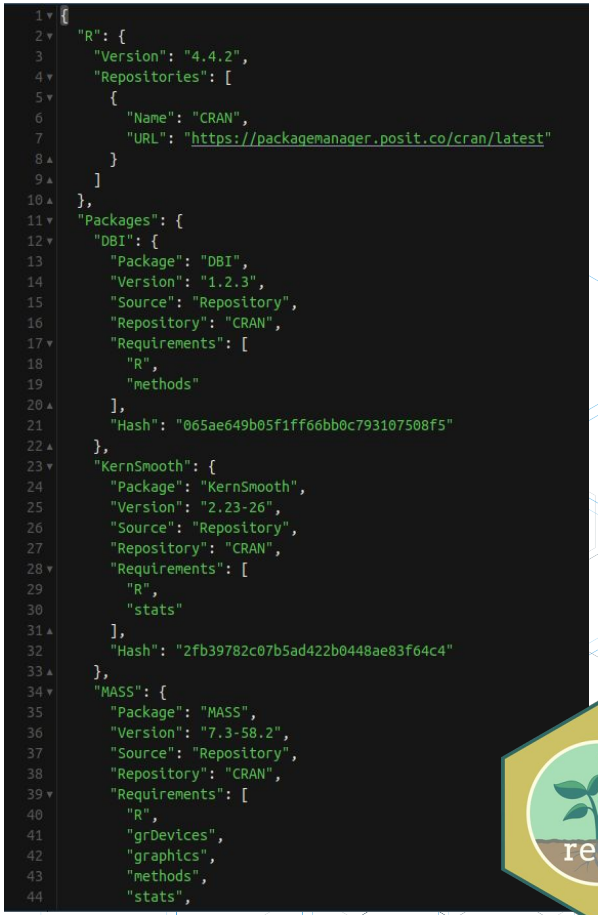
save and restore the exact set of R packages with specific versions



Appsilon

Initialization

- Use renv::init()
- The renv.lock file describes your environment:
  - Packages along with versions & sources
  - Repositories
  - R version



Appsilon

Adding packages

1. library(package)
2. renv::install("package")
3. renv::snapshot()
4. renv::status()

The intersection of used and installed packages is saved to renv.lock, along with all transitive dependencies.

Appsilon

Restoring Environments

By default renv::restore() doesn't remove packages

- easy to forget to update renv.lock

Always use renv::restore(clean = TRUE)

- restores the packages exactly as described in renv.lock

Appsilon

Managing versions

Install specific version

```
renv::install("package@version")
renv::install("user/package@tag")
```

Update to newest version

```
renv::update()
renv::update(packages)
```

Appsilon

Removing packages

Remove based on usage

```
renv::snapshot()
renv::restore(clean = TRUE)
```

Alternatively:

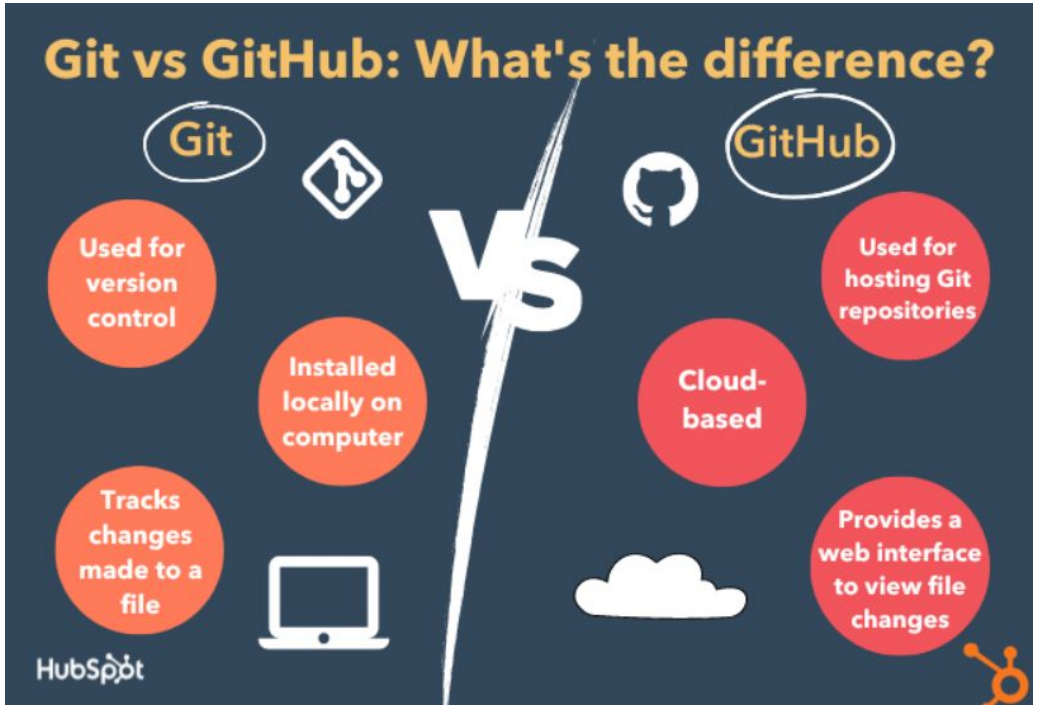
```
renv::remove(package)
renv::snapshot()
```

git

Appsilon

Appsilon

What is git?



Appsilon

Fundamentals (Local & Remote)

🏭 In a factory, there are many production lines, which all lead to packaging.

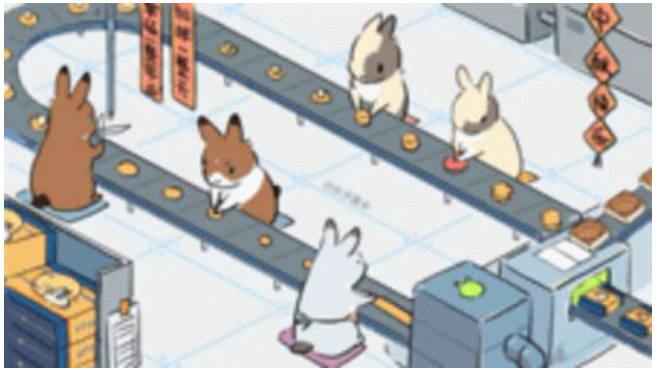
🛠️ Let’s say we want to experiment with a few new techniques but changing everything is impossible. What if we break things or worse, mess the supply? 😱

💡 What if we take one line off the supply and modify it? If it breaks, the supply will not be damaged. If it works, we will modify all lines to be just like this new one!

This is basically Git.

A cut-off line is your local repository

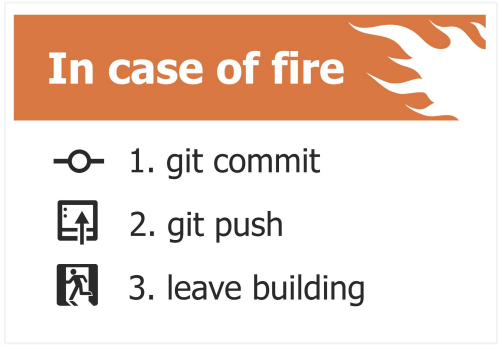
The factory is the remote repository.



Appsilon

Basic commands

`config`	Settings
`init`	Start
`clone`	Copy
`checkout`	Switch
`commit`	Save
`status`	Check
`push`	Send
`pull`	Receive
`merge`	Combine
`rebase`	Modify History



Appsilon

Branching

👨‍🍳 It is the weekend. You invite some friends over and decide to cook a big meal with three courses. But this is a long and complex task with many parts to it. 🤔

💡You divide your kitchen into three parts: one for the soup, one for the chicken, and one for dessert. **(branching)**

A friend arrives early and offers to help with dessert. **(checking out a branch)**

You begin making the soup and get it to simmer. You have some time now. You move to the chicken and start prepping it up. **(switching branches)**

Your friend keeps working on the dessert. **(independent development)**

The meal is served! 🍜🍗🍮

All the elements look and taste perfect. **(merging to production)**

Appsilon

Pull requests & Reviews

📅 Another weekend comes around, and you repeat the plan. Your friend loves cooking with you so they join you ahead of time. 👨‍🍳 👨‍🍳

🔀 You divide your project into three **branches**: one for the salad, one for pasta, one for cake!

You boil the pasta and you complete the sauce, but you are not sure if it is *al dente* (cooked but firm still).

You pull your friend by their arm and ask to come over and check it for you before you add the sauce (**pull request**).

They tell you it can use some more time in the water. (**review and requesting changes**)

A little while later, you call them over again. They say it looks good now (**approval**)

You add the pasta into the sauce. (**merge**) 🥗 🍝 🍰

Appsilon

Merge, Rebase & Conflicts

A skilled painter is commissioned to make a painting for some famous person they have never heard of.

They use references and make a portrait and show it to the agent.

Merge

The agent asks, “where is the dog?”

Apparently, the person being painted goes nowhere without their beloved chihuahua.

The painter adds the chihuahua standing beside the chair. All is good.

Conflict

The agent says, “but the dog is white!”

The painter assumed the chihuahua to be brown since they are often brown in movies.

“This is a conflict!” The painter tells the agent. Now, I must resolve it somehow.

Rebase

He paints over the original and adds a white chihuahua.

They show the agent. The agent says it is perfect, but we must keep it between us.

The client should never know.

The brown chihuahua is thus removed from the history of the painting.

Best Practices

- **Pull Before You Push:** Try to always stay on the recent version of **production,** always rebase your branch regularly, and keep things as clean as you can. This helps avoid conflicts.

- **Conventional Commits** (source): When you commit changes, you must provide a message to describe the changes. Conventional commits make it easier for you to keep track of the changes without even looking at the files changed. With a starter keyword such as **init:** or **test:** or **fix:**, the messages immediately make sense to anyone going through a project's history.

- **Branch Effectively:** It helps to add a directory prefix to branches. Features go in **feat/**, fixes go in **fix/**. For example, **fix/broken-icon-header-module** is a great branch name because it immediately tells you what it does.

  (But also, discuss a branching strategy for each project and stick to it. This is not the be all end all.)

Best Practices

- **Do Not Ignore .gitignore:** The .gitignore file is a special file which tells git which files to not commit or even add. This is helpful for environment variables, credentials and other critical files you might want to avoid pushing to the server. This is also true for artifact files like the pesky .DS_Store in macOS.

- **Resolve Conflicts Locally:** When you resolve conflicts on DevOps, GitHub or GitLab, you commit *before* you test. When you resolve conflicts locally, you commit *after* you test. This helps you avoid unexpected and unintentional breakage in the code.

- **Protect Production:** The production branch should never have the ability to be pushed to directly. It should always have branch protection rules in place and must always need a Pull Request to be changed.

- **Avoid Rewriting Shared History:** Rewriting history in Git is basically changing the original history by commands such as reset, rebase (re*) and then force pushing the code. This is okay if the branches are not shared. Not if someone is on the same branch.

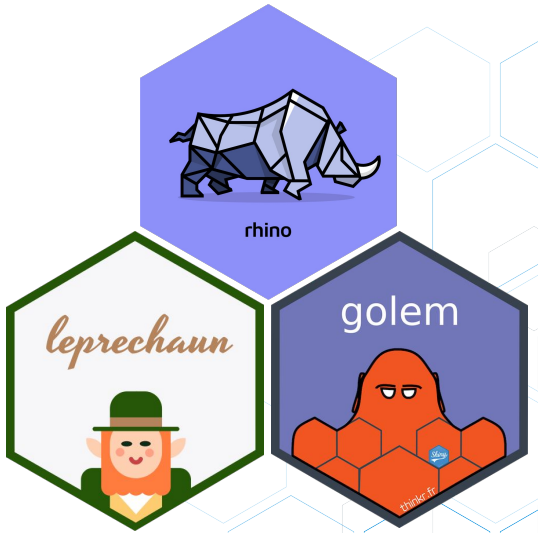
Shiny Structure

Appsilon

Appsilon

How to structure Shiny applications

- No framework
  - Single file
  - Folder structure
- Framework
  - Rhino
  - Golem
  - Leprechaun



Appsilon

How to structure Shiny applications

No framework - Single file

- Hard to extend
- A file can become quite long
- Low reusability
- Good for small applications or testing small concepts

Appsilon

How to structure Shiny applications

No framework - Folder structure

- Easy to extend
- No constraints - full flexibility when it comes to how we want to organise everything
- Requires overhead to structure
- Requires additional documentation to future proof

Golem

Key Features:

- Converts a Shiny app into an R package
- Encourages modular design by organizing code into separate files (e.g., **mod_*** functions)
- Provides utilities for testing, documentation, and deployment.
- Promotes best practices for production, such as separating logic, UI, and configuration
- Includes helpers for adding dependencies (CSS, JavaScript, etc.) and debugging

https://engineering-shiny.org/index.html

Appsilon

Leprechaun

Key Features:

- Provides helpers to add and customize CSS and JavaScript for Shiny apps.
- Includes tools for HTML templating and custom theming.
- Easy integration of design frameworks like Bootstrap or other external libraries.
- Focuses solely on appearance rather than app structure or organization

https://leprechaun.opifex.org/#/

Appsilon

Rhino

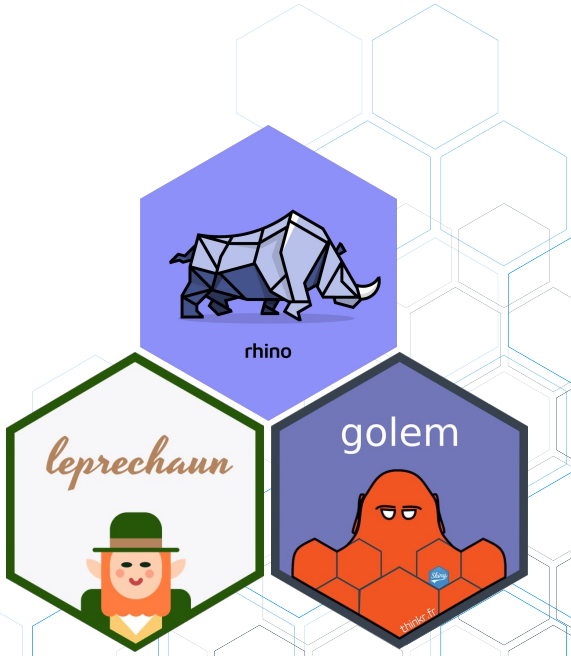
Key Features:

- Folder-based structure (no need to turn the app into an R package).
- Encourages modularization and separation of concerns.
- Built-in support for JavaScript, TypeScript, and CSS customization.
- Focus on collaboration, making it easier to work with developers familiar with web technologies.
- Integrated tools for testing, linting, and dependency management.
- Opinionated: It provides guidance on project organization but allows flexibility.

https://appsilon.github.io/rhino/

Which should you use?

- Use **{golem}** if you're building a large, complex Shiny app that needs to be production-ready, modular, and easy to maintain in the long term.
- Use **{rhino}** if you want a more modern, web-development-inspired approach to Shiny app building and are comfortable with or want to integrate JavaScript/TypeScript.
- Use **{leprechaun}** if your main goal is to customize the appearance of your Shiny app without worrying about its underlying structure or scalability.



Rhino

Appsilon

Appsilon

Why Rhino?

{shiny} is unique!

Appsilon

Why Rhino?

{shiny} is unique!

{shiny} delivery value quickly!

Appsilon

Why Rhino?

{shiny} is unique!

{shiny} delivery value quickly!

{shiny} apps make impact!

Appsilon

Why Rhino?

{shiny} is unique!

{shiny} delivery value quickly!

{shiny} apps make impact!

{shiny} is ready for enterprise!

Appsilon

Why Rhino?

{shiny} is unique!

{shiny} delivery value quickly!

{shiny} apps make impact!

~~**{shiny}** is ready for enterprise!~~

{shiny} is used in enterprise!

Appsilon

Why Rhino?

{shiny} is unique!

{shiny} delivery value quickly!

{shiny} apps make impact!

{shiny} is ready for enterprise!

{shiny} is used in enterprise!

Success is not guaranteed

Appsilon

App’s success factors

Focus on the users → build the “right thing”

Great UI and UX

Great Engineering and Software Quality → Maintainability, Reliability, Development Practices

Appsilon

What we want with Rhino?

Structure

Strict modularity and encapsulation

App NOT as a package

Sensible suggestions

App architecture

Working with configuration

Working with log messages

Full project setup

CI, unit tests, end-2-end tests, linter

Dependencies management

JavaScript and Sass bundling

Appsilon

Rhino

Enterprise-grade
Shiny Application Framework

Key features

Appsilon

Appsilon

Application structure

```
├── rhino.yml
├── .Rprofile                          Infra

├── app.R
├── config.yml                         Sources
├── app
│   ├── main.R
│   ├── logic
│   │   └── __init__.R
│   ├── view
│   │   └── __init__.R
│   ├── static
│   │   └── favicon.ico
│   ├── styles
│   │   └── main.scss
│   └── js
│       └── index.js
```

```
├── renv.lock
├── renv                               Renv
│   └── settings.dcf
├── dependencies.R
├── .renvignore

├── tests
│   ├── testthat                       Tests
│   │   └── test-main.R
│   └── cypress
│       └── integration
│           └── app.spec.js

├── example.Rproj
├── .lintr                             Extra
├── .github
│   └── workflows
│       └── rhino-test.yml
```

Why this matters?

- Solid structure out of the box
- Save time at project start
- Unified structure across apps

Appsilon

Modularization

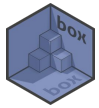
{shiny} modules


```
box::use(
    shiny[moduleServer, NS]
)

#' @export
ui <- function(id) {
    ns <- NS(id)
}

#' @export
server <- function(id) {
    moduleServer(
      id,
      function(input, output, session) {}
    )
}
```

{box} modules




```
box::use(shiny)
box::use(app/view/hello)
```

Why this matters?

- Explicit dependencies between files and packages
- Readable and maintainable code

Appsilon

Compiled JavaScript & styles (ES6, Sass)

```
app.R
app                          Sources
├── main.R
├── logic
│   └── __init__.R
├── view
│   └── __init__.R
├── static
│   └── favicon.ico
├── styles
│   └── main.scss
└── js
    └── index.js
```

```
rhino::build_sass()
```

```
rhino::build_js(
)
```

```
app.R
app                          Sources
├── main.R
├── logic
│   └── __init__.R
├── view
│   └── __init__.R
├── static
│   ├── favicon.ico
│   ├── css
│   │   └── app.min.css
│   └── js
│       └── app.min.js
├── styles
│   └── main.scss
└── js
    └── index.js
```

Why this matters?

- Powerful ES6 and Sass features
- JS & styles automatically included
- Improved application loading speed

Appsilon

Application monitoring: logging

```
box::use(
    rhino[log]
)

log$info("This is an info message.")
log$error("This is an error message.")
```

Why this matters?

- Discover bugs affecting end users
- Investigate bugs in production
- Switch log level as needed

Appsilon

Application quality

`rhino::lint_r()`

r-lib / lintr

`rhino::lint_js()`

ESLint

`rhino::lint_sass()`

stylelint

`rhino::format_r()`

r-lib / styler

`rhino::test_r()`

test that

`rhino::test_e2e()`

cypress

Why this matters?

- High quality from day 1, ready out of the box
- All tools working together
- Consistent code style, avoid bugs when making changes

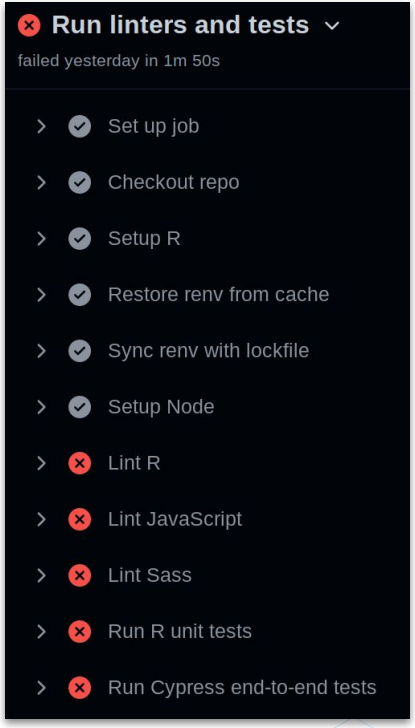
CI configuration - GitHub Actions

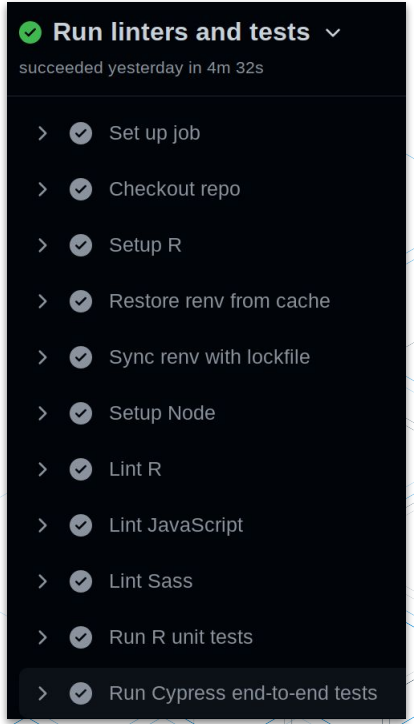
```
example.Rproj
.lintr
.github
└── workflows
    └── rhino-test.yml
```

Extra

Why this matters?

- Make sure all checks are always run
- Reproducibility
- Ready from day 1





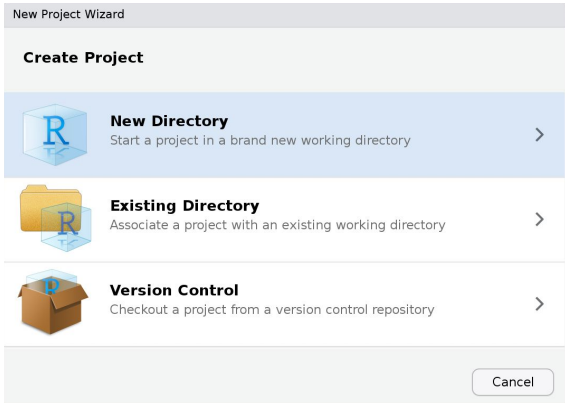
Rhino in practice

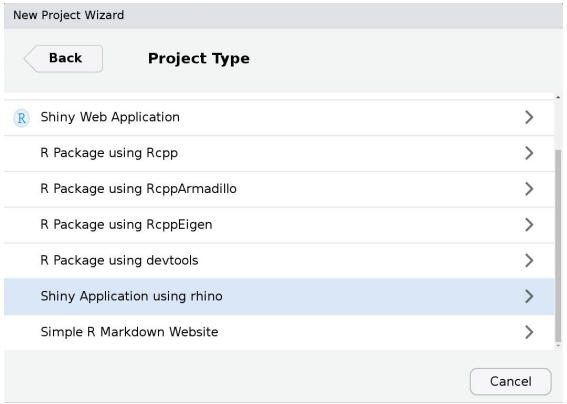
Appsilon

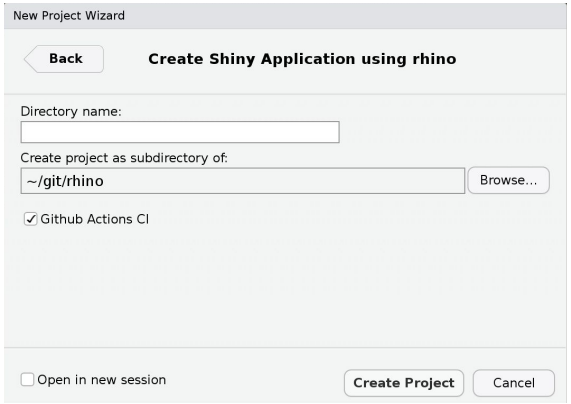
Appsilon

Creating a rhino app

1. RStudio GUI:







2. R console: rhino::init()

Commit all files!

Appsilon

Commands to explore

Lets see what the following commands do:

```
shiny::runApp()
rhino::format_r()
rhino::lint_r()
rhino::build_sass()
rhino::lint_sass()
rhino::build_js()
rhino::lint_js()
rhino::log$info("message")
rhino::test_r()
rhino::test_e2e()
```

Hands-on

Appsilon

Cheeses dataset

1. *Migrate to {rhino}*
    a. *Set up a new {rhino} project*
    b. *Transfer your existing Shiny app components into the {rhino} folder structure*
    c. *Ensure the app runs smoothly in the new structure*
2. *Lint Your Code (R, CSS & JS files)*
    a. *Fix any issues identified during the linting process*
3. *Add Logging*
    a. *Implement logging messages throughout the application*
    b. *Add logs for: User interactions and eactive calculations*

What else can I explore?

Appsilon
Douglas Mesquita

Belo Horizonte
2025-02-03 - 2025-02-06

Appsilon

What else?

Appsilon

shiny.tictoc - https://www.appsilon.com/rhinoverse/shiny-tictoc
profvis - https://profvis.r-lib.org/
shinyloadtest - https://rstudio.github.io/shinyloadtest/
shiny.benchmark - https://github.com/Appsilon/shiny.benchmark
Asynchronous shiny apps - https://hypebright.nl/en/shiny-en/mastering-async-programming
Caching - https://mastering-shiny.org/performance.html#caching
Creating own widgets - https://shiny.posit.co/r/articles/build/js-build-widget/
Using react - https://github.com/Appsilon/shiny.react
Testing
- https://rstudio.github.io/shinytest2/
- https://appsilon.github.io/rhino/articles/tutorial/write-end-to-end-tests-with-cypress.html

More
- https://github.com/grabear/awesome-rshiny
- https://github.com/nanxstats/awesome-shiny-extensions

Now, it’s in your hands

Appsilon
Douglas Mesquita

Belo Horizonte
2025-02-03 - 2025-02-06

Appsilon

Another dataset

1. *Select a dataset from Kaggle, TidyTuesday, or other open sources (e.g., data/superheroes.csv)*
2. *Load and explore the dataset to understand the key variables*
3. *Grab a pen and paper (or use an online tool like Figma) to sketch your app layout*
4. *Define inputs (filters, selections) and outputs (tables, plots, maps, etc.)*
5. *Plan how users will interact with the app*
6. *Set up a new {rhino} project structure*
7. *Build the UI by implementing the layout of your application*
8. *Add widgets to your app*
9. *Customize the CSS and JS if necessary*