Part 2: Reactivity

Ted Laderas
Fred Hutch Cancer Center

This section is for:

- Those who never really understood reactivity
- Don’t know what the difference is between `observeEvent()` and `eventReactive()`
- Want to know what `ExtendedTask` does

Why Reactivity?

- The bad old days: UI polling
- Poke every 100 ms at a UI element
  - Have you changed yet?
  - Have you changed yet?
  - Have you?
- Event-driven programming
  - Event: Change in inputs or reactives
- Only update outputs when necessary (lazy)

Shiny works best when you give it control

- Don’t tell Shiny **when** and **how** to update, only **how** to update
- Give Shiny control to update as it sees fit
- Don’t force order of operations - causes problems
- You need to trust Shiny

Reactivity

- All calculations are dependent on the *reactive graph*
- Depending on how inputs, reactives, and outputs are connected
- Only recalculate for *visible outputs* that have changes in inputs
- The reactive graph invalidates on changes in inputs / reactives

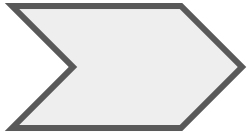
Key Players

`Input`

`input$select`
`reactiveValues()`

“Producers”

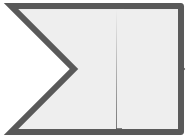
`reactive()`



`data <-`
`reactive({})`

“Consumers”/
“Producers”

`Output`



`output$plot`
`observe()`

“Consumers”

The Reactive Graph

- Calculates *visible outputs* (such as plots/tables) based on changes in inputs and reactives
- Initial calculation leads to an “equilibrium” state of graph
- Changes in inputs invalidate outputs and connected reactives
- Data flows from Inputs -> reactives() -> Outputs

`{reactlog}`

Lets you visualize the flow of information through the reactive graph

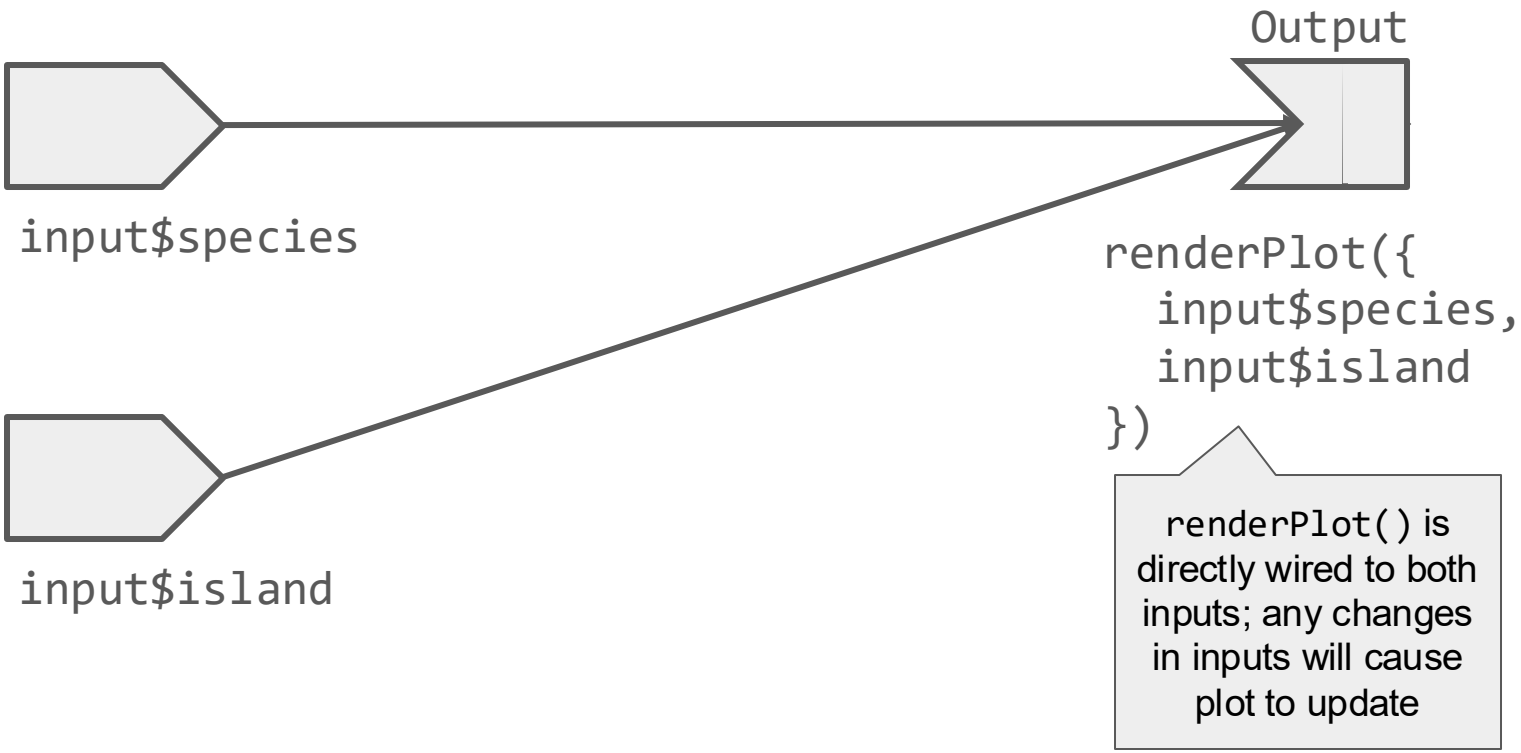
Invaluable for understanding how events trigger recalculation

Does not measure time of calculation

Use `{profvis}` for that

Direct Connections

`runApp("reactives/app_direct.R")`



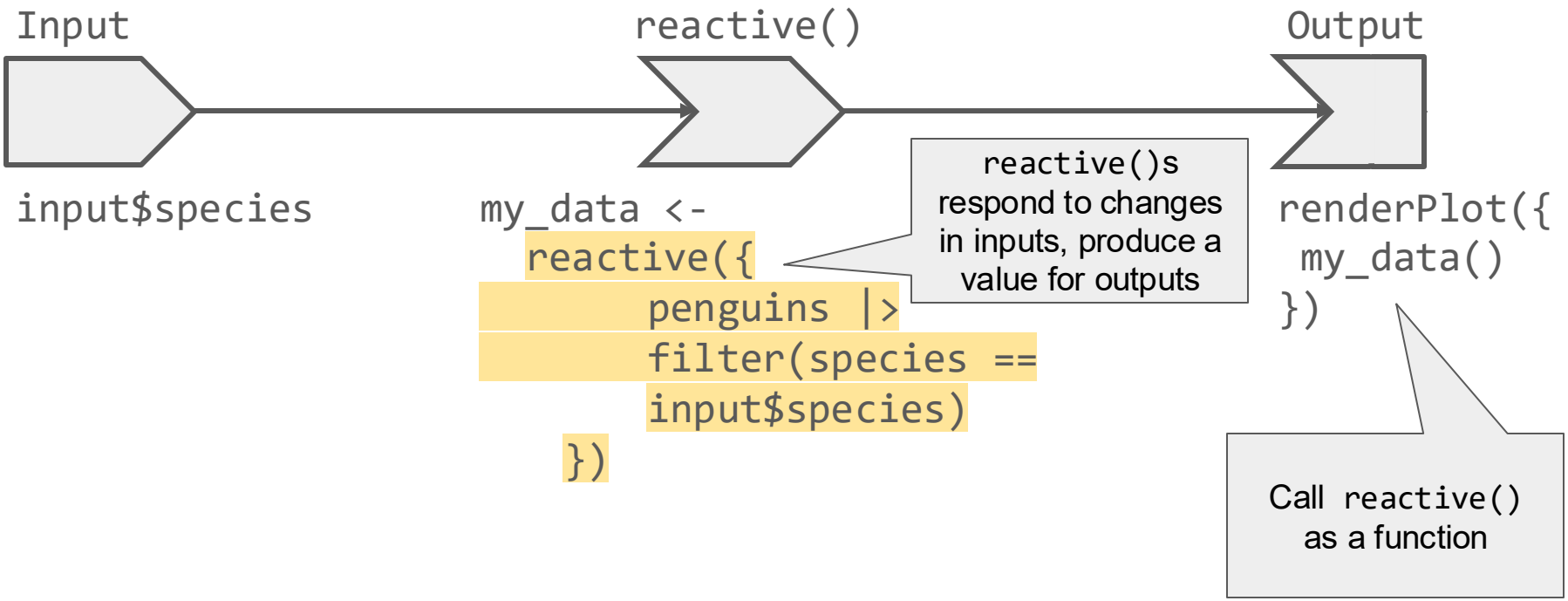
Quick reactlog demo
(hit Ctrl + F3 or Cmd + Fn + F3)

`reactive()`

- Short for *reactive expression*
- Dynamic (responds to other reactives or inputs)
- Returns a value (such as a data.frame)
- Decouples inputs from outputs
- Avoids recalculation unless necessary

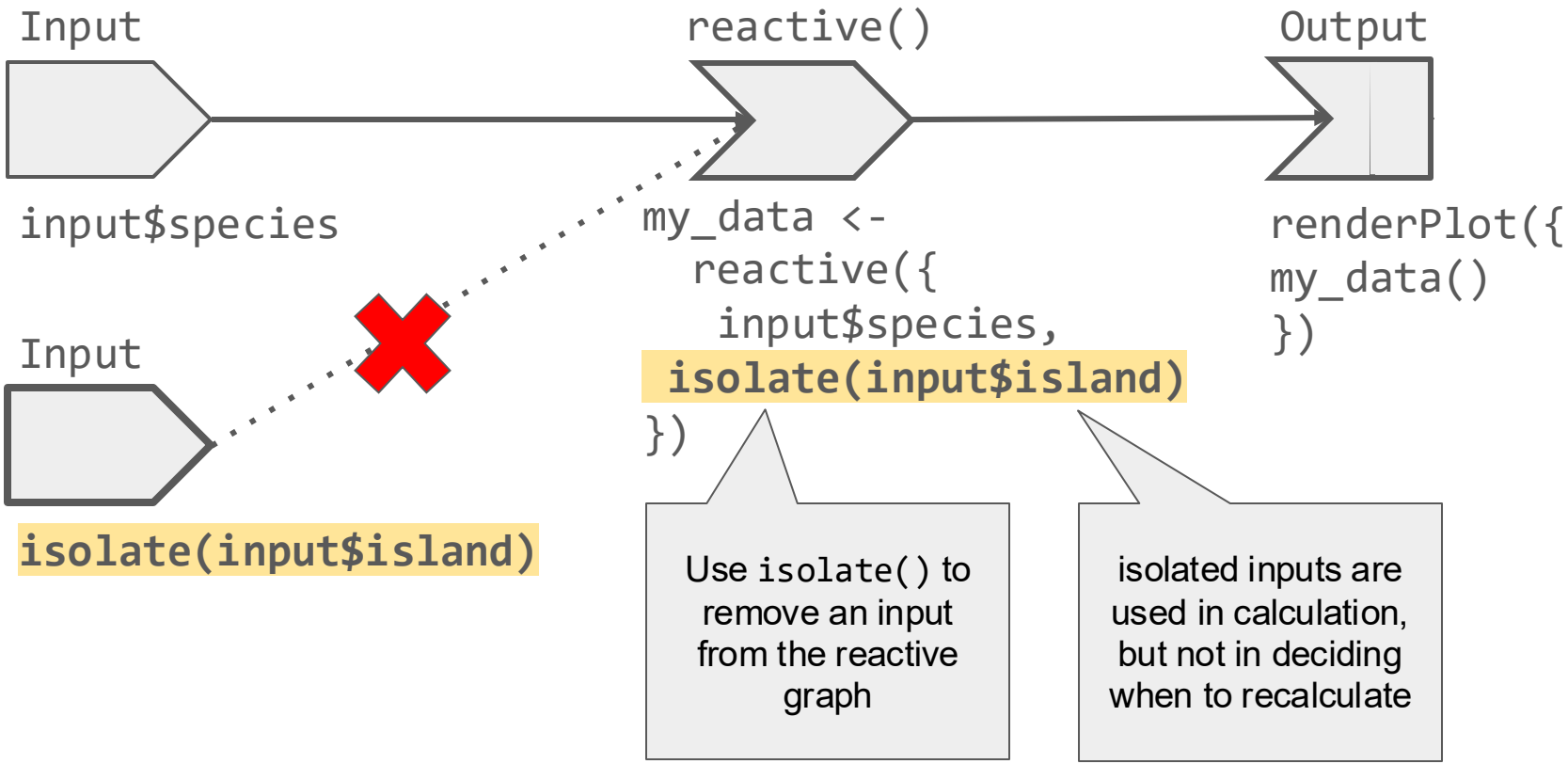
reactive()

runApp("reactives/app_reactive.R")



isolate()

runApp("reactives/app_isolate.R")



Exercise

- With your neighbor, compare the following two apps with reactlog (run the app, and use CTRL+F3 or CMD+Fn+F3)
  - `runApp(“reactivity/app_reactive.R”)`
  - `runApp(“reactivity/app_isolate.R”)`
- How do the graphs differ?
- Change the two controls and step through the graph - what are the differences?

Events

- Want app to respond to some sort of event or change
- Usually an `actionButton`
- but could also be a change in a `reactive()`

Pure Functions vs. Side Effects

Pure Functions

Return some sort of **value** that is used downstream triggered by event (reactives)

Side Effects

Used only for **side effects** triggered by event (doesn’t return a value)

Pure Functions vs. Side Effects

Pure Functions

Return some sort of **value** that is used downstream triggered by event (reactives)

Examples:

Reading a file,
Calculating a new variable,
Filtering data

Side Effects

Used only for **side effects** triggered by event (doesn’t return a value)

Examples:

Updating a Database,
Updating UI elements,
Saving data to a file

Pure Functions vs. Side Effects

Pure Functions

Return some sort of **value** that is used downstream triggered by event (reactives)

Examples:

Reading a file,
Calculating a new variable,
Filtering data

use `eventReactive()` or

`reactive() |> bindEvent()`

lazy execution

Side Effects

Used only for **side effects** triggered by event (doesn't return a value)

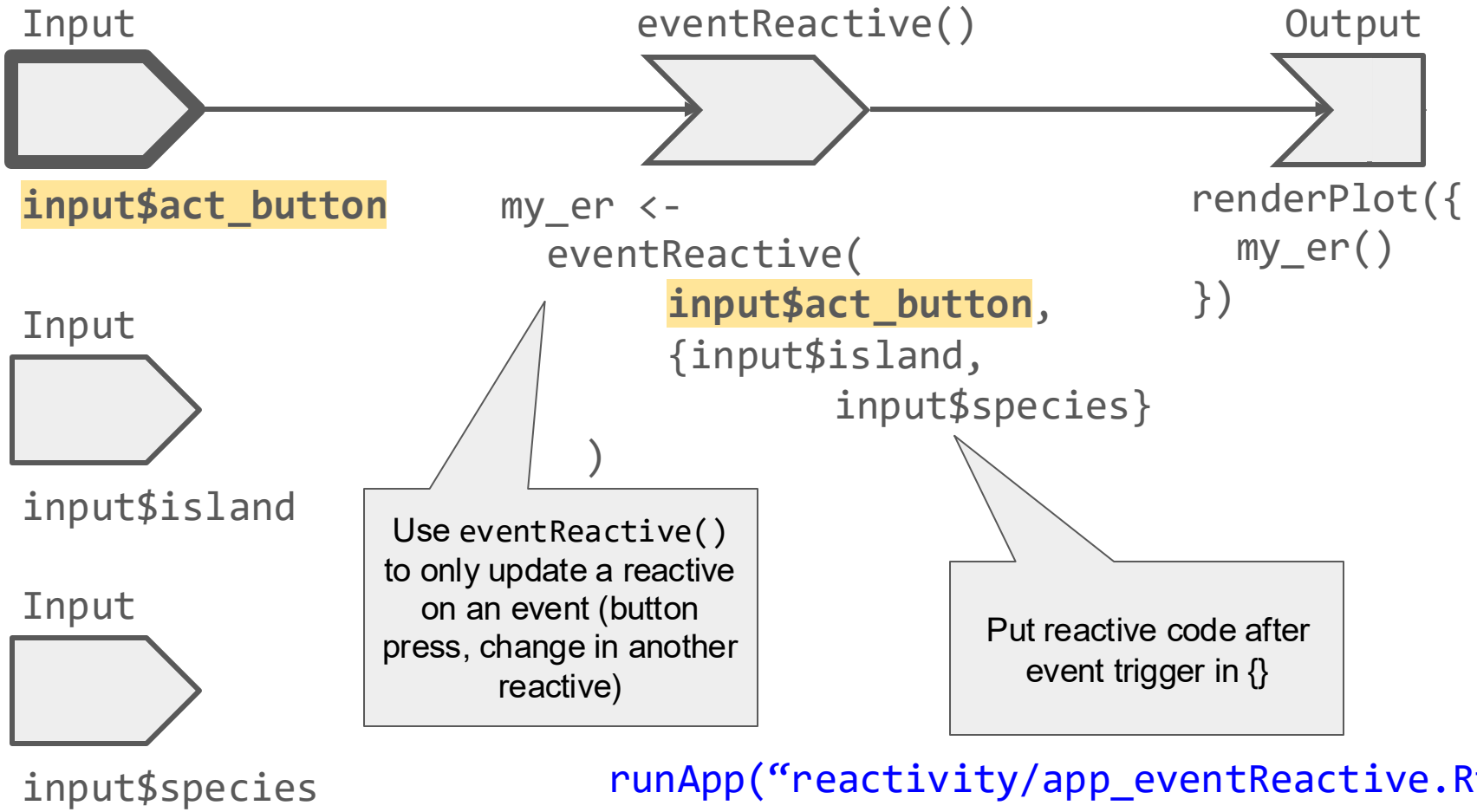
Examples:

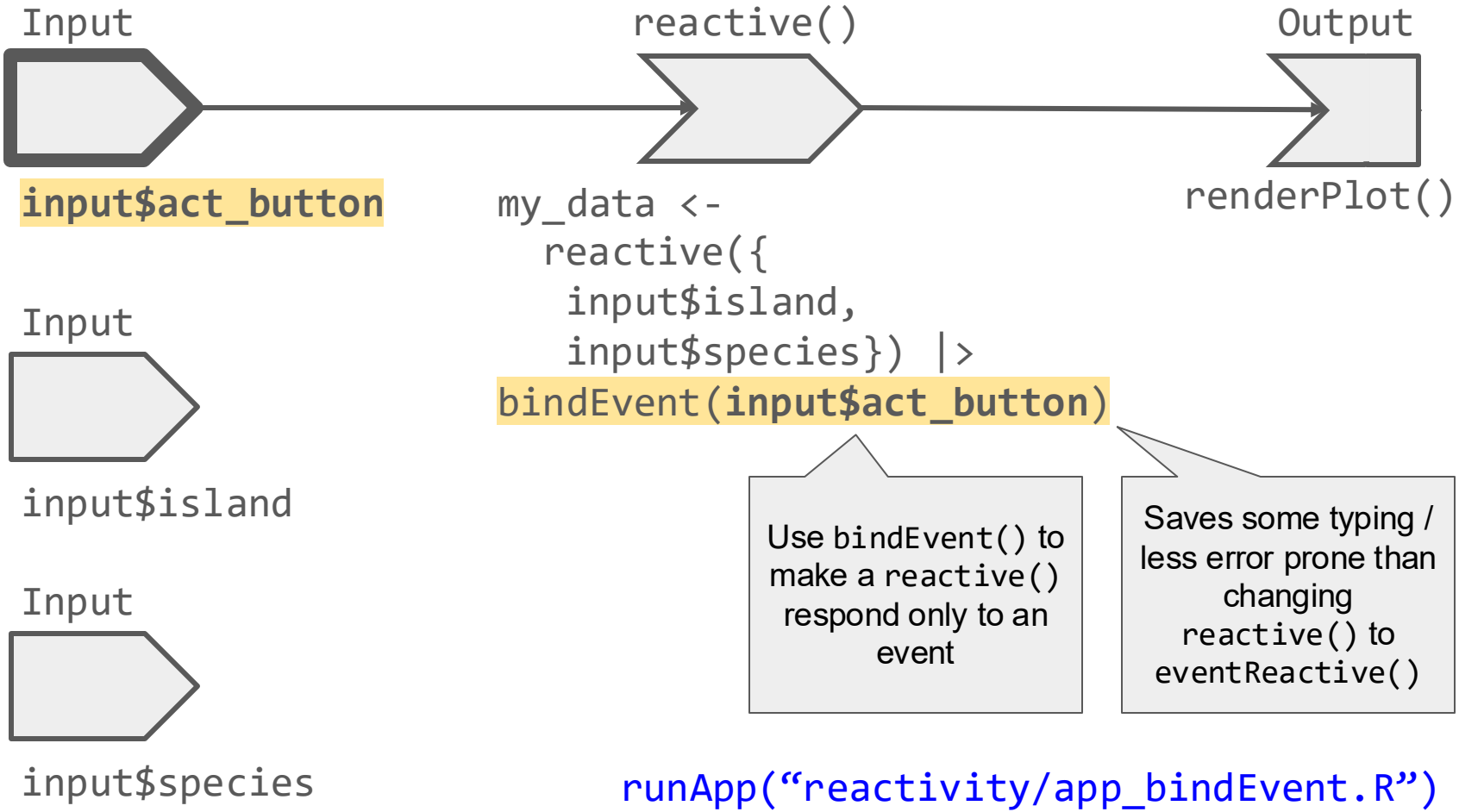
Updating a Database,
Updating UI elements,
Saving data to a file

use `observeEvent()` or

`observe() |> bindEvent()`

eager execution


Input
eventReactive()
Output
input$act_button
my_er <-
eventReactive(
input$act_button,
{input$island,
input$species}
)
renderPlot({
my_er()
})
Input
input$island
Input
input$species
Use eventReactive() to only update a reactive on an event (button press, change in another reactive)
Put reactive code after event trigger in {}
runApp("reactivity/app_eventReactive.R")




```
my_data <-
  reactive({
   input$island,
   input$species}) |>
bindEvent(input$act_button)
```

Use `bindEvent()` to make a `reactive()` respond only to an event

Saves some typing / less error prone than changing `reactive()` to `eventReactive()`

```
runApp("reactivity/app_bindEvent.R")
```

Why does lazy execution matter?

- Shiny tries to minimize the number of calculations
- Only tries to calculate outputs that are visible
- Reactive graph helps decide when to recalculate
- Invalidation -> calculation -> ready to display

runApp("reactivity/app_observeEvent.R")

input$act_button

```
observeEvent(
input$act_button,{
 dbAppendTable()
})
```

Input

input$island

Use `observeEvent()` only for actions that don't generate a value, such as database updates. It will run right away (eager execution)

Input

input$species

```
observe({dbAppendTable()}) |>
bindEvent(input$act_button)
```

Don’t mix values and side effects

Keep your side effects

Outside of your reactives

Or I will kill you.

Joe Cheng

Put values in a separate `eventReactive()` and the side effects in a separate `observeEvent()` that respond to the same event

runApp("reactivity/app_observe_update_ui.R")

Using observeEvent() to update UI

Change in `input$n` is the triggering event

```
ui <- fluidPage(
numericInput("n", "Simulations", 10),
actionButton("simulate", "Simulate")
)
```

When `input$n` changes, read the value

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

 observeEvent(input$n,
{label <- paste0("Simulate ",
input$n, " times")
updateActionButton(inputId =
"simulate", label = label)
})

}
```

When `input$n` changes, update the button UI label using `updateActionButton()`

Good uses of `observeEvent()`

- Dynamically updating UI based on User Inputs
  - `updateActionButton()` example
- Writing lines to a database
- Printing to the console
- Saving a File
- Doing R Stuff that doesn’t produce any output

`observe()` + `reactiveValues()`

- Often an anti-pattern
- Programmers try to use it to force order of execution
- Can be hard to test because of race conditions when multiple `observe()` statements are updating a `reactiveValues()` object
  - Example: reactiveValues with bank balance
- Use sparingly
- `ExtendedTask` is an exception

Exercise

- Do this with your neighbor and discuss
- Compare the reactive graphs between (CTRL+F3 or CMD+F3)
  - `runApp(“reactivity/app_eventReactive.R”)`
  - `runApp(“reactivity/app_observeEvent.R”)`
- How are they wired differently?
- Try triggering the event (pushing the button) and step through the graph

Reactivity: Optimizing

Optimizing Your Shiny App

- Common Sense Stuff
- Leverage Reactivity
- `reactivePoll()`
- `ExtendedTask`
- `bindCache()`

Common Sense Approaches

- Define “real-time” for your app
- Precompute as much as possible (memory is cheap, compute time is not)
- Decouple data pulls from your shiny app if possible
  - Example: Scheduled process that pulls every 10 minutes

Leverage Reactivity

- Reactivity means that only *visible* outputs are recalculated
- Using tabs / panels to only show the relevant parts of an app at once
- `bsCollapse()` / `bsCollapsePanel()`/`updateCollapse()`

`reactivePoll()`

- Use for expensive data updates
- Use when connected to a data source that updates itself
- Only updates the reactive when the data has changed
- You need to write a function that tests for changes in it

https://github.com/rstudio/shiny-examples/tree/main/059-reactive-poll-and-file-reader

`bslib::input_task_button()` for `ExtendedTask`

Won’t trigger an event multiple times

```
runApp(“app_input_task_button.R”)
```

`ExtendedTask`

- Lets you run a long running operation in the background (slow API call, etc)
- Non-blocking: you and other users can work with your app without being interrupted by the long task
- Uses `future_promise()` to spin off into its own R-session
- Running models, long calculations, etc
- `ExtendedTask` is an `R6` object
- YouTube Video: https://www.youtube.com/watch?v=GhX0PcEm3CY

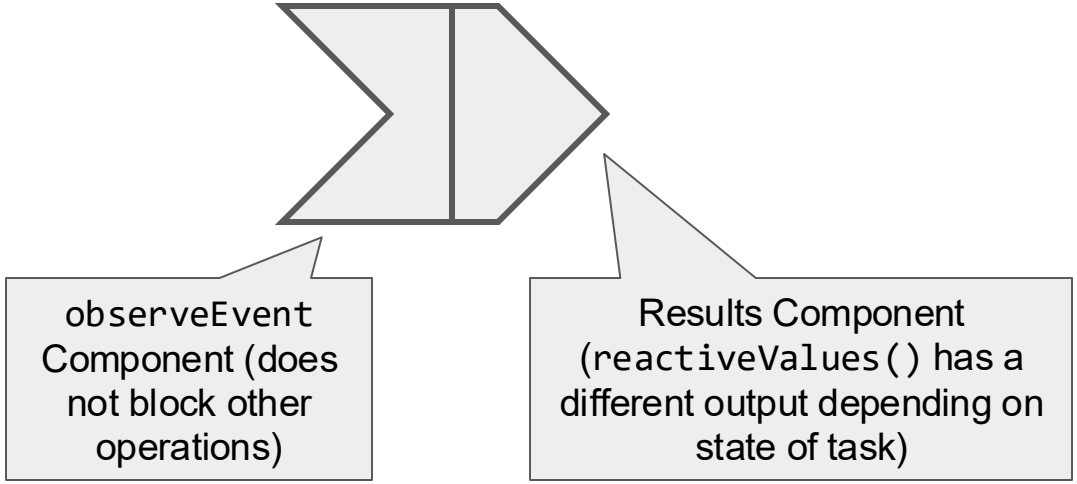
Setup for ExtendedTask

```
library(future)

library(promises)

future::plan(multisession)
```

Add this in your setup code (before ui and server)

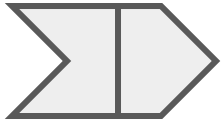
Sets up multiple sessions for future (lets you spin off R-sessions)

`ExtendedTask` is Two parts



ExtendedTask Methods (use in server)

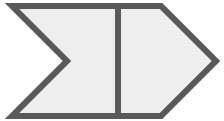
ExtendedTask



```
my_task <-
ExtendedTask$new(
  future_promise(
  function(x){}
))
```

Use `$new` method to make a new ExtendedTask

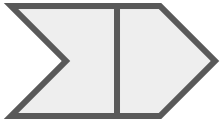
ExtendedTask



```
my_task$invoke(input$x)
```

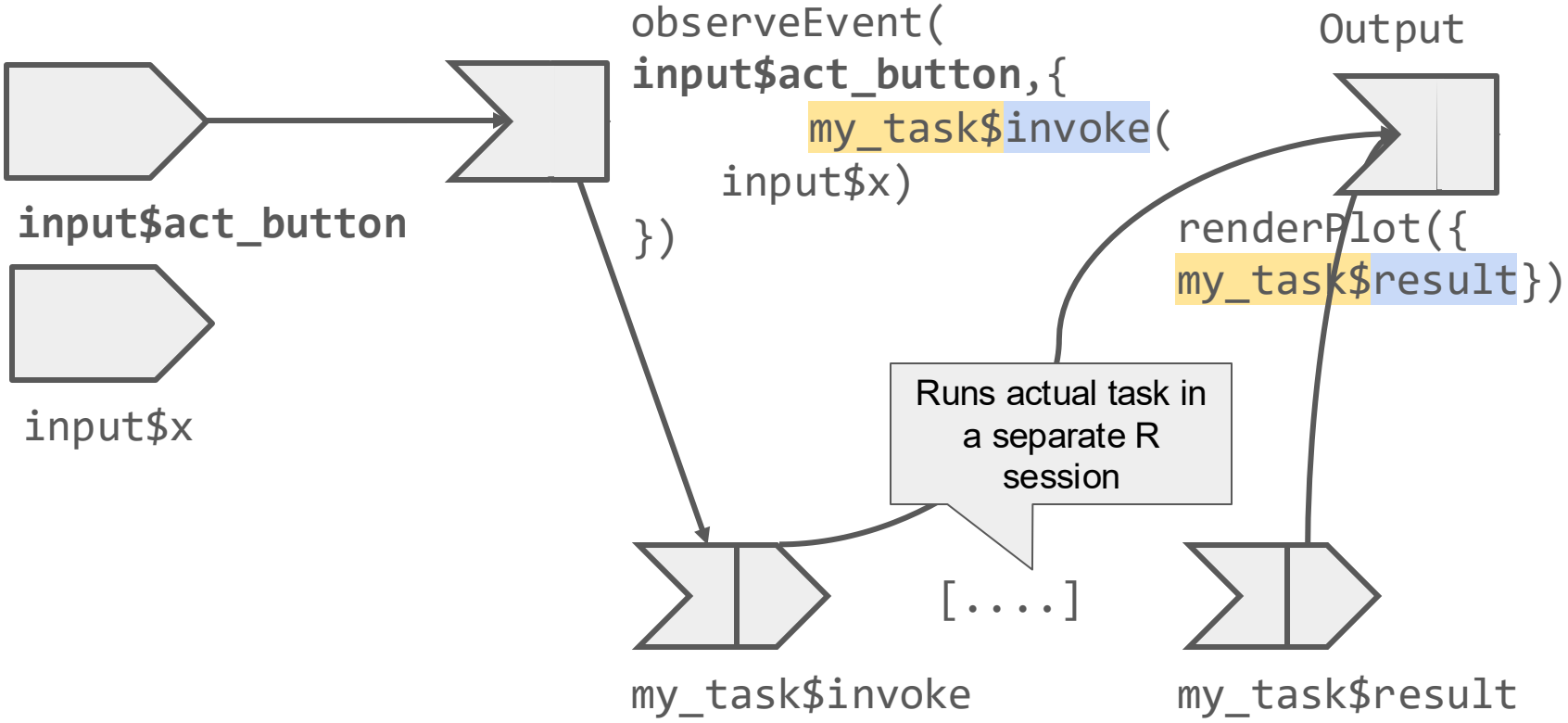
Use `$invoke` method to start a task up (runs in a separate session)

ExtendedTask



```
my_task$result
```

Use `$result` method to use results



runApp("reactivity/app_ExtendedTask.R")

Exercise (if we have time)

Examine the Reactive Graph using reactlog for the following application:

- `runApp(“app_ExtendedTaskSingle.R”)`

Click the button and trace the path

`bindCache()`

- Shiny usually only caches the last value
- `bindCache()` lets you cache values based on an event
  - Shared across all users of the app
  - Can change to per-session
- Can be a reactive or a plot
- Saves initial computation to reduce load time