ADVANCED REACTIVITY

Shiny from RStudio™

‣ Reactivity catalog

‣ Reactivity review

‣ Checking preconditions

‣ Time as a reactive source

‣ Limiting rate

Reactivity
catalog

REACTIVITY CATALOG

‣ Store values: **reactiveValues** / **input** / **makeReactiveBinding**

‣ Calculate values: **reactive** / **eventReactive**

‣ Execute tasks: **observe** / **observeEvent**

‣ Preventing reactivity: **isolate**

‣ Checking preconditions: **req**

‣ Time (as a reactive source): **invalidateLater** / ~~**reactiveTimer**~~ *(invalidateLater is a safer and simpler alternative)*

‣ Rate-limiting: **debounce** / **throttle**

‣ Live data: **reactiveFileReader** / **reactivePoll**

(Pretty sure this is just the beginning…)

REACTIVITY CATALOG

‣ Store values: `reactiveValues` / `input` / `makeReactiveBinding`

‣ Calculate values: `reactive` / `eventReactive`

‣ Execute tasks: `observe` / `observeEvent`

‣ Preventing reactivity: `isolate`

‣ Checking preconditions: `req`

‣ Time (as a reactive source): `invalidateLater` / `reactiveTimer` *(invalidateLater is a safer and simpler alternative)*

‣ Rate-limiting: `debounce` / `throttle`

‣ Live data: `reactiveFileReader` / `reactivePoll`

(Pretty sure this is just the beginning…)

Highlighted functions are fundamental, all others are built on top.

Reactivity review

REVIEW: REACTIVE EXPRESSIONS

▸ Use to calculate new values based on reactive values and other reactive expressions.

▸ Caches its return value, until notified of reactive dependencies being out-of-date.

▸ Lazily executes — Shiny wants to avoid running these whenever possible. For this reason, meaningful side effects are prohibited from reactive expressions.

▸ Call it like a function when you want to read its value.

REVIEW: REACTIVE EXPRESSIONS

```
# Declare
movies_subset <- reactive({
  movies %>% filter(title_type %in% input$type)
})

# Read
output$scatterplot <- renderPlot({
  ggplot(movies_subset(), aes(...)) + geom_point()
})
```

▸ Use to execute actions based on changing reactive values and other reactive expressions.

▸ Doesn't return a value. So performing side effects is usually the only reason you'd want to create one of these.

▸ Eagerly executed by Shiny.

```
observe({
  print(paste("The value of x is", input$x))
})

## [1] The value of x is 10
## [1] The value of x is 16
## [1] The value of x is 9
```

Shiny from RStudio™

REACTIVE EXPRESSIONS VS. OBSERVERS

reactive()	observer()
Callable	Not callable
Returns a value	No return value
Lazy	Eager
Cached	N/A
No side effects	Only for side effects

REACTIVE EXPRESSIONS VS. OBSERVERS VS. FUNCTIONS

reactive()	observer()	function()
Callable	Not callable	Callable
Returns a value	No return value	Returns a value
Lazy	Eager	Lazy
Cached	N/A	Not cached
No side effects	Only for side effects	Side effects optional

OBSERVEEVENT VS. EVENTREACTIVE

▸ Every reactive expression or reactive value read by a **reactive()** or **observe()** block automatically becomes a reactive dependency of that reactive expression/observer.

▸ **observeEvent** and **eventReactive** give us finer control.

```
observeEvent(input$save_button, {
  write.csv(movies_subset(), "movies.csv")
})
```

"When the save_button button is clicked, write the value of `movie_subset` **to disk."** (Don't write to disk automatically when `movie_subset` changes.)

OBSERVEEVENT AND EVENTREACTIVE

▸ **observeEvent** is for event handling

▸ **eventReactive** is for delayed computation

```
observeEvent(when_this_changes, {
  do_this
})


r <- eventReactive(when_this_changes, {
  recalculate_this
})
```

Use these functions when you want to **explicitly name your reactive dependencies**, as opposed to letting `reactive`/`observe` implicitly depend on anything they read.
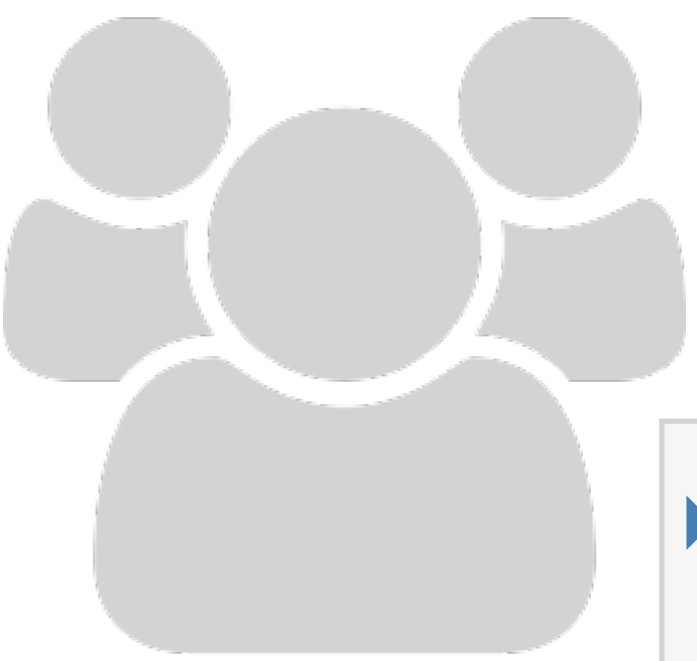
▸ Open **apps/adv-reactivity/cranlogs.R** and run it. This app has several problems:

  ▸ We get an error right off the bat — the plot is running before the user has specified any packages.

  ▸ Unless you're a very fast typist, typing package names will cause the cranlogs server to be queried with many incomplete queries.

  ▸ Add an "Update" actionButton to the UI, and make sure nothing happens until it's clicked.

5ₘ **00**ₛ

See **apps/adv-reactivity/cranlogs-solution.R**

```
packages <- reactive({
  strsplit(input$packages, " *, *")[[1]]
})
```
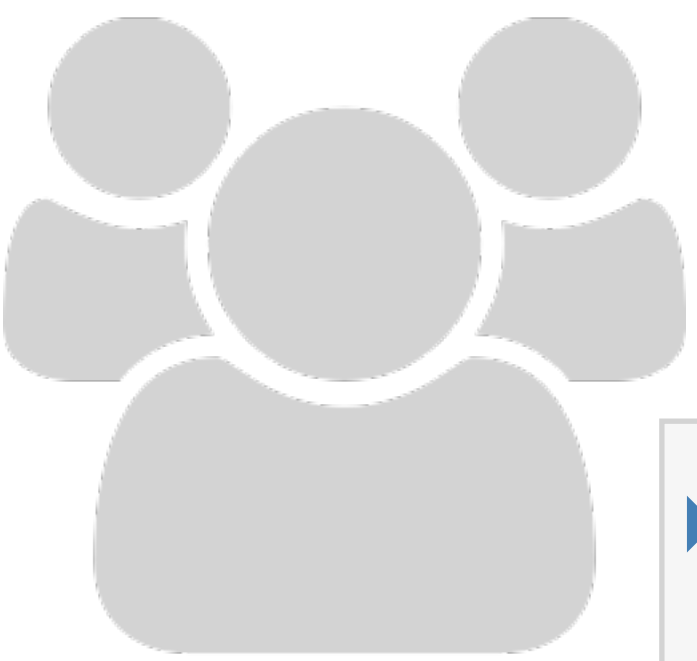
```
packages <- eventReactive(input$update, {
  strsplit(input$packages, " *, *")[[1]]
})
```

REVIEW: REACTIVE VALUES

‣ Read/write versions of input.

‣ Try not to use this to store *calculated* values. But in some cases, it's unavoidable.

```
# Create
rv <- reactiveValues(x = 10)

# Read
rv$x

# Write
rv$x <- 20
```

- Open the file **apps/adv-reactivity/counter.R**. It has three action buttons:

  - Increment: Increase the value by 1

  - Decrement: Decrease the value by 1

  - Reset: Set the value to 0

- Unfortunately, it doesn't work. See if you can implement the server side.

5m 00s

Shiny from RStudio™

See **apps/adv-reactivity/counter-solution.R**

```
rv <- reactiveValues(count = 0)

observeEvent(input$increment, {
  rv$count <- rv$count + 1
})
observeEvent(input$decrement, {
  rv$count <- rv$count - 1
})
observeEvent(input$reset, {
  rv$count <- 0
})

output$value <- renderText({
  rv$count
})
```

WHEN TO USE REACTIVEVALUES

‣ Don't use **`reactiveValues`** when you're calculating a value based on other values and calculations that are already available to you.

‣ Do use **`reactiveValues`** to store state that otherwise would be lost from your graph of reactive objects.

REACTIVEVALUES EXAMPLE 1

(1) A calculation over the history of something reactive:

```
observeEvent(input$add, {
  rv$total <- rv$total + input$x
})
```

(Or a more elegant way to do the same, using hadley/shinySignals:)

```
total <- shinySignals::reducePast(reactive(input$x), `+`, 0)
```

REACTIVEVALUES EXAMPLE 2

(2) Tracking which of several events happened most recently:

```
observeEvent(input$editMode, {
  rv$mode <- "edit"
})


observeEvent(input$previewMode, {
  rv$mode <- "preview"
})


output$page <- renderUI({
  if (rv$mode == "edit") {
    ...
  } else if (rv$mode == "preview") {
    ...
  }
})
```

REACTIVEVALUES EXAMPLE 3

(3) To change rules of reactivity:

‣ Normally, as soon as reactive expressions are invalidated (before they have recalculated) they invalidate everyone downstream who depends on them.

‣ But sometimes recalculating will end up giving us the same value as the previous anyway, and any downstream recalculations might have been wasted work.

```r
dedupeReactive <- function(rexpr, priority = 10) {
  rv <- reactiveValues(value = NULL)

  observe({
    rv$value <- rexpr()  # TODO: Handle errors
  }, priority = priority)

  reactive(rv$value)
}
```

PREVENTING REACTIVITY WITH ISOLATE

‣ Use `isolate` from inside a reactive expression or observer, to ignore the implicit reactivity of a piece of code.

‣ Wrap it around expressions or a whole code block.
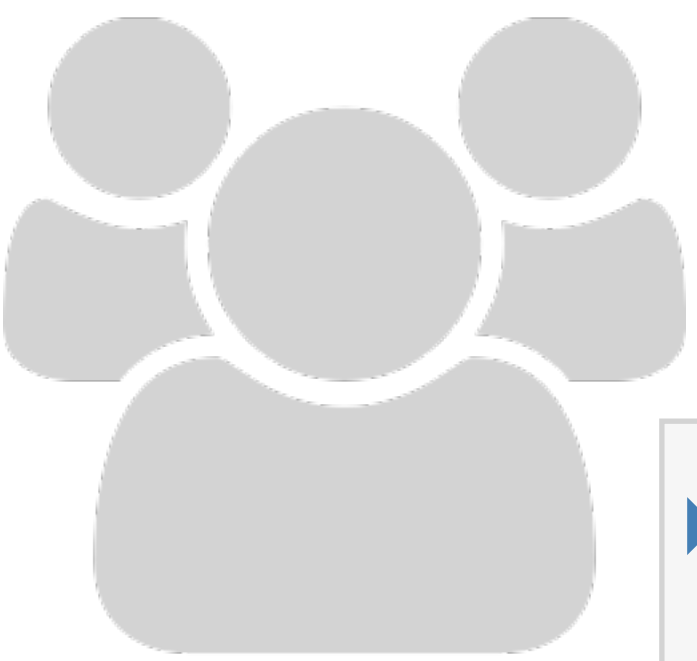
▶ Determine when r1, r2, and r3 update:

```
r1 <- reactive({
  input$x * input$y
})



r2 <- reactive({
  input$x * isolate({ input$y })
})



r3 <- reactive({
  isolate({ input$x * input$y })
})
```

```
# Updates every time input$x or input$y change
r1 <- reactive({
  input$x * input$y
})

# Updates only when input$x changes
r2 <- reactive({
  input$x * isolate({ input$y })
})

# Never updates; it will always have its original value
r3 <- reactive({
  isolate({ input$x * input$y })
})
```

Checking preconditions

CHECKING PRECONDITIONS WITH REQ

‣ Cancel the current output (or observer) if a condition isn't met.

  ‣ `req(input$text)`: Ensure the user has provided a value for the "text" input

  ‣ `req(input$button)`: Ensure the button has been pressed at least once

  ‣ `req(x %% 2 == 0)`: Ensure that x is an even number

  ‣ `req(FALSE)`: Unconditionally cancel the current reactive, observer, or output

CHECKING PRECONDITIONS WITH REQ

‣ `req(cond)` is similar to:

  ‣ `stopifnot(cond)`
  ‣ `if (!cond) stop()`
  ‣ `assertthat::assert_that(cond)`

‣ but with these differences:

  ‣ Errors during output rendering show up with bold red text in the UI; **req** just makes the output blank
  ‣ Rather than verifying that **cond** is *true*, **req** verifies that **cond** is *truthy* (see `?isTruthy`)
    ‣ Feels unnatural to be so arbitrary and nebulous, but this definition is just too practical for UI programming
  ‣ Most importantly, **req** is like an error in that it "infects" the downstream elements of the reactive graph (if a reactive throws an error, then any other reactive/observer/output that tries to access it will also throw an error)
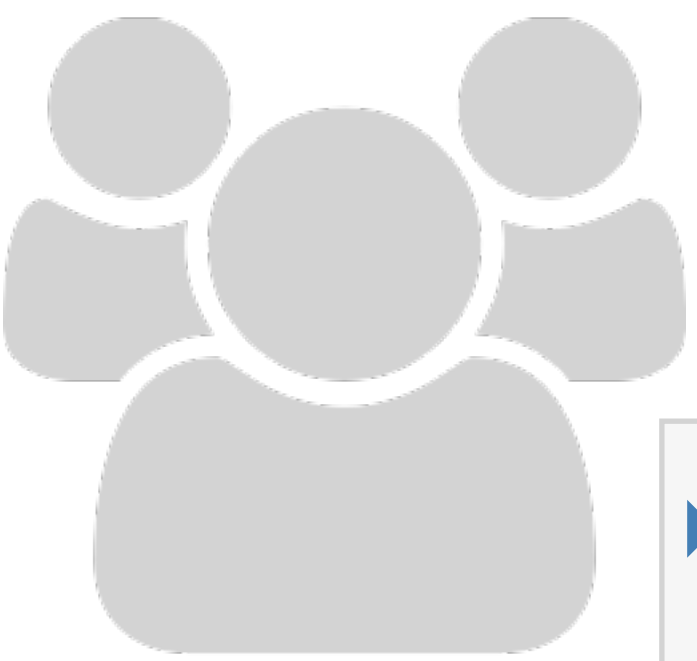
▸ Open `app/adv-reactivity/dynamic.R` and run it.

▸ It has lots of errors in the browser and the R console — ignore those for the moment.

▸ From the app, upload the `diamonds.csv` file found in the same directory. Now everything looks good.

▸ See if you can figure out why these errors appear when the app first comes up, and how you can get them to go away (first without `req`, and then, if you have time and can figure out how, using `req`).

5_m 00_s

▸ Antisolution: `apps/adv-reactivity/dynamic-antisolution.R`.

  ▸ This is how you used to have to do it: check for missing values yourself, and `return(NULL)`.

  ▸ You had to do this in every reactive, observer, or output that could have a missing value, plus all of the reactives, observers, and outputs that are downstream!

▸ Solution: `apps/reactivity/dynamic-solution.R`.

  ▸ Now you can use req in the reactives, observers, and outputs that directly use potentially-missing inputs, and everything downstream can just not worry about it.

Shiny from R Studio™
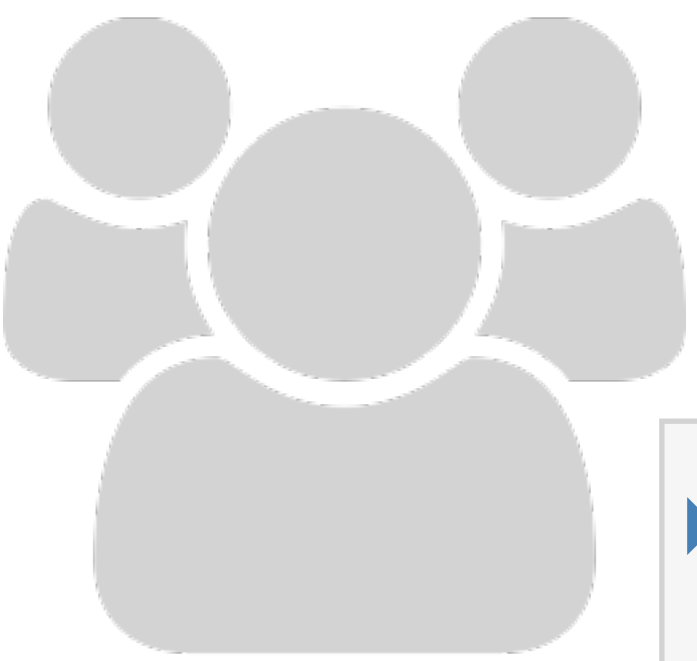
Time as a reactive source
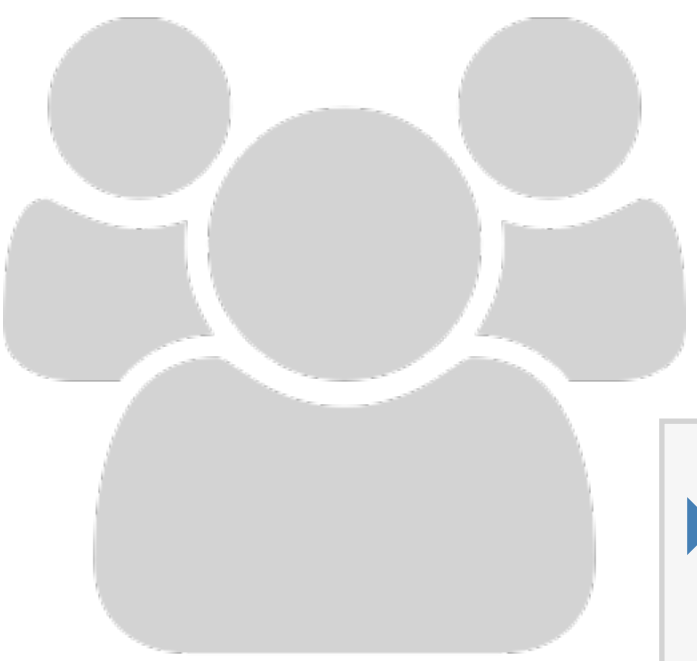
▸ What will this prouduce?

```
ui <- basicPage( verbatimTextOutput("text") )

server <- function(input, output){

  r <- reactive({ Sys.time() })
  output$text <- renderPrint({ r() })

}

shinyApp(ui, server)
```

An app that reports Sys.time() at the time of first launch, and then doesn't update it

▸ What will this prouduce?

```
ui <- basicPage( verbatimTextOutput("text") )

server <- function(input, output){

  r <- reactive({
    invalidateLater(1000)
    Sys.time()
  })
  output$text <- renderPrint({ r() })

}

shinyApp(ui, server)
```

An app updates reported
Sys.time() every second

Limiting rate

DEBOUNCE AND THROTTLE

▸ If a reactive value or expression changes too fast for downstream calculations to keep up, you can end up with a bad user experience (laggy experience, wasted work).

   ▸ **debounce** and **throttle** take a reactive expression object as input, and return a rate-limited version of that reactive expression.

```r
# A reactive that updates as often as every 50 milliseconds
fast_reactive <- reactive({ ... })

# A reactive that updates no more often than every 2000 milliseconds
throttled_reactive <- throttle(fast_reactive, 2000)

# A reactive that doesn't update until fast_reactive has stopped
# changing for at least 1000 milliseconds
debounced_reactive <- throttle(fast_reactive, 1000)
```
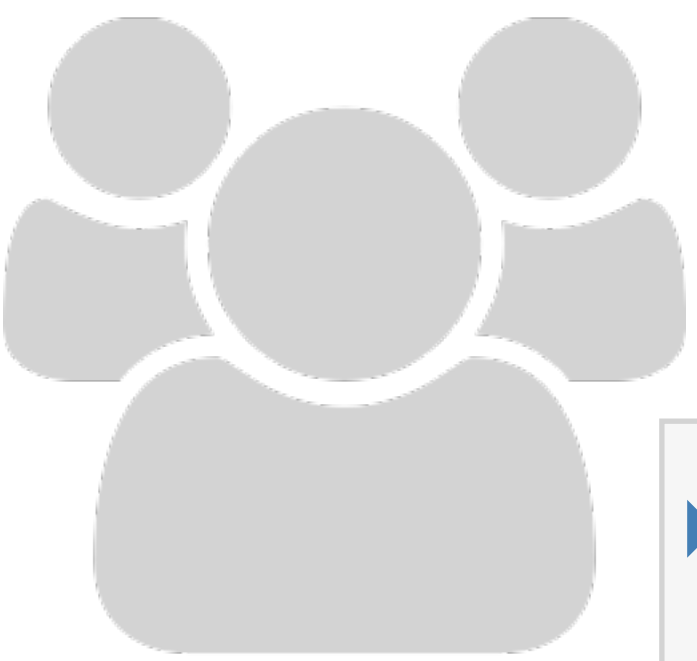
▸ Open and run **apps/adv-reactivity/points.R**. Click on the plot a few times to create points. Notice the annoying laggy behavior — this is due to a (simulated) expensive summary output.

▸ Use **debounce** or **throttle** to prevent the summary output from running so often.

5ₘ 00ₛ

See **apps/adv-reactivity/points-solution.R**

ADVANCED REACTIVITY

Shiny from RStudio