**Building Web Applications in R with Shiny**

**Reactive Programming**

**Reactive Sources**

User input that comes through a browser interface.

Sources can only be parents.

**Reactive Conductors**

Reactive component between a source and an endpoint.

Conductors can be both a dependent (child) and have dependents (parent).

**Reactive Endpoints**

Something that appears in the user’s browser window (e.g. a plot or a table)

Endpoints can only be children.

One reactive source can be connected to multiple endpoints. Also one reactive endpoint can be connected to multiple sources.

**reactive**() when the same dataset is used in multiple outputs, it makes sense to make our code more efficient by using a reactive data frame. reactive() is a reactive conductor. reactive() is an implementation of reactive conductors.

Example: movies\_selected <- reactive({

req(input$selected\_var)

movies %>% select(input$selected\_var)

})

correlated

output$moviestable <- DT::renderDataTable({

req(input$selected\_var)

DT::datatable(data = movies\_selected() %>% select(input$selected\_var),

options = list(pageLength = 10),

rownames = FALSE)

})

Why use reactives?

1. By using a reactive expression for the subsetted data frame, you can subset once and reuse the subsetted result multiple times. This makes code far more efficient by doing away with the need for repeating code
2. Decompose large, complex calculations into smaller pieces to make them more understandable. This also allows breaking down large complex R scripts into a series of small functions that build on each other.

**Reactive Programming (continued)**

**Functions versus Reactives**

Each time you call a function, R will evaluate it. On the contrary, reactive expressions are lazy in that they only get executed when their input changes. Even if you call a reactive expression multiple times, it only re-executes when its input(s) change.

Using many reactive expressions can create a complicated dependency structure in your app.

**reactlog** is a graphical representation of this dependency structure. To view reactlog:

1. In a fresh R session, run options(shiny.reactlog = TRUE)
2. Then, launch you app as you normally would
3. In the app, press Ctrl + F3

**observeEvent**() and **updateNumericInput**() example:

observeEvent(input$selected\_type, {

n\_max <- nrow(movies\_selected())

updateNumericInput(session, “n\_samp”, max = n\_max)

updateNumericInput(session, “n\_samp”,

label = paste0(“Sample size (max = “, n\_max, “):”))

})

**sample\_n**()

sample\_n(movies\_selected(), input$n\_samp)

**reactiveValues**() is an implementation of reactive sources. For example, input$\* is a reactive value that looks like a list and contains many individual reactive values that are set by input from the web browser.

**reactive**() is an implementation of reactive conductors. For example, reactive data frame subsets. reactive() can access values or other reactive expressions and they return a value. reactive() is useful for caching the results of any procedure that happens in response to user input.

**observe()** is an implementation of reactive endpoints. An output$\* object is an observer. observe() can access reactive sources and reactive expressions, but they don’t return a value. observe() is used for its side effects, which primarily involves sending data to the web browser.

**Reactives versus Observers**

* Both store expressions that can be executed.
* Reactive expressions return values whereas observers do not return values.
* Observers and endpoints in general eagerly respond to changes in their dependencies whereas reactive expressions and conductions in general do not.
* Reactive expressions must not have side effects, while observers are only useful for their side effects. reactive() is used for calculating values without side effects. observe() is used for performing actions with side effects.
* Do not use an observe() when calculating a value and especially do not use reactive() for performing actions with side effects.

**Stops, Triggers, and Delay**

**Isolating Reactions**

output$scatterplot <- renderPlot({

ggplot(data = movies\_subset(), aes\_string(x = input$x, y = input$y)) +

geom\_point() +

labs(title = isolate({input$plot\_title})

})

Plot title will update when any of the inputs other than input$plot\_title changes. However, the plot title will not update when only input$plot\_title changes. Isolate can take as its single argument an expression that accesses reactive values or expressions.

Isolate executes expr in a scope where reactive values or expression can be read, but they do not trigger an output that depends on them to be re-evaluated. However, if that output depends on other reactives as well, when one of those changes, the output is re-evalued with the new value of the isolated expr.

**Triggering Reactions**

expression to call whenever eventExpr is invalidated, code is put in {}

**observeEvent**(eventExpr, handlerExpr, …) you might want to do this, for example, when an action button is clicked.

* simple reactive value, for example, input$click
* call to a reactive express, for example, df()
* complex expression inside {}

**Delaying Reactions**

**eventReactive**(eventExpr, handlerExpr, ignoreNULL = FALSE, …)

when ignoreNULL is set to FALSE, the app initially performs the action/calculation in the handlerExpr when it is first launched and then subsequently waits for eventExpr to be invalidated before re-executing handlerExpr.

**Customizing Appearance**

Embedding HTML in your app using HTML tags is a great way to customize the appearance of your app. Currently, there are 110 HTML tabs supported in Shiny.

**tags$h3**(…) h3(“…”) is a level 3 header tag

**tags$br**() br() line break tag

**tags$hr**() hr() horizontal line separator

Images can added use the **img**() tag.

img(src = "https://www.rstudio.com/wp-content/uploads/2014/04/shiny.png", height = "30px")

**Layout Panels**

Use panels to group multiple elements into a single element that has its own properties. Panels are especially important and useful for complex apps with a large number of inputs and outputs such that it might not be clear to the user where to get started. There are twelve panel types in Shiny.

Note that you can use the width argument with sidebarPanel and mainPanel.

**wellPanel**() is useful for grouping related UI widgets into one section.

**titlePanel**() example 🡪 titlePanle(“Hello World!”, windowTitle = “Hello World Test”)

**conditionalPanel**() conditionalPanel(“input.show\_data == true”, h3(“Data table”)),

**Javascript syntax**

**fluidRow**() partitions a panel into rows

**column**() partitions a panel in columns

**Tabs and Tabsets**

mainPanel(

tabsetPanel( type = "tabs",

id = “tabsetpanel”,

tabPanel("Plot",

plotOutput(outputId = "scatterplot"),

br(),

h5(textOutput("description"))

),

tabPanel("Data",

DT::dataTableOutput(outputId = "moviestable")

)

)

)

**Tabs and Tabsets (continued)**

In the server function, you can make tabs display conditionally…

observeEvent(input$show\_data, {

if(input$show\_data) {

showTab(inputId = “tabsetpanel”, target = “Data”, select = TRUE)

} else {

hideTab(inputId = “tabsetpanel”, target = “Data”)

}

})

Using themes is a quick and easy way of customizing the appearance of your app. The shinythemes package allows you to implement various themes. <https://rstudio.github.io/shinythemes/> is a good resource of view different theme options.

library(shinythemes)

To try out various themes add the themeSelector() widget to your ui component.

Try out various themes. Once you decide on a theme, remove the themeSelector() widget and then apply the shinytheme() function…

theme = shinytheme(“cosmo”) this snippet goes into the ui component

Go to <https://shiny.rstudio.com/deploy> to find out different ways to deploy your app.