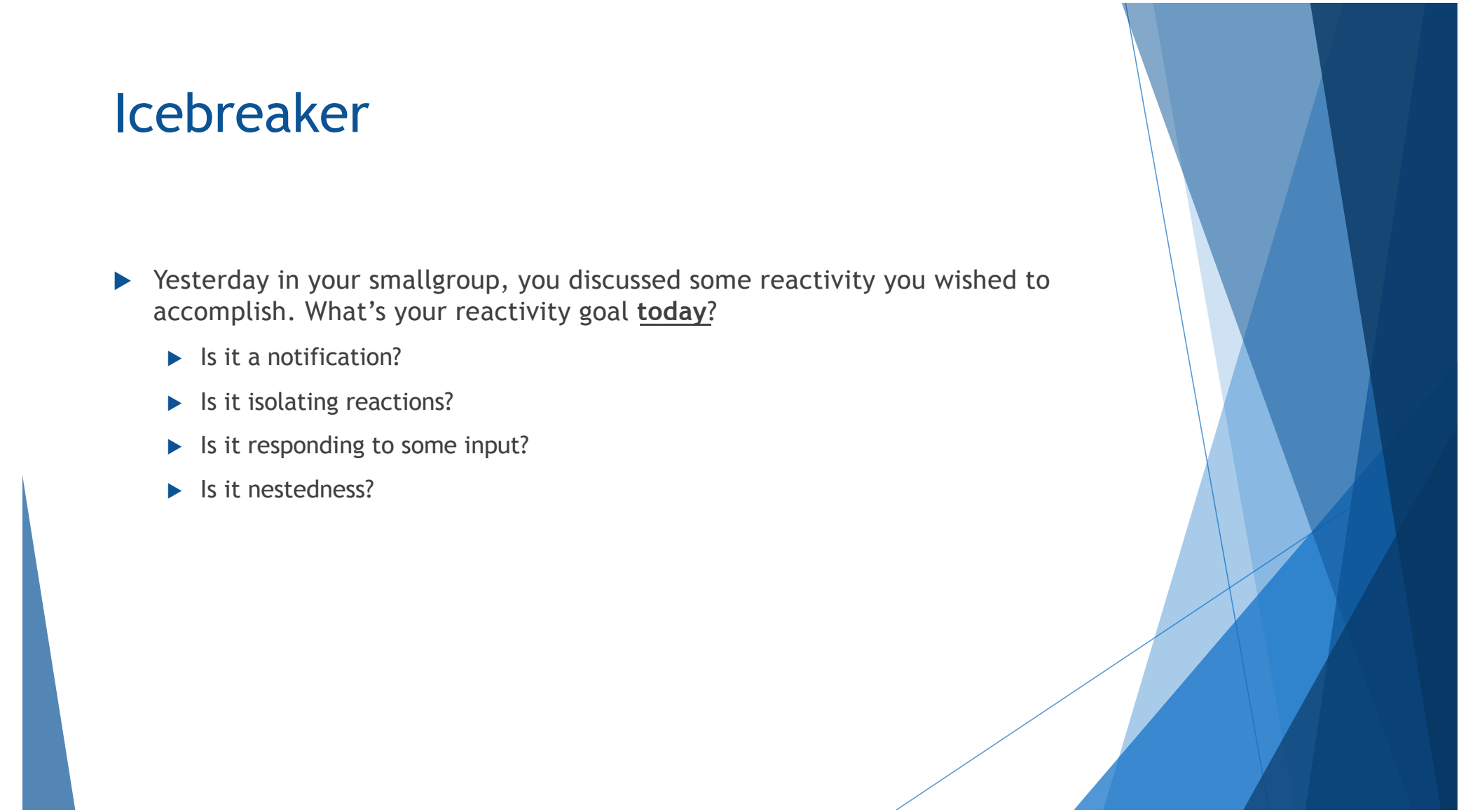
Building R-Shiny Applications

FAS 6932 – Special topics

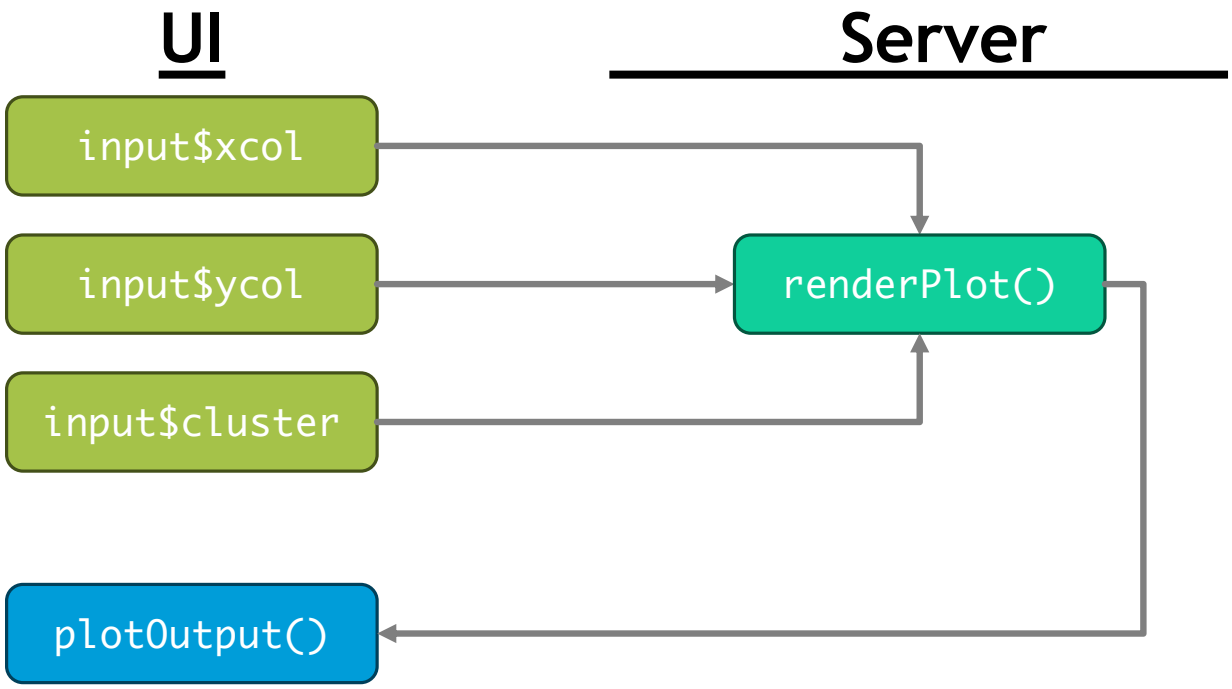
Summer C 2025

Icebreaker

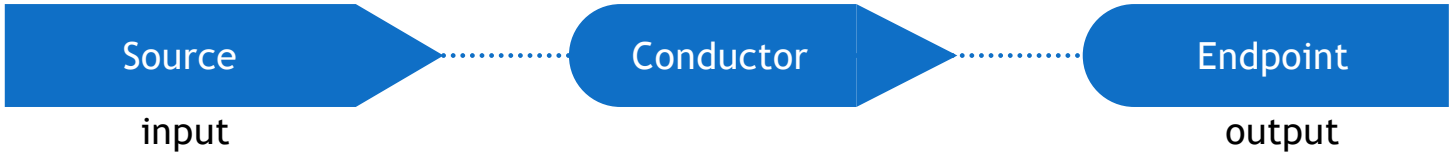
- Yesterday in your smallgroup, you discussed some reactivity you wished to accomplish. What's your reactivity goal **today**?
  - Is it a notification?
  - Is it isolating reactions?
  - Is it responding to some input?
  - Is it nestedness?



Basic Reactivity

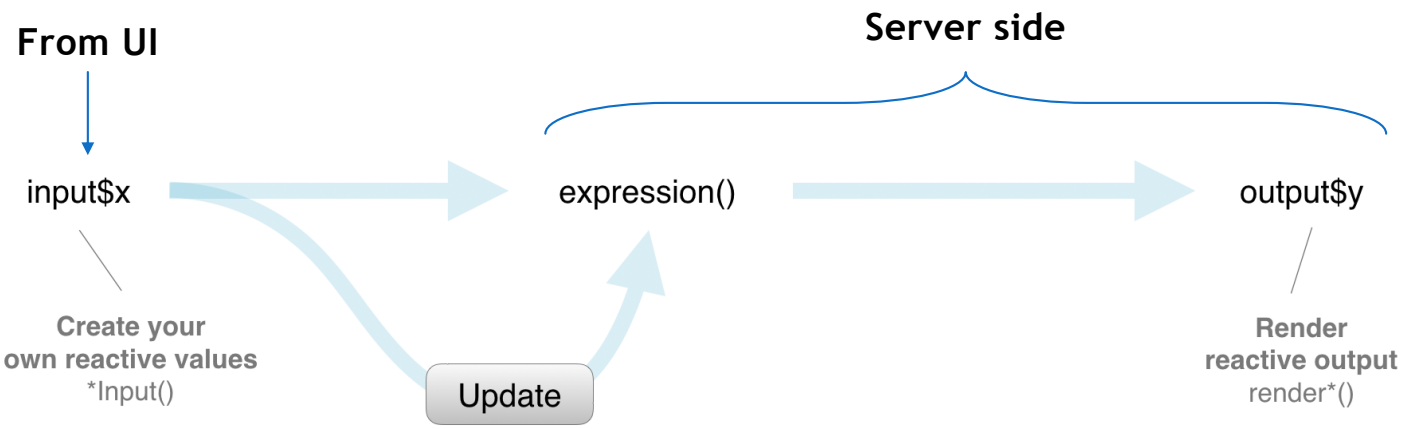


Reactive programming



- For most things, we do not use conductors
- But, conductors are useful for isolating large calculations
  - Simulating data
  - Filtering the data a lot (especially involving matches)
  - Spatial calculations
  - Literally calculation you do more than once in any render call

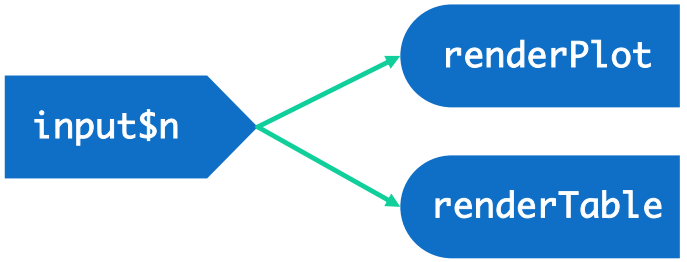
Source to endpoint



Conductor example

```
sim <- function(x, ...){}

server <- function(input, output, session){
    output$plot <- renderPlot({
        sim1 <- sim(input$n, ...)
        plot(sim1$x, sim1$y, pch = 16)
    })
    output$tab <- renderTable({
        table(sim(input$n,...))
    })
}
```

Notice, we call the **`sim`** function twice here, once in each render

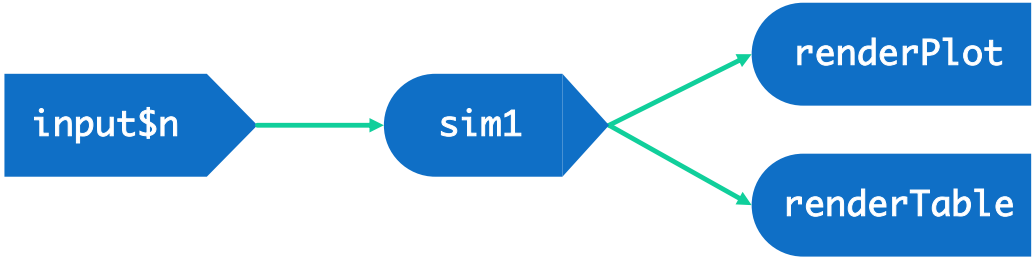


Conductor example

```
sim <- function(x, ...){}

server <- function(input, output, session){
    sim1 <- reactive({sim(input$n, ...)})
    output$plot <- renderPlot({
        plot(sim1()$x, sim1()$y, pch = 16)
    })
    output$tab <- renderTable({
        table(sim1())
    })
}
```

The `input$n` and sim function is now called within reactive making `sim1` a reactive function

Notice, now when we call `sim1`, we have to use `sim1()` because it is a reactive function



Conductor example

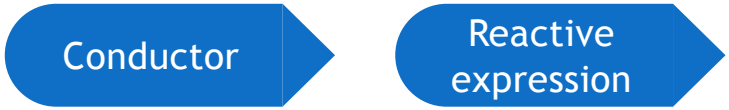
```
sim <- function(x, ...){}

server <- function(input, output, session){
    sim1 <- reactive({sim(input$n, ...)})
    ...
}

server <- function(input, output, session){
    sim1 <- sim(input$n, ...)
    ...
}
```

The `input$n` and sim function is now called within reactive making `sim1` a reactive function

This is not the same as `sim1` is just whatever is returned from the `sim` function and not reactive and will cause errors!

Reactive programming

- implemented in a list-like object
- contains many individual reactive values
- `input` is an example of this
- but we can make our own



- cache their return values in response to changes in reactive values
- Expensive calculations
- Accessing database
- Taxing filters
- Anything spatial

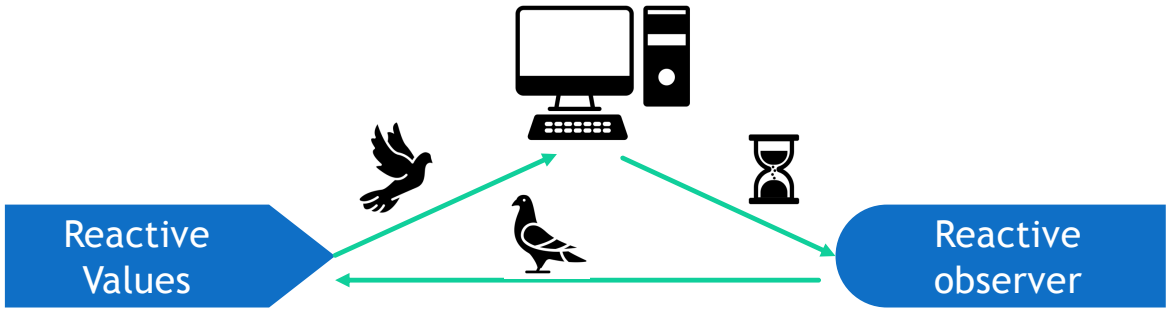
- do not cache return values but instead send along to web browser (UI)
- the `render` functions are not actual observers but when output$x <- happens they essentially make an observer

R-Shiny acts like carrier pigeons

Carrier pigeon tech

- Re-evaluating all the reactive expressions would bog a computer down
- This stalls the user experience and lags the reaction of rendered elements

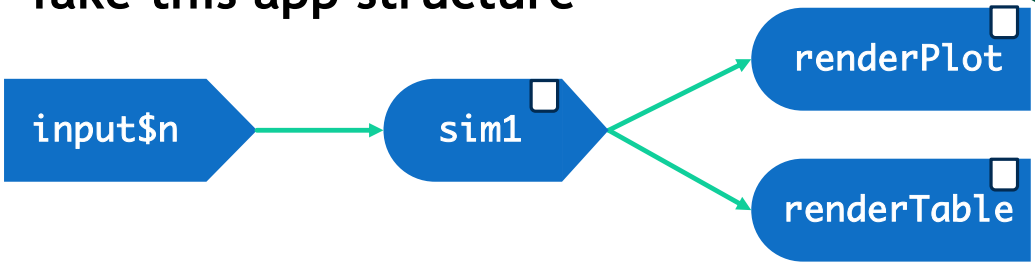
- R-Shiny gives a carrier pigeon to reactive values to send a notification to the server if they change
- this notice gets put into a queue at the server, waiting to re-evaluated
- when the notice makes it to the head of the line, the dependent reactive expressions get re-evaluated by checking the reactive value



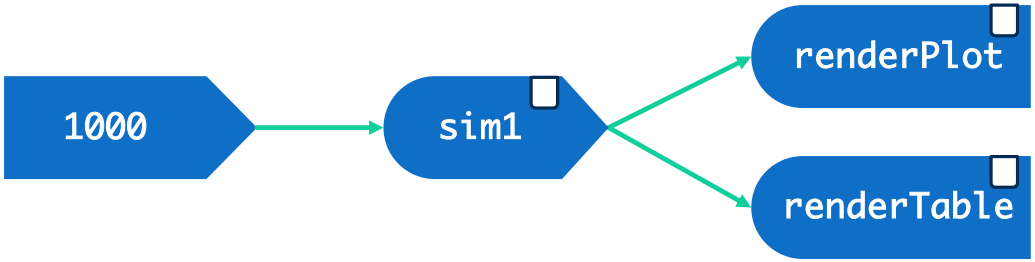
Invalidation and scheduling

Let's see it in action

Take this app structure



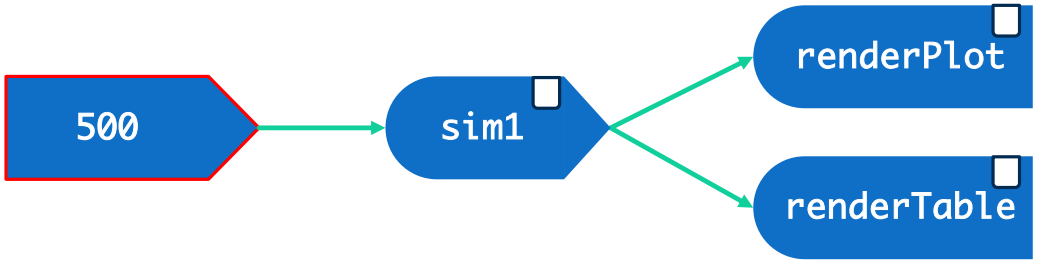
with an `input$n = 1000`



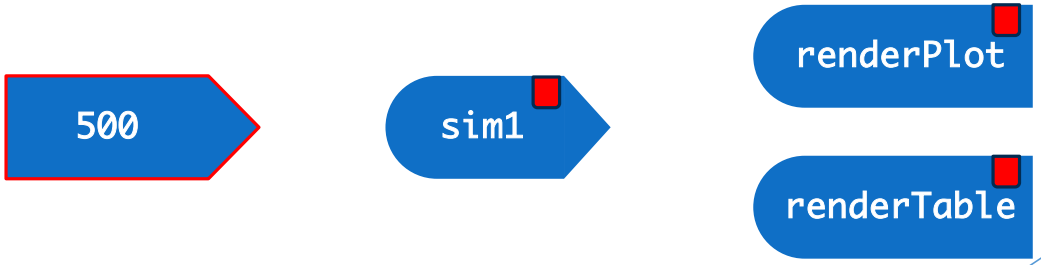
This ☐ designate as state of our reactive expressions and observers. When ☐ it means that the reaction is clean when ■ it means it is invalidated

Invalidation and scheduling

Now the input changes



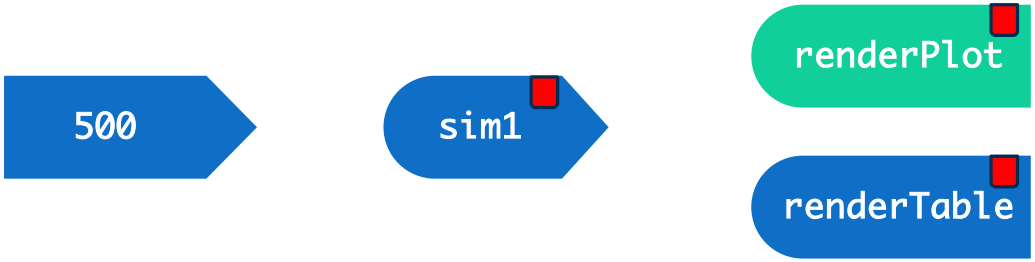
which invalidates all the children (reactive objects downstream)



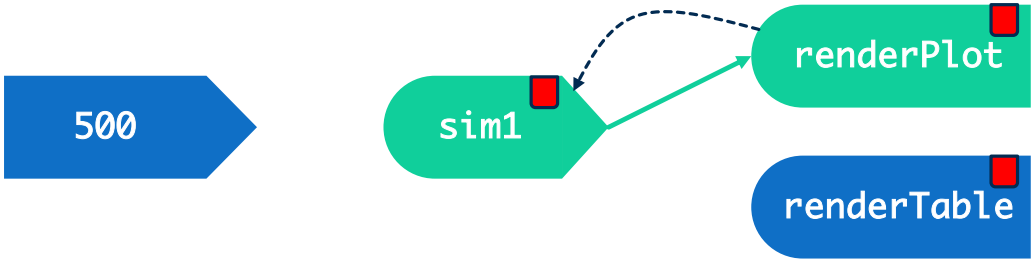
Once everything is invalidated, the reactive environment gets flushed and re-evaluation begins

Invalidation and scheduling

Now the re-evaluation begins starting at endpoints

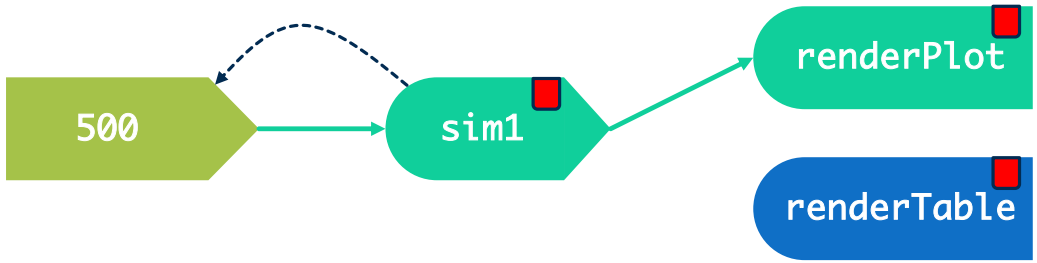


which calls its immediate parent, `sim1`

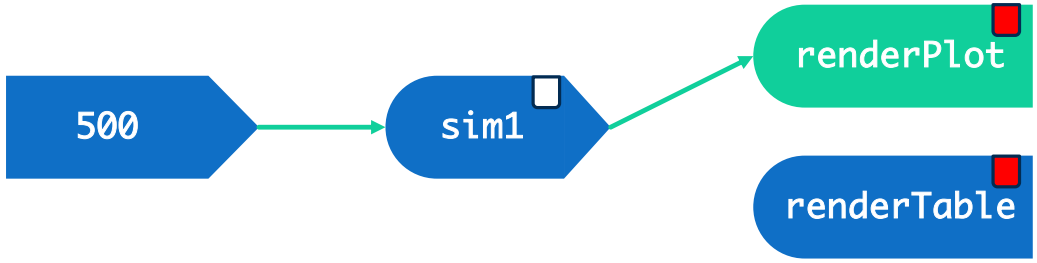


Invalidation and scheduling

this re-evaluation chains upwards to the next parent

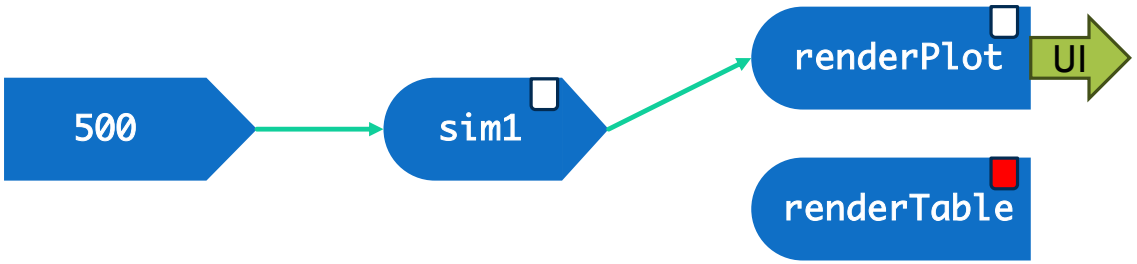


since `input$n` is a reactive source it chains back, `sim1` is evaluated and marked clean

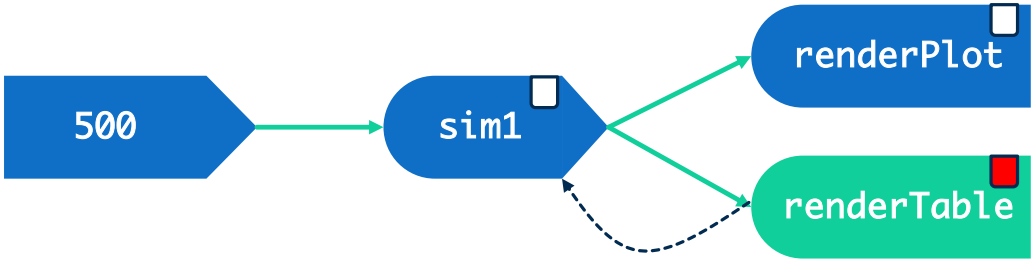


Invalidation and scheduling

this evaluation chains downwards to the next child, which marks `renderPlot` clean and output sends to UI

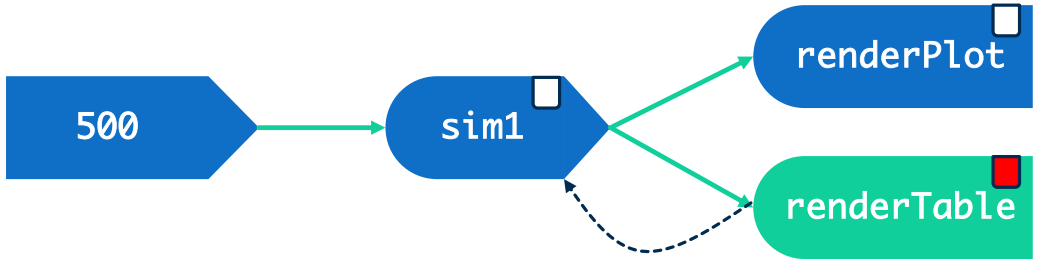


now `renderTable` can re-evaluate and checks its parent, `sim1`

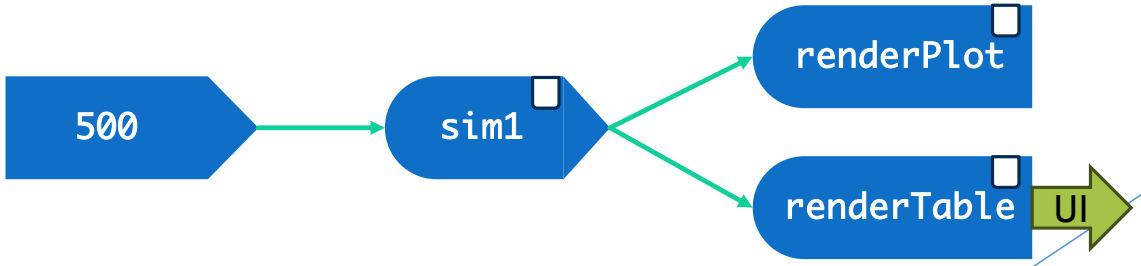


Invalidation and scheduling

since sim1 is marked clean, meaning it has re-evaluated since an input has change



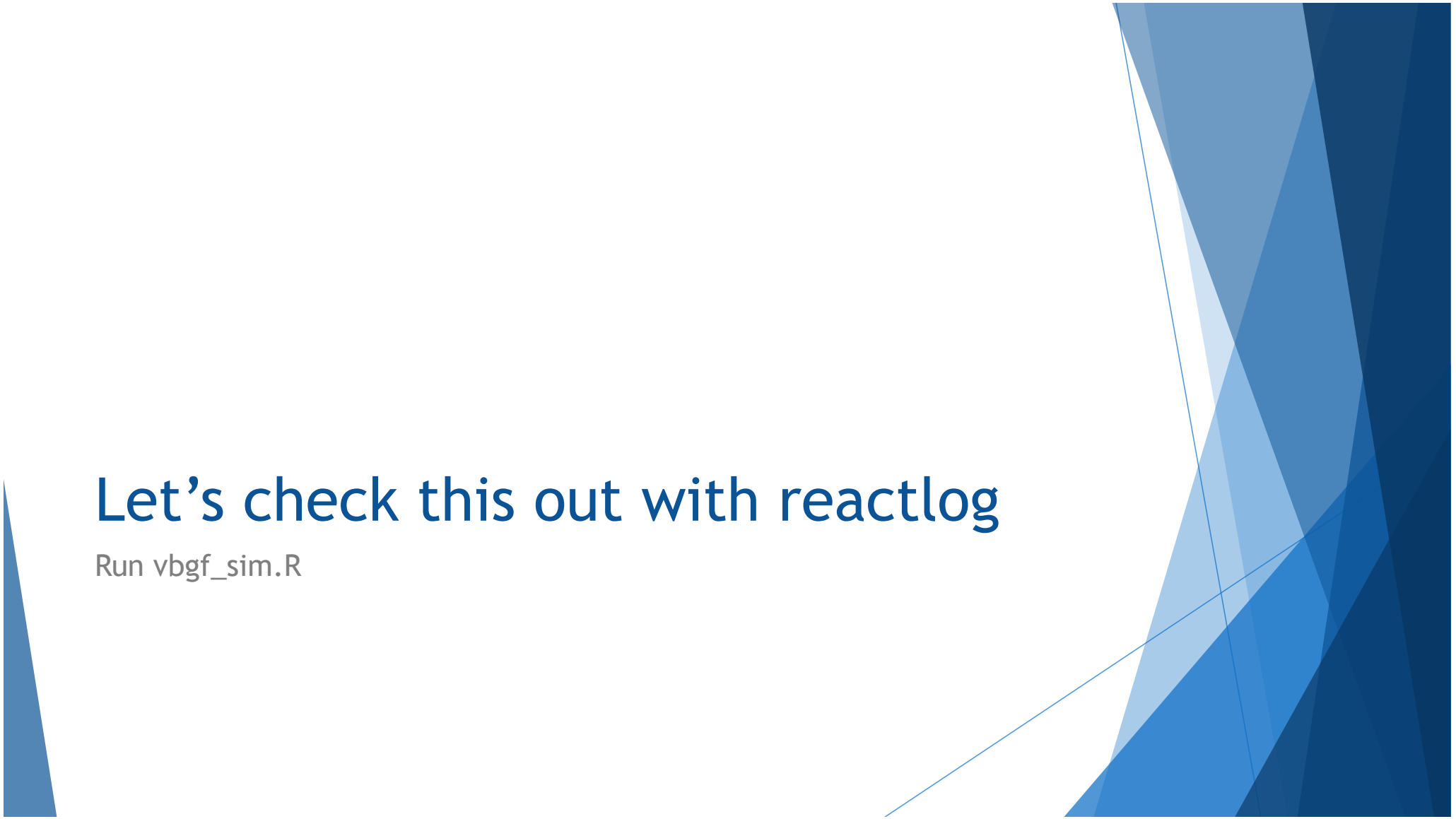
`renderTable` pulls the updated `sim1` values and can output to the UI and gets marked clean



10 minute break

Let’s check this out with reactlog

Run vbgf_sim.R



Get familiar with if{}else{} returns

- Often times you do not want to return anything until a button is clicked or until something is selected (with `selectInput` or `selectizeInput`)

```
output$distPlot <- renderPlot({
   if (input$goButton == 0)
     return()

   # plot-making code here
 })
```

- This is also the most basic of error handling you can do. If the user chooses some value or if your downstream reactive expressions break you can put in `if{}else{}` handlers

reactive({})

- We can use reactive({}) to make new Reactive Conductors
- reactive is lazy, it only executes when a parent changes and invalidates the reactive value
  - Any input$XXX that is called will be set up as a parent of the object declared by reactive({})
- These are great when you want to make a set of inputs that you want to listen to:

```
rickerListen <- reactive({
        list(input$simulate,input$ricker_a, input$ricker_b, input$ricker_sigma)
    })
```

- Use reactive({}) for calculations where you do not care about the intermediate products only the result
  - they act like R functions, they return the last result or whatever is in return()

reactiveValues

- We can also use reactiveVal() and reactiveValues() to make new Reactive Conductors
- reactiveVal is for a single reactive value
  - `r <- reactiveVal()`
- reactiveValues is for multiple
  - operates like a list, similar to how we called input
  - We have to declare an object as the container for the Reactive Values, then we can store things in it:
    - `values <- reactiveValues()`
    - `values$a <- 3 values[['b']] <- 4`
  - or
    - `values <- reactiveValues(a = 1, b = 2)`
- We can use reactiveValues to create a list where we can store lots of objects that depend on lots of other Reactive Values

observe and observeEvent

- observers are what they sound like, they observe
  - So they will wait for the pigeon notification then re-execute
  - But they do not have a return like reactive({});
  - They can have side effects
    - output is an observer that sends data to the web browser
- We can then react to a series of inputs similar to how renders with observe()
  - so any input within the expression is listened to
- Or, with something like reactiveValues made, we can listen to a single event with observeEvent()
  - This becomes useful for isolating reactions and not updating everything every time a single input changes:

```
observeEvent(rickerListen(),{…})
```

observer Example

```
server <- function(input, output, session){
    parms <- reactiveValues() #storage for reactive Values

    observeEvent(input$button,{
        #Mace & Doonan 1988 steepness from Goodyear CR
        parms$steepness <- input$CR/(4+input$CR)
        #Beverton-Holt SRR a parameter
        parms$BHa <- input$CR/input$EPRo
        #Beverton-Holt SRR b parameter
        parms$BHb <- (input$CR - 1)/(input$Ro * input$EPRo)
    })
    output$pars <- renderText({paste(parms,collapse=", ")})
    ...
}
```

Notice, we declare object into our reactiveValues list-like object the same way we do list

observer Example

```
ui <- fluidPage(
  textInput("name", "name"),
  actionButton("add", "add"),
  textOutput("names")
)
server <- function(input, output, session) {
  r <- reactiveValues(names = character())
  observeEvent(input$add, {
    r$names <- c(input$name, r$names)
    updateTextInput(session, "name", value = "")
  })
  output$names <- renderText(r$names)
}
```

Notice, we declared a Reactive Value container to store the names in

We are having our **observer** update the UI here. So once a user clicks ‘Add’, we then turn the text entry blank

We can then send those Reactive Values to the web browser

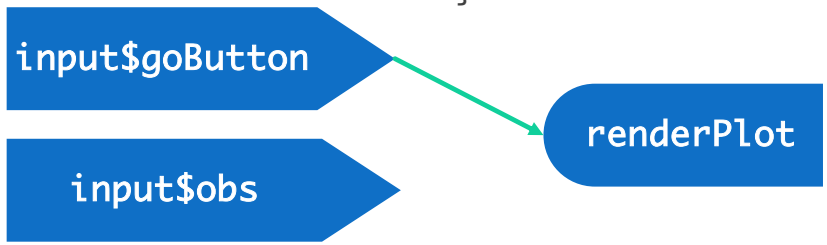
Isolating reactivity

```
ui <- pageWithSidebar(
  headerPanel("Click the button"),
  sidebarPanel(
    sliderInput("obs", "Number of
observations:", min = 0, max =
1000, value = 500),
    actionButton("goButton", "Go!")
  ),
  mainPanel(
    plotOutput("distPlot")
  )
)
```

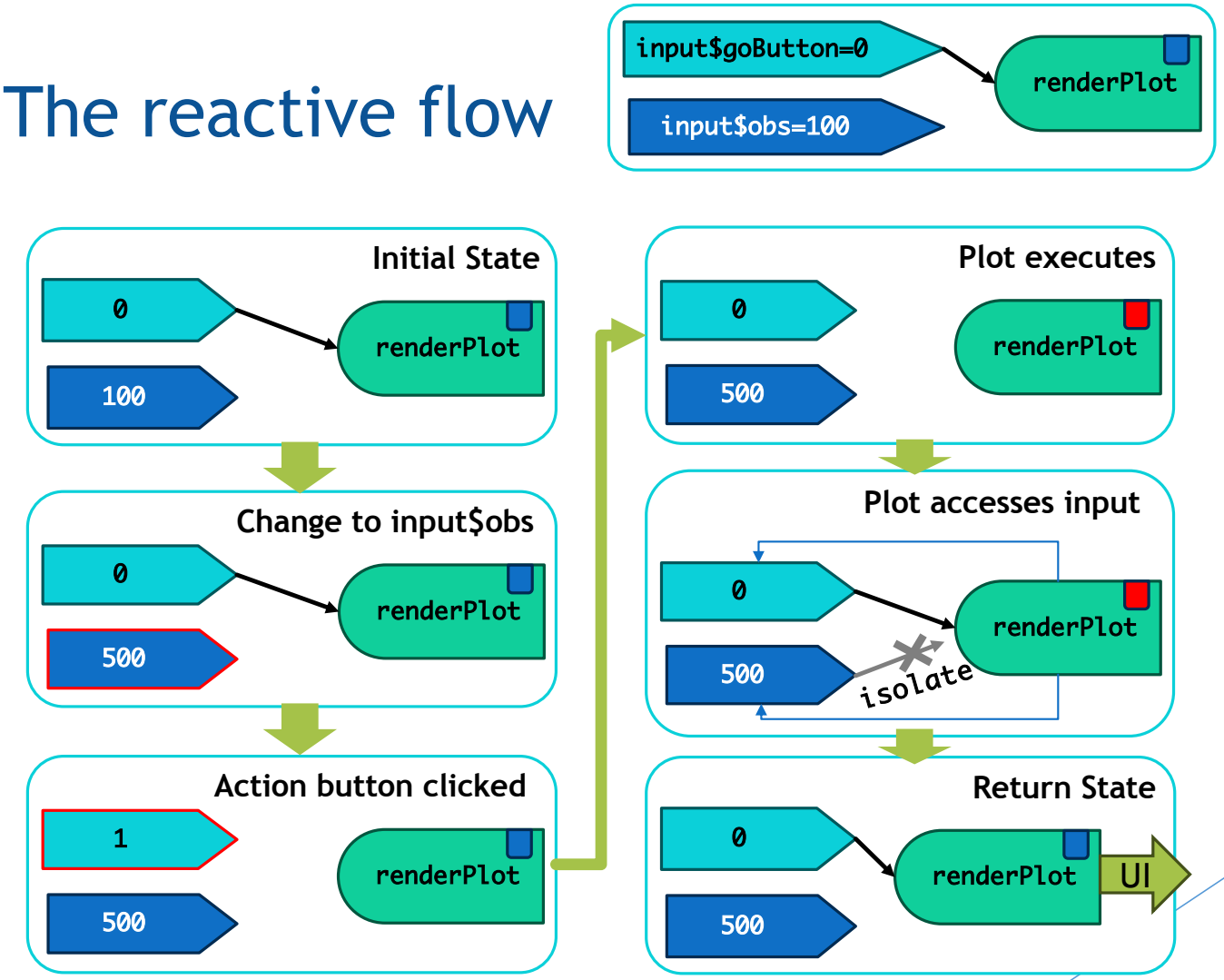
```
server <- function(input, output) {
  output$distPlot <- renderPlot({

    # Take a dependency on input$goButton
    input$goButton

    # Use isolate() to avoid dependency on input$obs
    dist <- isolate(rnorm(input$obs))
    hist(dist)
  })
}
```

Anytime that we put a reactive value (input$goButton), we introduce a dependency

the isolate function allows us to turn off that dependency



The reactive flow



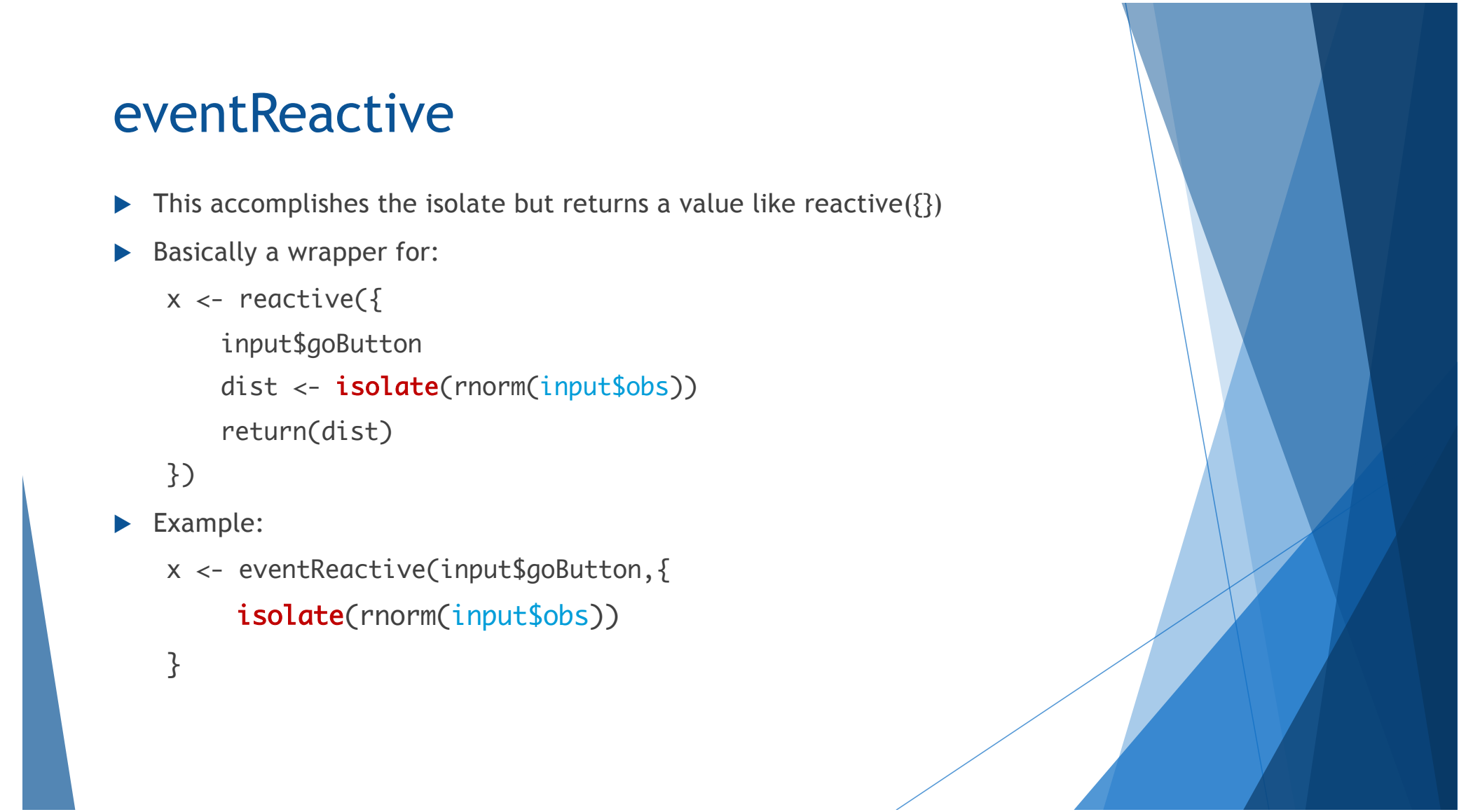
eventReactive

- This accomplishes the isolate but returns a value like reactive({})
- Basically a wrapper for:

```
x <- reactive({
    input$goButton
    dist <- isolate(rnorm(input$obs))
    return(dist)
})
```

- Example:

```
x <- eventReactive(input$goButton,{
     isolate(rnorm(input$obs))
}
```

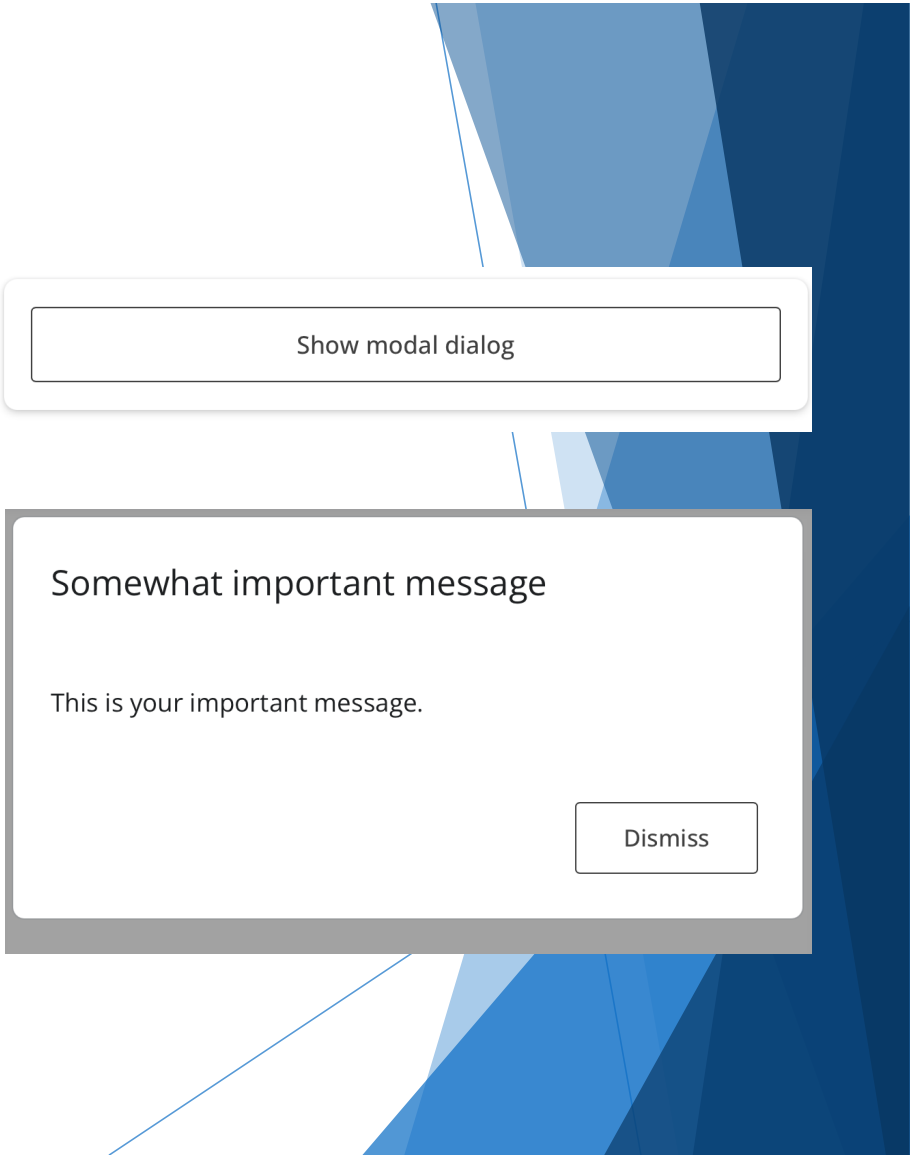


Recap

- Containers for Reactive Values:
  - `reactiveVal` or `reactiveValues`
- Reactive Expressions:
  - `reactive()` does calculations and returns ONE output
  - `eventReactive()` does calculations and returns ONE output but wraps `isolate()`
  - Observers, like `observe()` and `observeEvent()`, do actions (often declarations or web browser updates) and do not return anything!
    - `observe` ≈ `reactive`
    - `observeEvent` ≈ `eventReactive`
- `isolate`, `observeEvent`, and `eventReactive` are ways to selectively listen to only specific Reactive Values

Notifications

```
library(shiny)
library(bslib)

ui <- page_fixed(
     fixedRow(card()),
  card(actionButton("show", "Show modal dialog"))
)

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

  observeEvent(input$show,{
    showModal(
      modalDialog(
        title = "Somewhat important message",
        easy_close = TRUE,
        "This is your important message."
      )
    )
  })
}

shinyApp(ui, server)
```



Tooltips & Popovers

- Tooltips show on hover
- Popovers pop on click
- Can add to an HTML tagged object:

```
card_header(
    tooltip(
      span("Card 1 ", bsicons::bs_icon("question-circle-fill")),
      "This is the iris dataset",
      placement = "right"
    )
```

- Or can wrap around a xxxxOutput object:

```
tooltip(
        plotOutput('plot3',
                   hover = hoverOpts(id = 'plot3_hover')),
        'This is a plot of the iris dataset',
        placement = 'right')
      )
```

Let’s check out some notifications

Run notify_walkthrough.R