Reactivity in the `server`

Justin Post

Recap

- `app.R` file contains `ui`, `server`, and code to run the app
- UI can be built in many ways!
    - `bslib` functions give nice layouts and functionality (`page_sidebar()`, `cards()`, `value_box()`, etc.)
- Widgets (`*Input` functions), Text, HTML elements, etc. are added to the UI
- `render*` functions go in the server with `*Output` functions in the UI
    - Server code can access widget inputs via `input$*`

Recap: Server file

`server` also called the 'back-end' because it works behind-the-scenes

```
## set up server
server <- function(input, output, session) {
  # add stuff
}
```

Recap: Server file

`server` also called the 'back-end' because it works behind-the-scenes

```
## set up server
server <- function(input, output, session) {
  # add stuff
}
```

The arguments for the server are `input`, `output`, and `session`. Allow us to

1. Take in inputs from the UI
2. Run functions on them
3. Create outputs to send back

Recap: Accessing Input Values in `server`

- Every input object has an `inputId`
- In `server.r`, reference input value by

```
input$inputId
```

- Example

```
#input widget code from ui.r file
sliderInput(inputId = "slide",label = "Select the Range Here",min = 0,max = 1,
            value = c(0,1))

#reference in server might look like
output$userPlot<-renderPlot({
  range<-input$slide
  #create plot that changes based on user input
  plot(data,xlim=range)
})
```

Recap: Creating Output to Send to UI

Example syntax

```
server <- function(input, output, session) {
  output$nameOfOutputObject <- renderPlot({
    #code that will return a plot
  })

  output$otherOutput <- renderText({
    #code that will return something that R can coerce to a string
  })
}

#in ui.r file, reference would look like
plotOutput("nameOfOutputObject")
textOutput("otherOutput")
```

Input and Output

- `input` and `output` objects are kind of like **lists**
- Shiny passes the information back and forth through them

- Notice how we name our output objects

```
output$nameOfOutputObject <- renderPlot(...)
```

- Notice how we access our inputs

```
output$nameOfOutputObject <- renderPlot(
   range <- input$slide
   ...
)
```

Reactivity

- Output objects do not have to depend on an input
- Those that don't will be static
  - Any 'chunk' of code in `server` that references a user input must be **reactive**
  - When a user changes an input the `input$` value associated invalidates and causes dependent chunks of code to **re-evaluate** in the server

Example Reactivity

```
##code chunk "reacts" to and re-evaluates if
##input$sampleSize or input$otherInput changes

output$dataPlot <- renderPlot({

  n <- input$sampleSize
  input$otherInput #not used anywhere else, but entire
                   #renderPlot chunk still re-evaluates
                   #if changed

  hist(rbinom(n = 1, size = n, prob = 0.4))

})
```

- type `shiny::runExample("01_hello")` into the console

Reactivity

- `server` can run any R code, but can't access inputs unless put into a reactive context
- All `render*` functions are reactive contexts

Error Using Reactive Variables

This type of error is common when first starting!

```
server <- function(input, output, session) {
  print(input$numeric_value + 10) #error due to this!
  output$string <- renderText({
    paste("value plus 10 is", input$numeric_value + 10)
    })
}
```

> Warning: Error in .getReactiveEnvironment()$currentContext: Operation not allowed without an active reactive context. (You tried to do something that can only be done from inside a reactive expression or observer.)

Reactive Contexts

- `render*()` functions
- Functions that can return an object:
  - `reactive({})` creates a reactive context and allows for the creation of a new variable
  - `reactiveValues({})` similar to `reactive` but is easier to create multiple items
  - `eventReactive({})` allows for easy control of reevaluation
- Functions that allow for side-effects
  - `observe({})` function allows for reactivity and reevaluation of code
  - `observeEvent({})` similar to observe but allows for more control over revaluation of code

Reactvity Examples

- Let's download and run this sampling distribution app
- `clone` it!
- Run the app with: `runApp(display.mode = "showcase")`

More on `reactive({})`

- 'Wraps' a normal expression to create a reactive expression (code user can cause to change)
  - Can read reactive values and call other reactive expressions
  - Only re-evaluates *if necessary*
  - Usually used to save something you'll call in multiple other places
  - Access object as though calling it as a function

More on `reactive({})`

- Access object as though calling it as a function

```
server <- function(input, output, session) {
  #Creates a new reactive variable
  newVar <- reactive({
    value <- c(input$NI + 10, input$NI * 3)
  })

  output$textString <- renderText({
    value <- newVar()  #access like a function!
    paste0("Input plus 10 is ", value[1], " and Input times 3 is ", value[2])
  })

  output$otherString <- renderText({
    value <- newVar()
    paste0(value[1], ",", value[2])
  })
}
```

`reactiveValues()`

- Create list like object with `reactiveValues()`
- Access elements via `$`
- Elements can be **modified** in a reactive context

```
server <- function(input, output, session) {
  #Creates a new reactive values
  vals <- reactiveValues(data = rnorm(150), initial = 0)

  output$textString <- renderText({
    paste0("The value of initial is ", vals$initial)
  })

  output$hist <- renderPlot({
    hist(vals$data)
  })
}
```

observe({})

- Can read reactive values and call reactive expressions
  - *Automatically* re-execute when any dependencies change
  - Doesn't yield a result - just re-executes the code

```
server <- function(input, output, session) {
  #would now print to console
  observe({
    print(input$NI+10)
    })

  #update UI
  observe({
    input$noPitch
    updateCheckboxGroupInput(session, "pitchTypeChoice", selected = c(""))
  })
}
```

observeEvent({})

- Similar to observe but allows for control of dependencies
  - Place explicit dependencies prior to `{}`
  - Useful when writing things to a database or file

```
server <- function(input, output, session) {
  #would now print to console
  observeEvent(input$NI, {print(input$data)})

  #update UI
  observeEvent(input$noPitch, {
    updateCheckboxGroupInput(session, "pitchTypeChoice", selected = c(""))
  })
}
```

Developing an App

Highly Recommended:

1. Draw out what you want the app to look like
    - Determine UI elements, what you want the user to control
    - Map out reactivity required in server
2. Create static code that works
    - Write R code in a script or quarto file that does what you need with static inputs
    - Produce plots, tables, text, etc. required for app
3. Translate to appropriate Shiny `render*` and `*Output` functions

Recap

Reactive Contexts can use inputs from widgets

- `render*()` functions
- Functions that can return an object:
  - `reactive({})`
  - `reactiveValues({})`
  - `eventReactive({})`
- Functions that allow for side-effects
  - `observe({})`
  - `observeEvent({})`

Back to the Tutorial!

- Complete all of 'Build an App' on the tutorial
- Then you are ready to complete your first homework assignment!