BUILDING WEB APPLICATIONS IN R WITH SHINY

Reactive elements

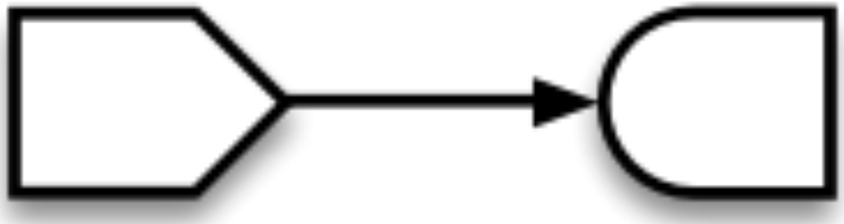
Reactive objects

Reactive source

Reactive conductor

Reactive endpoint

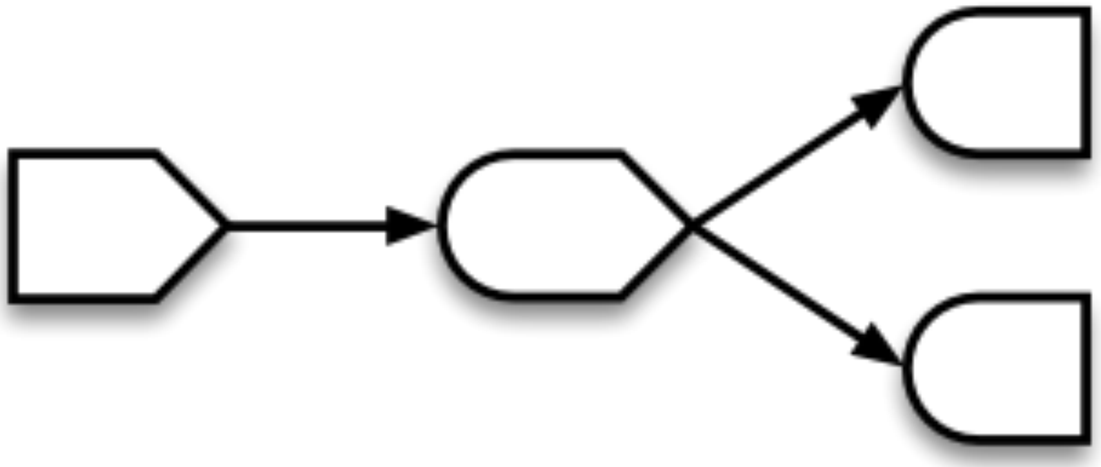
Reactive sources and endpoints



- **Reactive source:** User input that comes through a browser interface, typically
- **Reactive endpoint:** Something that appears in the user's browser window, such as a plot or a table of values
- One reactive source can be connected to multiple endpoints, and vice versa

Reactive conductors

- **Reactive counductor:** Reactive component between a source and an endpoint
- A conductor can both be a dependent (child) and have dependents (parent)
  - Sources can only be parents (they can have dependents)
  - Endpoints can only be children (they can be dependents)

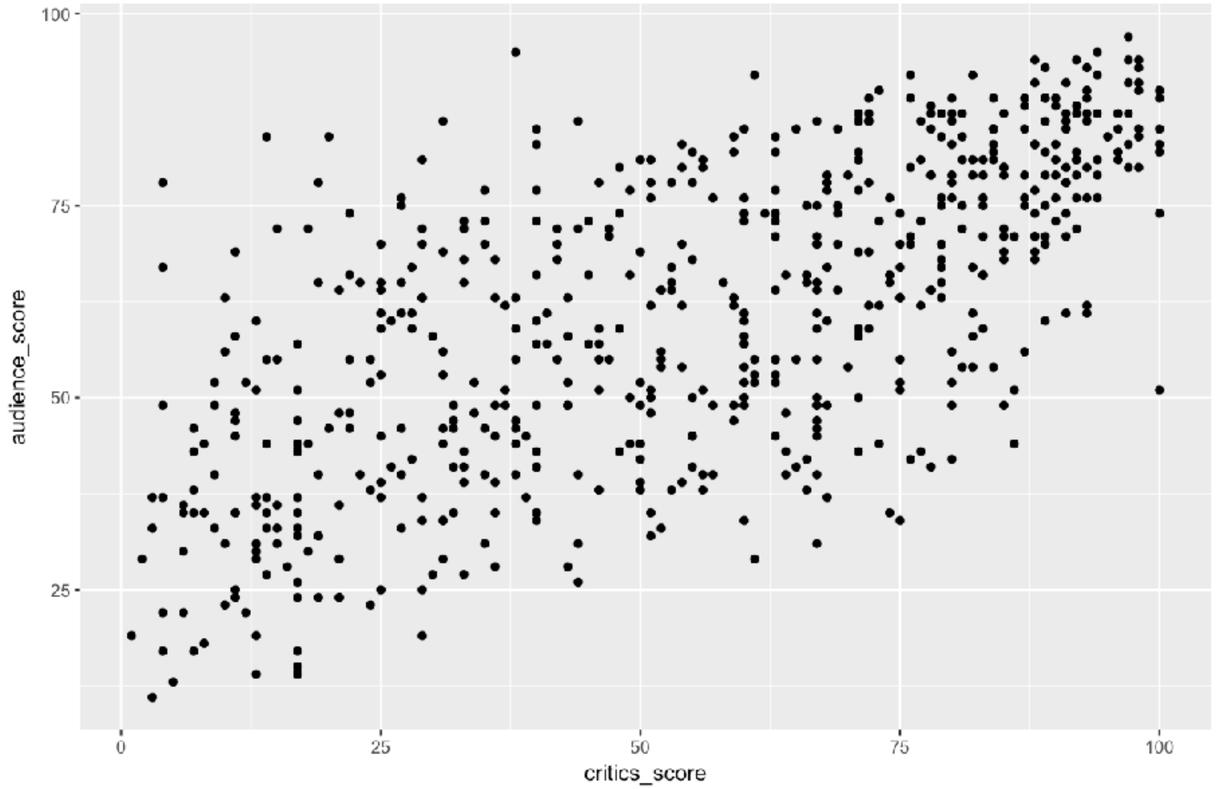


Y-axis:

audience_score

X-axis:

critics_score



Y-axis:

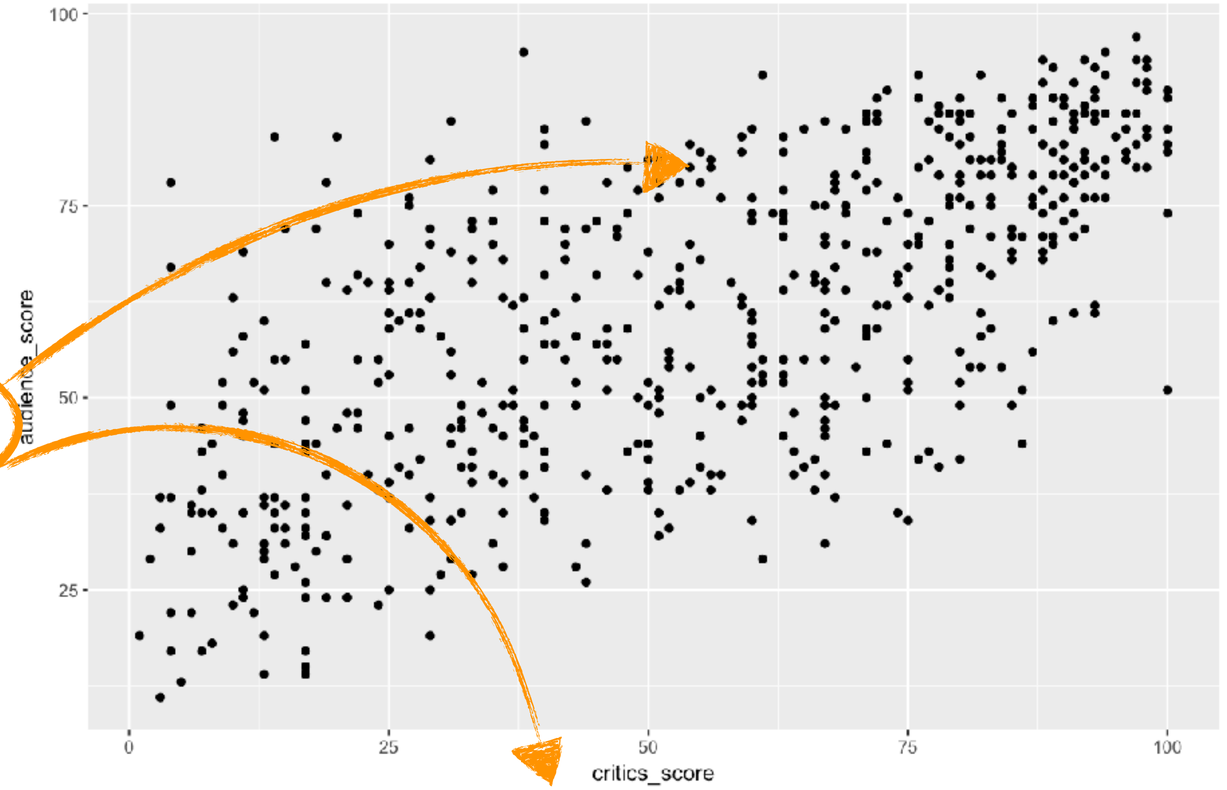
audience_score

X-axis:

critics_score

Select movie type:

Feature Film



The plot displays the relationship between the audience and critics' scores of 591 **Feature Film** movies.

1. **ui:** Add a UI element for the user to select which type(s) of movies they want to plot.

```
# Select which types of movies to plot
selectInput(inputId = "selected_type",
            label = "Select movie type:",
            choices = levels(movies$title_type),
            selected = "Feature Film")
```

2. **server:** Filter for chosen title type and save the new data frame as a reactive expression.

```
# Create a subset of data filtering
movies_subset <- reactive({
  req(input$selected_type)
  filter(movies, title_type %in% in
})
```

Creates a **cached expression** that knows it is out of date when input changes

3. **server:** Use `movies_subset` (which is reactive) for plotting.

```
# Create scatterplot
output$scatterplot <- renderPlot({
    ggplot(data = movies_subset(),
           aes_string(x = input$x, y = input
      geom_point()
})
```

Cached - only re-run when inputs change

3. **ui** & **server:** Use `movies_subset` (which is reactive) for printing number of observations.

```
# ui - Lay out where text should appear on app
mainPanel(
  …
  # Print number of obs plotted
  uiOutput(outputId = "n"),
  …
  )
```

```
# server - Print number of movies plotted
output$n <- renderUI({
  HTML(paste0("The plot displays the relationship between the <br>
              audience and critics' scores of <br>",
              nrow(movies_subset()),
              " <b>", input$selected_type, "</b> movies."))
})
```

BUILDING WEB APPLICATIONS IN R WITH SHINY

Let's practice!

BUILDING WEB APPLICATIONS IN R WITH SHINY

Using reactives

Why use reactives?

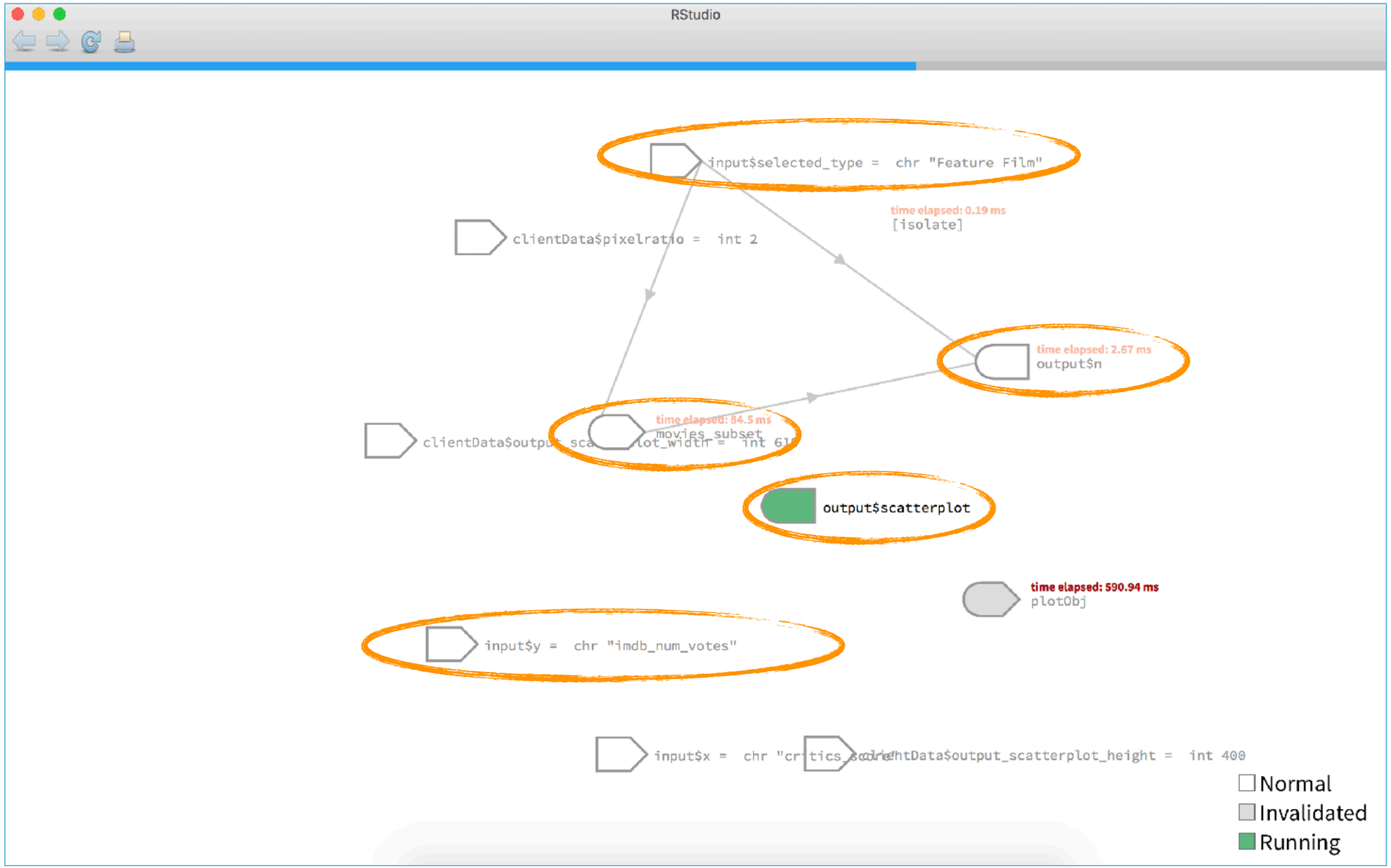
- By using a reactive expression for the subsetted data frame, we were able to get away with subsetting once and then using the result twice
- In general, reactive conductors let you
  - not repeat yourself (i.e. avoid copy-and-paste code)
  - decompose large, complex calculations into smaller pieces to make them more understandable
- Benefits similar to decomposing a large complex R script into a series of small functions that build on each other

Functions vs. reactives

- Each time you call a function, R will evaluate it.
- Reactive expressions are lazy, 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.

Reactlog

- Using many reactive expressions in your app can create a complicated dependency structure in your app.
- The **reactlog** is a graphical representation of this dependency structure, and it also gives you you very detailed information about what's happening under the hood as Shiny evaluates your application
- To view:
  - In a fresh R session, run `options(shiny.reactlog = TRUE)`
  - Then, launch your app as you normally would
  - In the app, press Ctrl+F3


RStudio
input$selected_type = chr "Feature Film"
time elapsed: 0.19 ms
[isolate]
clientData$pixelratio = int 2
time elapsed: 2.67 ms
output$n
time elapsed: 84.5 ms
movies_subset
output$scatterplot
time elapsed: 590.94 ms
plotObj
input$y = chr "imdb_num_votes"
input$x = chr "critics_score"
clientData$output_scatterplot_height = int 400
Normal
Invalidated
Running

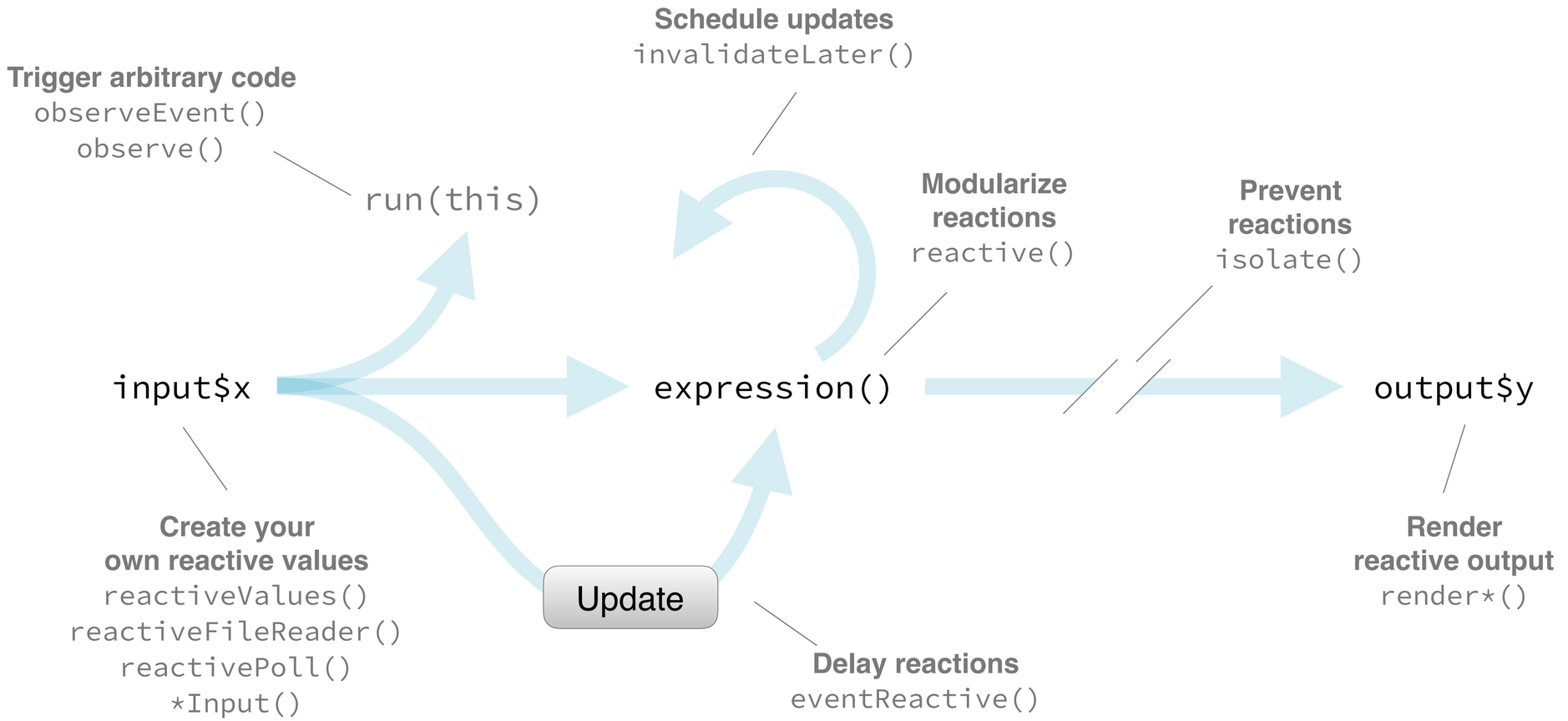
BUILDING WEB APPLICATIONS IN R WITH SHINY

Let's practice!

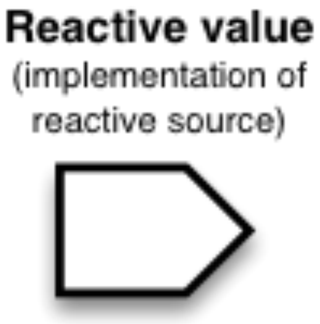
BUILDING WEB APPLICATIONS IN R WITH SHINY

Reactives and observers

Reactive flow



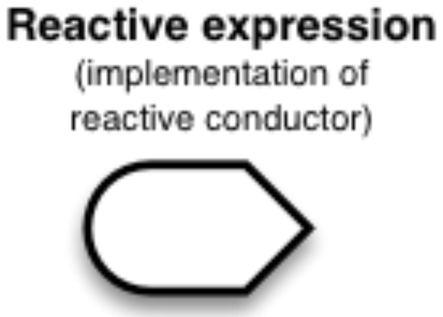
Implementation of reactive sources



`reactiveValues()`

e.g. `input$*:` Reactive value that looks like a list, and contains many individual reactive values that are set by input from the web browser.

Implementation of reactive conductors



`reactive()`

e.g. Reactive data frame subsets we created earlier.

- Can access reactive values or other reactive expressions, and they return a value
- Useful for caching the results of any procedure that happens in response to user input

Implementation of reactive endpoints

`observe()`

e.g. An `output$*` object is an observer. Actually what's going on under the hood is that a render function returns a reactive expression, and when you assign it to an `output$*` value, Shiny automatically creates an observer that uses the reactive expression.

- Can access reactive sources and reactive expressions, but they don't return a value
- They are used for their **side effects**, typically sending data to the web browser

Reactives vs. observers

- Similarities: Both store expressions that can be executed
- Differences:
  - Reactive expressions return values, but observers don't
  - Observers (and endpoints in general) eagerly respond to changes their dependencies, but reactive expressions (and conductors in general) do not
  - Reactive expressions must not have side effects, while observers are only useful for their side effects
- Most importantly:
  - `reactive()` is for calculating values, without side effects
  - `observe()` is for performing actions, with side effects
  - Do not use an `observe()` when calculating a value, and especially don't use `reactive()` for performing actions with side effects

Reactives vs. observers

	`reactive()`	`observer()`
Purpose	Calculations	Actions
Side effects	Forbidden	Allowed

BUILDING WEB APPLICATIONS IN R WITH SHINY

Let's practice!

BUILDING WEB APPLICATIONS IN R WITH SHINY

Stop-trigger-delay

Isolating reactions

Goal: Update plot (and title) when inputs other than `input$plot_title` changes.

```
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 other **`input`**s in this chunk change

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

Triggering reactions

expression to call whenever `eventExpr` is invalidated

```
observeEvent(eventExpr, handlerExpr, …)
```

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

Triggering reactions

Goal: Write a CSV of the sampled data when action button is pressed.

```
# ui
actionButton(inputId = "write_csv", label = "Write CSV")
```

```
# server
observeEvent(input$write_csv, {
             filename <- paste0("movies_",
                                str_replace_all(Sys.time(), ":|\ ", “_"),
                                ".csv")
             write_csv(movies_sample(), path = filename)
             }
)
```

Delaying reactions

expression to call whenever `eventExpr` is invalidated

```
eventReactive(eventExpr, handlerExpr, …)
```

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

Delaying reactions

Goal: 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”.

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

```
# server
movies_sample <- eventReactive(input$get_new_sample,{
                                  req(input$n_samp)
                                  sample_n(movies_subset(), input$n_samp)
                                  },
                                  ignoreNULL = FALSE
)
```

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

observeEvent vs. eventReactive

- observeEvent() is to to perform an action in response to an event
- eventReactive() is used to create a calculated value that only updates in response to an event

observeEvent/eventReactive vs. observe/reactive

- observe() and reactive() functions automatically trigger on whatever they access
- observeEvent() and eventReactive() functions need to be explicitly told what triggers them

isolate vs. event handling functions

- isolate() is used to stop a reaction
- observeEvent() is used to perform an action in response to an event
- eventReactive() is used to create a *calculated value* that only updates in response to an event

BUILDING WEB APPLICATIONS IN R WITH SHINY

Let's practice!

BUILDING WEB APPLICATIONS IN R WITH SHINY

Reactivity recap

Three lessons

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.

2. Reactives are for reactive values and expressions, observers are for their side effects.

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

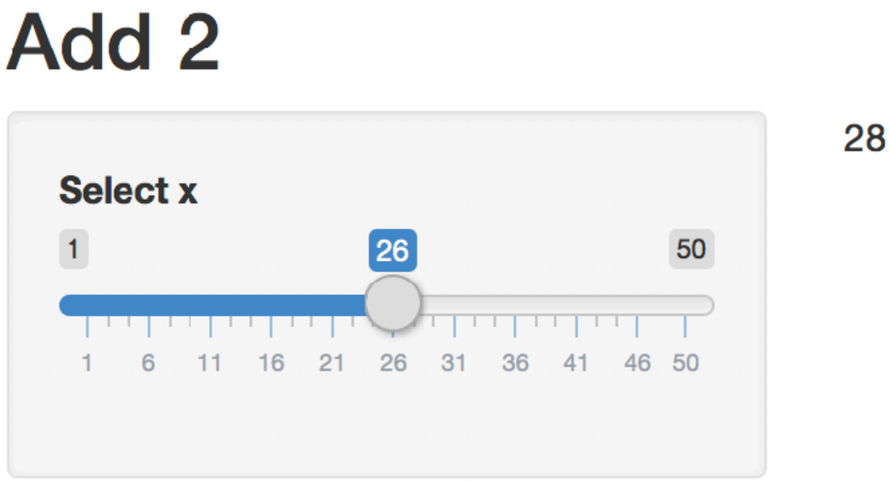
What's wrong?

```
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 })
}

shinyApp(ui, server)
```

What's wrong?



What's wrong?

```
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 })
}

shinyApp(ui, server)
```

What's wrong?

```
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 })
}
```

What's wrong?

```
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() })
}
```

BUILDING WEB APPLICATIONS IN R WITH SHINY

Let's practice!