FREE eBook

LEARNING shiny

#shiny

Table of Contents

About

Chapter 1: Getting started with shiny

Remarks

This section provides an overview of what shiny is, and why a developer might want to use it.

It should also mention any large subjects within shiny, and link out to the related topics. Since the Documentation for shiny is new, you may need to create initial versions of those related topics.

Examples

Installation or Setup

Shiny can run as a standalone application on your local computer, on a server that can provide shiny apps to multiple users (using shiny server), or on shinyapps.io.

1. **Installing Shiny on a local computer:** in R/RStudio, run `install.packages("shiny")` if installing from CRAN, or `devtools::install_github("rstudio/shiny")` if installing from the RStudio Github repository. The Github repository hosts a development version of Shiny which can possibly have more features when compared to the CRAN version, but it may also be unstable.

When would I use shiny?

1. I have some data analysis done on some data and have many 'non-coding' people on the team, who have similar data like mine, and have similar analysis requirements. In such cases, I can build a web application with shiny, which takes in user specific input data files, and generate analyses.
2. I need to share analyzed data or relevant plots with others in the team. Shiny web apps can be useful in such situations.
3. I don't have significant experience with web application programming, but need to quickly assemble a simple interface. Shiny to the rescue with easy UI and server elements and minimum coding.
4. Interactive elements allow your users to explore what element of the data is relevant to them. For example, you could have data for the whole company loaded, but have a dropdown per department like "Sales", "Production", "Finance" that can summarise the data the way the users want to view it. The alternative would be producing a huge report pack with analyses for each department, but they only read their chapter and the total.

Simple App

Each `shiny` app contains two parts: A user interface definition (`UI`) and a server script (`server`). This example shows how you can print "Hello world" from UI or from server.

UI.R

In the UI you can place some view objects (div, inputs, buttons, etc).

```
library(shiny)

# Define UI for application print "Hello world"
shinyUI(

  # Create bootstrap page
  fluidPage(

    # Paragraph "Hello world"
    p("Hello world"),

    # Create button to print "Hello world" from server
    actionButton(inputId = "Print_Hello", label = "Print_Hello World"),

    # Create position for server side text
    textOutput("Server_Hello")

  )
)
```

Server.R

In the server script you can define methods which manipulate data or listen to actions.

```
# Define server logic required to print "Hello World" when button is clicked
shinyServer(function(input, output) {

  # Create action when actionButton is clicked
  observeEvent(input$Print_Hello,{

    # Change text of Server_Hello
    output$Server_Hello = renderText("Hello world from server side")
  })


})
```

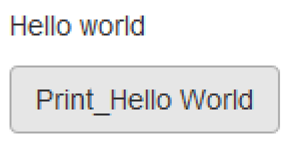
How to run?

You can run your app in several ways:

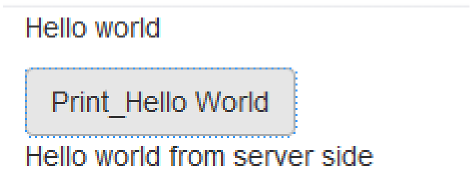
1. Create two different files and place them into one directory, then use `runApp('your dir path')`
2. You can define two variables (ui and server, for example) and then use `shinyApp(ui,server)` to run your app

Result

In this example you will see some text and a button:


Hello world

And after button click the server responds:


Hello world

Hello world from server side

Including plots

The simplest way to include plots in your shinyApp is to use `plotOutput` in the ui and `renderPlot` in the server. This will work with base graphics as well as `ggPlot`s

```
library(shiny)
library(ggplot2)

ui <- fluidPage(
  plotOutput('myPlot'),
  plotOutput('myGgPlot')
)

server <- function(input, output, session){
  output$myPlot = renderPlot({
    hist(rnorm(1000))
  })
  output$myGgPlot <- renderPlot({
    ggplot(iris, aes(Sepal.Length, Sepal.Width, color = Species)) + geom_point()
  })
}

shinyApp(ui, server)
```

Including tables

Tables are most easily included with the DT package, which is an R interface to the JavaScript library DataTables.

```
library(shiny)
library(DT)

ui <- fluidPage(
  dataTableOutput('myTable')
)

server <- function(input, output, session){
  output$myTable <- renderDataTable({
    datatable(iris)
  })
}

shinyApp(ui, server)
```

Read Getting started with shiny online: https://riptutorial.com/shiny/topic/2667/getting-started-with-shiny

Chapter 2: How to write MCVE (Minimal, Complete, and Verifiable example) Shiny apps

Introduction

If you are having issues with your Shiny apps, it is good practice to create an app that illustrates your point. This app should be as simple as possible while still reflecting your problem. This means using simple datasets, self-explanatory naming (especially for I/O IDs) and replacing plots with simpler ones.

It is also advisable to create your MCVE in a way that as little non-standard libraries as possible are required.

Examples

Basic structure

MCVE's should start the Shiny app when they are copied in the console. An easy way to do this is using the `shinyApp` function. For example:

> why is my checkbox not responding?

```
library(shiny)

ui <- fluidPage(
  checkboxInput('checkbox', 'click me'),
  verbatimTextOutput('text')
)

server <- function(input, output, session){
  output$text <- renderText({
    isolate(input$checkbox)
  })
}

shinyApp(ui, server)
```

Alternatively, you can also not assign variables to `ui` and `server`.

```
library(shiny)

shinyApp(
  fluidPage(
    checkboxInput('checkbox', 'click me'),
    verbatimTextOutput('text')
  ),
  function(input, output, session){
    output$text <- renderText({
      isolate(input$checkbox)
    })
  }
)

shinyApp(ui, server)
```

Avoid unnecessary details

In practice, shiny Apps are often very complicated and full of features that have been developed over time. More often than not, those additional details are not necessary to reproduce your issue. It is best if you skip such details when writing MCVE's.

WRONG

> Why is my plot not showing?

```
library(shiny)
library(ggplot2)

ui <- fluidPage(
  plotOutput('plot')
)

server <- function(input, output, session){
  df <- data.frame(treatment = rep(letters[1:3], times = 3),
                   context = rep(LETTERS[1:3], each = 3),
                   effect = runif(9,0,1))
  df$treat.con <- paste(df$treatment,df$context, sep = ".")
  df$treat.con <- reorder(df$treat.con, -df$effect, )
  output$plot = renderPlot({
    myPlot <- ggplot(df, aes(x = treat.con, y = effect)) +
       geom_point() +
       facet_wrap(~context,
                  scales = "free_x",
                  ncol = 1)
  })
}

shinyApp(ui, server)
```

RIGHT

> Why is my Plot not showing?

```
library(shiny)
library(ggplot2)

ui <- fluidPage(
  plotOutput('plot')
)

server <- function(input, output, session){
  output$plot = renderPlot({
    myPlot <- ggplot(mtcars, aes(mpg, wt)) + geom_point()
  })
}

shinyApp(ui, server)
```

Read How to write MCVE (Minimal, Complete, and Verifiable example) Shiny apps online: https://riptutorial.com/shiny/topic/10653/how-to-write-mcve--minimal--complete--and-verifiable-example--shiny-apps

Chapter 3: Javascript API

Syntax

- session$sendCustomMessage(*name*,*list of parameters*)
- Shiny.addCustomMessageHandler(*name*, *JS function that accepts list of parameters*)
- Shiny.onInputChange(*name*,*value*)

Examples

Sending data from server to client

In many instances, you will want to send data from the R server to the JS client. Here is a very simple example:

```
library(shiny)
runApp(
  list(
    ui = fluidPage(
      tags$script(
        "Shiny.addCustomMessageHandler('message', function(params) { alert(params); });"
      ),
      actionButton("btn","Press Me")
    ),
    server = function(input, output, session) {
      observeEvent(input$btn,{
        randomNumber <- runif(1,0,100)
        session$sendCustomMessage("message",list(paste0(randomNumber," is a random number!")))
      })
    }
  )
)
```

The workhorses here are the `session$sendCustomMessage` function in `R` and the `Shiny.addCustomMessageHandler` function in `javascript`.

The `session$sendCustomMessage` function lets you send parameters from `R` to a `javascript` function, and `Shiny.addCustomMessageHandler` let's you define the `javascript` function that accepts the parameters from `R`.

Note: Lists are converted to JSON when they are passed from `R` to `javascript`

Sending data from client to server

In some instances, you will want to send data from JS client to the R server. Here is a basic example using javascript's `Shiny.onInputChange` function:

```
library(shiny)
runApp(
  list(
    ui = fluidPage(
      # create password input
      HTML('<input type="password" id="passwordInput">'),
      # use jquery to write function that sends value to
      # server when changed
      tags$script(
        '$("#passwordInput").on("change",function() {
          Shiny.onInputChange("myInput",this.value);
        })'
      ),
      # show password
      verbatimTextOutput("test")
    ),
    server = function(input, output, session) {
      # read in then show password
      output$test <- renderPrint(
        input$myInput
      )
    }
  )
)
```

Here we create a password input with id `passwordInput`. We add a Javascript function on the UI that reacts to changes in `passwordInput`, and sends the value to the server using `Shiny.onInputChange`.

`Shiny.onInputChange` takes two parameters, a name for the `input$*name*`, plus a value for `input$*name*`

Then you can use `input$*name*` like any other Shiny input.

Read Javascript API online: https://riptutorial.com/shiny/topic/3149/javascript-api

Chapter 4: reactive, reactiveValue and eventReactive, observe and observeEvent in Shiny

Introduction

reactive, reactiveValue and eventReactive are various kinds of reactive expressions in Shiny. They yield output which can be used as input in other expressions, which will in turn take a dependency on the reactive expression.

observe and observeEvent are similar to reactive expressions. The big difference is that the observers do not yield any output and thus they are only useful for their side effects.

Examples of their use are given in this document.

Examples

reactive

A reactive can be used to make output depend on another expression. In the example below, the output$text element is dependent on text_reactive, which in turn is dependent on input$user_text. Whenever input$user_text changes, output$text element and text_reactive become **invalidated.** They are recalculated based on the new value for input$user_text.

```
library(shiny)

ui <- fluidPage(
  headerPanel("Example reactive"),

  mainPanel(

    # input field
    textInput("user_text", label = "Enter some text:", placeholder = "Please enter some
text."),

    # display text output
    textOutput("text"))
)

server <- function(input, output) {

  # reactive expression
  text_reactive <- reactive({
    input$user_text
  })

  # text output
  output$text <- renderText({
    text_reactive()
  })
}

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

eventReactive

eventReactives are similar to reactives, they are constructed as follows:

```
eventReactive( event {
code to run
})
```

eventReactives are not dependent on all reactive expressions in their body ('*code to run*' in the snippet above). Instead, they are only dependent on the expressions specified in the *event* section.

In the example below, we have added a submit button, and created an eventReactive. Whenever input$user_text changes, the eventReactive is not invalidated, since the eventReactive is only dependent on the actionButton input$submit. Whenever that button is pressed, text_reactive and subsequently output$text are invalidated, and will be recalulated based on the updated input$user_text.

```
library(shiny)

ui <- fluidPage(
  headerPanel("Example eventReactive"),

  mainPanel(

    # input field
    textInput("user_text", label = "Enter some text:", placeholder = "Please enter some
text."),

    # submit button
    actionButton("submit", label = "Submit"),

    # display text output
    textOutput("text"))
)

server <- function(input, output) {

  # reactive expression
  text_reactive <- eventReactive( input$submit, {
    input$user_text
  })

  # text output
  output$text <- renderText({
    text_reactive()
  })
}
```

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

reactiveValues

reactiveValues can be used to store objects, to which other expressions can take a dependency.

In the example below, a reactiveValues object is initialized with value "No text has been submitted yet.". A separate observer is created to update the reactiveValues object whenever the submit button is pressed. Note that the reactiveValues itself does not take a dependency on the expressions in its body.

```
library(shiny)

ui <- fluidPage(
  headerPanel("Example reactiveValues"),

  mainPanel(

    # input field
    textInput("user_text", label = "Enter some text:", placeholder = "Please enter some
text."),
    actionButton("submit", label = "Submit"),

    # display text output
    textOutput("text"))
)

server <- function(input, output) {

  # observe event for updating the reactiveValues
  observeEvent(input$submit,
               {
    text_reactive$text <- input$user_text
  })

  # reactiveValues
  text_reactive <- reactiveValues(
    text = "No text has been submitted yet."
  )

  # text output
  output$text <- renderText({
    text_reactive$text
  })
}

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

observeEvent

An observeEvent object can be used to trigger a piece of code when a certain event occurs. It is constructed as:

```
observeEvent( event {
code to run
})
```

The observeEvent will only be dependent on the *'event'* section in the small piece of code above. It will not be dependent on anything in the '*code to run*' part. An example implementation can be found below:

```
library(shiny)

ui <- fluidPage(
  headerPanel("Example reactive"),

  mainPanel(

    # action buttons
    actionButton("button1","Button 1"),
    actionButton("button2","Button 2")
  )
)

server <- function(input, output) {

  # observe button 1 press.
  observeEvent(input$button1, {
    # The observeEvent takes no dependency on button 2, even though we refer to the input in
the following line.
    input$button2
    showModal(modalDialog(
      title = "Button pressed",
      "You pressed one of the buttons!"
    ))
  })
}

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

observe

An observe expression is triggered every time one of its inputs changes. The major difference with regards to a reactive expression is that it yields no output, and it should only be used for its side effects (such as modifying a reactiveValues object, or triggering a pop-up).

Also, note that observe does not ignore NULL's, therefore it will fire even if its inputs are still NULL. observeEvent by default does ignore NULL, as is almost always desirable.

```
library(shiny)

ui <- fluidPage(
  headerPanel("Example reactive"),

  mainPanel(

    # action buttons
    actionButton("button1","Button 1"),
    actionButton("button2","Button 2")
  )
)
```

```
server <- function(input, output) {

  # observe button 1 press.
  observe({
    input$button1
    input$button2
    showModal(modalDialog(
      title = "Button pressed",
      "You pressed one of the buttons!"
    ))
  })
}

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

Read reactive, reactiveValue and eventReactive, observe and observeEvent in Shiny online: https://riptutorial.com/shiny/topic/10787/reactive--reactivevalue-and-eventreactive--observe-and-observeevent-in-shiny

Chapter 5: Upload Data to shiny

Examples

Upload .RData Files to shiny with fileInput()

The example allows you to upload .RData files. The approach with `load` and `get` allows you to assign the loaded data to a variable name of your choice. For the matter of the example being "standalone" I inserted the top section that stores two vectors to your disk in order to load and plot them later.

```
library(shiny)

# Define two datasets and store them to disk
x <- rnorm(100)
save(x, file = "x.RData")
rm(x)
y <- rnorm(100, mean = 2)
save(y, file = "y.RData")
rm(y)

# Define UI
ui <- shinyUI(fluidPage(
  titlePanel(".RData File Upload Test"),
  mainPanel(
    fileInput("file", label = ""),
    actionButton(inputId="plot","Plot"),
    plotOutput("hist"))
  )
)

# Define server logic
server <- shinyServer(function(input, output) {

  observeEvent(input$plot,{
    if ( is.null(input$file)) return(NULL)
    inFile <- input$file
    file <- inFile$datapath
    # load the file into new environment and get it from there
    e = new.env()
    name <- load(file, envir = e)
    data <- e[[name]]

    # Plot the data
    output$hist <- renderPlot({
      hist(data)
    })
  })
})

# Run the application
shinyApp(ui = ui, server = server)
```

Uploading csv files to Shiny

It is also possible to have an user upload csv's to your Shiny app. The code below shows a small example on how this can be achieved. It also includes a radioButton input so the user can interactively choose the separator to be used.

```
library(shiny)
library(DT)

# Define UI
ui <- shinyUI(fluidPage(

  fileInput('target_upload', 'Choose file to upload',
            accept = c(
              'text/csv',
              'text/comma-separated-values',
              '.csv'
            )),
  radioButtons("separator","Separator: ",choices = c(";",",",":"), selected=";",inline=TRUE),
  DT::dataTableOutput("sample_table")
)
)

# Define server logic
server <- shinyServer(function(input, output) {

  df_products_upload <- reactive({
    inFile <- input$target_upload
    if (is.null(inFile))
      return(NULL)
    df <- read.csv(inFile$datapath, header = TRUE,sep = input$separator)
    return(df)
  })

  output$sample_table<- DT::renderDataTable({
    df <- df_products_upload()
    DT::datatable(df)
  })

}
)

# Run the application
shinyApp(ui = ui, server = server)
```

Read Upload Data to shiny online: https://riptutorial.com/shiny/topic/7576/upload-data-to-shiny

Credits

S. No	Chapters	Contributors
1	Getting started with shiny	Batanichek, Bogdan Rau, chrki, Community, Florian, Gregor de Cillia, jsb, Mathias711, micstr, sigmabeta
2	How to write MCVE (Minimal, Complete, and Verifiable example) Shiny apps	Florian, Gregor de Cillia
3	Javascript API	Carl, Tomás Barcellos
4	reactive, reactiveValue and eventReactive, observe and observeEvent in Shiny	Florian
5	Upload Data to shiny	Florian, symbolrush