



ADVANCED REACTIVITY

OUTLINE

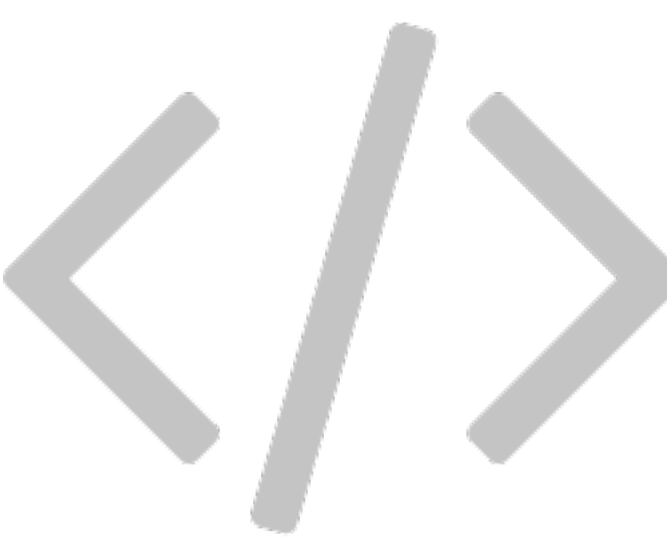
- ▶ Stop - trigger - delay
 - ▶ `isolate()`
 - ▶ `observeEvent()`
 - ▶ `eventReactive()`
- ▶ Performance considerations
 - ▶ Schedule with `invalidateLater()`
 - ▶ Monitor with `reactivePoll()`
 - ▶ `reactiveFileReader()`
- ▶ Reactivity best practices

Stop - trigger -
delay

**Stop with
isolate()**

REACTIVE VS. ISOLATE

- ▶ **reactive()** creates output that is reactive, **isolate()** creates output that is not reactive
- ▶ **isolate()** also takes in a chunk of code, but the result does not respond to a reactive value in the code



Only update plot title when other components of the plot are also updated. See **movies_13.R**.

server:

```
pretty_plot_title <- reactive({ toTitleCase(input$plot_title) })  
output$scatterplot <- renderPlot({  
  ggplot(data = movies_subset(), aes_string(x = input$x, y = input$y, color = input$z)) +  
    geom_point(alpha = input$alpha, size = input$size) +  
    labs(title = isolate({ pretty_plot_title() }))  
})
```

Plot title will update
when any of the other **inputs**
in this chunk change

Plot title will **not** update
when **input\$plot_title**
changes

**Trigger with
observeEvent()**


TRIGGERING A REACTION

- ▶ **observeEvent()** can be used to trigger a reaction
- ▶ It uses a different syntax

```
observeEvent(eventExpr, handlerExpr, ...)
```

simple reactive value - **input\$click**,
call to reactive expression - **df()**,
or complex expression inside **{}**

expression to call whenever
eventExpr is invalidated



Download a CSV of the sampled data when action button is pressed. See **movies_14.R**.

ui:

```
actionButton(inputId = "download_csv", label = "Download CSV")
```

server:

```
observeEvent(eventExpr = input$download_csv,  
             handlerExpr = { write.csv(movies_sample(),  
                                       file = paste("movies_", Sys.time(), ".csv"),  
                                       row.names = FALSE) }  
)
```

ISOLATE VS. OBSERVEEVENT

- ▶ **isolate()** is used to stop a reaction
- ▶ while **observeEvent()** is used to perform an **action** in response to an event
 - ▶ Note: "recalculate a value" does not generally count as performing an action, we'll next discuss **eventReactive()** for that

**Delay reactions with
eventReactive()**

OBSERVEEVENT VS. EVENTREACTIVE

- ▶ **observeEvent()** is to to perform an **action** in response to an event
- ▶ while **eventReactive()** is used to create a **calculated value** that only updates in response to an event
 - ▶ Just like a normal reactive expression except only invalidates in response to the given event.

```
observeEvent(eventExpr, valueExpr, ...)
```

EXERCISE



- ▶ Change how the random sample is generated such that it is updated when the user clicks on an action button that says “Get new sample”.
- ▶ Use **movies_14.R** as the basis of the script and make the updates there.
- ▶ Run the app to ensure that the behavior is as described
- ▶ Compare your code / output with the person sitting next to / nearby you

5_m 00_s



SOLUTION

Solution can also be found in `movies_15.R`.

ui:

```
actionButton(inputId = "get_new_sample", label = "Get new sample")
```

server:

```
movies_sample <- eventReactive(eventExpr = input$get_new_sample,  
                               valueExpr = {  
                                 movies_subset() %>%  
                                   sample_n(input$n_samp)  
                               },  
                               ignoreNULL = FALSE  
)
```

Why
would it not work to
use observeEvent
here?

Initially perform the action/
calculation and just let the
user re-initiate it (like a
"Recalculate" button)


Performance

considerations

**Schedule with
invalidateLater()**

INVALIDATELATER

- ▶ If this is placed within an observer or reactive expression, that object will be invalidated (and re-execute) after the interval has passed
- ▶ The re-execution will reset the invalidation flag, so in a typical use case, the object will keep re-executing and waiting for the specified interval.
- ▶ It's possible to stop this cycle by adding conditional logic that prevents the **invalidateLater()** from being run.



Tell the user how long they have been viewing your app for. See **movies_16.R**.

ui:

```
textOutput(outputId = "time_elapsed")
```

server:

```
# Calculate time difference between when app is first launched and now
beg <- reactive({ Sys.time() })
now <- reactive({ invalidateLater(millis = 1000); Sys.time() })
diff <- reactive({ round(difftime(now(), beg(), units = "secs")) })

# Print time viewing app
output$time_elapsed <- renderText({
  paste("You have been viewing this app for", diff(), "seconds.")
})
```

EXERCISE



- ▶ Change how the random sample is generated such that it is updated every 5 seconds
 - ▶ Don't forget to remove now unused functionality for the action button to get a new sample
- ▶ Use **movies_16.R** as the basis of the script and make the updates there
- ▶ Run the app to ensure that the behavior is as described
- ▶ Compare your code / output with the person sitting next to / nearby you

5_m 00_s



SOLUTION

Solution can also be found in `movies_17.R`.

ui:

```
actionButton(inputId = "get_new_sample", label = "Get new sample")
```

server:

```
# Get new sample every 5 seconds
movies_sample <- reactive({ invalidateLater(millis = 5000)
  movies_subset() %>%
    sample_n(input$n_samp)
})
```


**Monitor with
reactivePoll()**

REACTIVEPOLL

- ▶ **reactivePoll()** pairs a relatively cheap "check" function with a more expensive value retrieval function
 - ▶ **Check function:** is executed periodically and should always return a consistent value until the data changes
 - ▶ Note doesn't return **TRUE** or **FALSE**, instead it indicates change by returning a different value from the previous time it was called
 - ▶ **Value retrieval function:** is used to re-populate the data when the check function returns a different value
- ▶ Similar to **invalidateLater()**, but it's based on a change in a file as opposed to a periodic change



Periodically check and report the names and dimensions of CSV files in the directory.


1. Write the check and value retrieval functions for **reactivePoll()**
2. Count and list CSV files in the directory every 5 seconds with **reactivePoll()**
3. Store CSV files in the directory as a data table in **output\$csv_files**
4. Print **output\$csv_files** in the UI, use tabs to reduce clutter

1. Write the check and value retrieval functions for reactivePoll()

```
# Check function
count_files <- function(){ length(dir(pattern = "*.csv")) }

# Value retrieval function
list_files <- function(){
  files <- dir(pattern = "*.csv")
  if(length(files) == 0){ return( data.frame() ) }
  sapply(files, function(file) dim(read.csv(file))) %>%
    unlist() %>%
    t() %>%
    as.data.frame() %>%
    setNames(c("rows", "cols"))
}
```

There are many ways of doing this, don't focus too much on this code



2. Count and list CSV files in the directory every 5 seconds with `reactivePoll()`

```
# Count and list CSV files in the directory every 5 seconds
csv_files <- reactivePoll(intervalMillis = 5000,
  session,
  checkFunc = count_files,
  valueFunc = list_files)
```

3. Store CSV files in the directory as a data table in output\$csv_files

```
# Print CSV files in the directory
output$csv_files <- DT::renderDataTable(
  DT::datatable(data = csv_files(),
    options = list(pageLength = 10),
    rownames = TRUE)
```


4. Print `output$csv_files` in the UI, use tabs to reduce clutter

```
# Use tabs for the data tables to reduce clutter
tabsetPanel(
  # Show data table
  tabPanel("Plotted data", dataTableOutput(outputId = "moviestable")),

  # Show CSV files in directory
  tabPanel("Files in directory", dataTableOutput(outputId = "csv_files"))
)
```

This is new syntax we haven't
seen before



Putting it all together...

`movies_18.R`

See it in action: Change sample size, get new sample, download data, check out the “Files in directory” tab. Then, delete all CSV files in directory, and see the list update.

reactiveFileReader()

REACTIVEFILEREADER

- ▶ **reactiveFileReader()** works by periodically checking the file's last modified time
 - ▶ If the file has changed, it is re-read and any reactive dependents are invalidated
- ▶ Also similar to **invalidateLater()** but instead of periodic updates, updates are based on changes in a file

Reactivity

best practices



EXERCISE

Is there something wrong with this? If so, what?

```
ui <- fluidPage(  
  titlePanel("Add 2"),  
  sidebarLayout(  
    sidebarPanel( sliderInput("x", "Select x", min = 1, max = 50, value = 30) ),  
    mainPanel( textOutput("x_updated") )  
  )  
)  
  
server <- function(input, output) {  
  add_2 <- function(x) { x + 2 }  
  current_x <- add_2(input$x)  
  output$x_updated <- renderText({ current_x })  
}
```

1m 00s



SOLUTION

Yup! See `add_2.R`.

```
ui <- fluidPage(
  titlePanel("Add 2"),
  sidebarLayout(
    sidebarPanel( sliderInput("x", "Select x", min = 1, max = 50, value = 30) ),
    mainPanel( textOutput("x_updated") )
  )
)

server <- function(input, output) {
  add_2 <- function(x) { x + 2 }
  current_x <- reactive({ add_2(input$x) })
  output$x_updated <- renderText({ current_x() })
}
```

LESSON 1

Reactives are equivalent to no argument functions

Think about them as functions, think about them as variables that can depend on user input and other reactives



EXERCISE

`observe()` vs. `reactive()`

Which one should you use if you want to create an object that you can later use in a render function?

Which one if you want to update the minimum value of a slider input based on the choices a user makes in the app?

1_m 00_s



SOLUTION

`observe()` vs. `reactive()`

Which one should you use if you want to create an object that you can later use in a render function?

`reactive()`

Which one if you want to update the minimum value of a slider input based on the choices a user makes in the app?

`observe()`

LESSON 2

Reactives are for reactive values and expressions

Observers are for their side effects



EXERCISE

Is there something wrong with this? If so, what?

```
ui <- fluidPage(  
  sidebarLayout(  
    sidebarPanel(sliderInput("n", "Select n", min = 1,  
                             max = 50, value = 30)),  
    mainPanel(  
      plotOutput("hist"),  
      textOutput("med")  
    )  
  )  
)
```

```
server <- function(input, output) {  
  dist <- reactive({ rnorm(input$n) })  
  output$hist <- renderPlot({  
    hist(dist())  
    med <- reactive({ median(dist()) })  
    abline(v = med(), col = "red")  
  })  
  output$med <- renderText({  
    paste("The median is", round(med(), 3))  
  })  
}
```



SOLUTION

Oh yeah! See `hist_med.R`.

```
ui <- fluidPage(  
  sidebarLayout(  
    sidebarPanel(sliderInput("n", "Select n", min = 1,  
                             max = 50, value = 30)),  
    mainPanel(  
      plotOutput("hist"),  
      textOutput("med")  
    )  
  )  
)
```

```
server <- function(input, output) {  
  dist <- reactive({ rnorm(input$n) })  
  med <- reactive({ median(dist()) })  
  output$hist <- renderPlot({  
    hist(dist())  
    abline(v = med(), col = "red")  
  })  
  output$med <- renderText({  
    paste("The median is", round(med(), 3))  
  })  
}
```


LESSON 3

Do not define a **reactive()** inside a **render*()** function



ADVANCED REACTIVITY

OTHER NOTES FOR BEST PRACTICES

didn't add these
in yet, maybe
there's enough
for an hour
already?

- do not define a `reactive()` or `observer()` inside a `reactive()`
- things nested inside a `reactive()` are not reactive
 - reactives are designed for top level
 - `v <- reactiveValues(foo = list())`
 - `vfoobar <- 10`