While I do not doubt there are additional tricks with modularizing Shiny code, this is the last one I have for you. Technically I have already written the pieces of it, but sometimes completing the puzzle is necessary.

Something I have not mentioned but has been true for the example applet is a bug concerning \*\*randCount\*\*. Each time the number of modules, \*\*modCount\*\*, is changed, the value of \*\*randCount\*\* for each module is reset to the default, which is 3. There is a fix for this, and actually a somewhat elegant one I think, and the first step is to change \*\*exampleUI\*\* a little.

```{r echo = TRUE, eval = FALSE}

exampleUI <- function(name, VAL = 3) {

ns <- NS(name)

tagList(

fluidRow(

column(2, numericInput(inputId = ns('randCount'), label = paste0("Count ", name), value = VAL, min = 0)),

column(2, checkboxInput(inputId = ns('striping'), label = "Table Striping")),

),

tableOutput(ns('TAB'))

)

}

```

As you can see, I have replaced the explicitly given \*\*value\*\* argument for the \*\*numericInput\*\* to instead take the module’s new argument, \*\*VAL\*\*, which has a default of 3. The reason for the issue I described is that each time \*\*modCount\*\* changes, the modules are recreated. This is because \*\*lapply\*\* always runs through the entire list it is given, and each time \*\*exampleUI\*\* is called, that \*\*value\*\* argument is reapplied. By making the \*\*VAL\*\* argument it is possible to set \*\*randCount\*\* value each time the module is created, so while new random numbers will be generated, their count per module can remain unchanged.

Of course a catch to this is we need some way to pass \*\*exampleUI\*\* the current value of \*\*randCount\*\* for each instance, but also recognize when no such value exists, so that a default value is provided. Turns out this is actually not that difficult to do, though I would not consider this solution easiest to think of because this is not how \*\*moduleServer\*\* has been used before.

```{r echo = TRUE, eval = FALSE}

UIcreate <- function(name, VAL = 3) {

moduleServer(name, function(input, output, session) {

if (isTruthy(input$randCount)) return(input$randCount)

return(VAL)

}) |> exampleUI(name, VAL = \_)

}

```

In some but not all respects, calling \*\*moduleServer\*\* is like calling \*\*function\*\*, and in this case we are exploiting the fact it does return a value; specifically the value returned by the \*\*function\*\* call we make within it

Ignoring the contents of \*\*moduleServer\*\* for the moment, the value it returns is passed by a pipe into \*\*exampleUI\*\*. Normally a pipe will pass a value in as the first argument for a function, but that would be \*\*name\*\* for \*\*exampleUI\*\*, so we use the pipe placeholder, “\*\*\_\*\*”. This allows the value, or more generically the object being piped in to go wherever we wish in the function, though it must be a named argument. In other words we do have to specify it is for the \*\*VAL\*\* argument of \*\*exampleUI\*\* for this to work.

Looking inside \*\*moduleServer\*\* we find a conditional using the \*\*isTruthy\*\* function, which is a Shiny function but so handy I somewhat wish it were part of base R. Basically it has multiple reasonable conditions that will return \*\*FALSE\*\*, such as being `FALSE` or `NULL`. In this case it will return \*\*FALSE\*\* whenever there is no \*\*randCount\*\* object within the namespaced \*\*input\*\*, such as when that specific instance did not previously exist. When that is the case, the \*\*VAL\*\* argument of the outer \*\*UIcreate\*\* function will be returned by \*\*moduleServer\*\* and piped into \*\*exampleUI\*\*.

If \*\*randCount\*\* does exist in that namespace, however, then \*\*isTruthy\*\* will return \*\*TRUE\*\*, and then the specific value is returned. It is the calling of \*\*exampleUI\*\* that was resetting \*\*randCount\*\*, so by getting its value before it is called again, and passing it into the UI module, the value does not change. The specific random numbers are, because the modules are being recreated, but their count can be preserved.

Before ending this, there is something more we can do with \*\*UIcreate\*\* that is kind of neat.

```{r echo = TRUE, eval = FALSE}

UIcreate <- function(name, VAL = 3) {

exampleServer(name)

moduleServer(name, function(input, output, session) {

if (isTruthy(input$randCount)) return(input$randCount)

return(VAL)

}) |> exampleUI(name, VAL = \_)

}

```

By calling \*\*exampleServer\*\* outside of \*\*moduleServer\*\*, the namespaces will not be nested, which you can do, but personally I want to avoid and would not be helpful here. It also means that one call of \*\*UIcreate\*\* calls both the UI and server modules, simplifying the applet. Also, it is necessary to call \*\*exampleUI\*\* last, because that is what we want to be returned by \*\*UIcreate\*\*.

```{r echo = TRUE, eval = FALSE}

observe({

output$modules <- renderUI( lapply(modList(), UIcreate, 3) )

output$randCount <- renderText({

COUNT <- unlist(lapply(modList(), function(IN) input[[ NS(IN, "randCount") ]]))

paste0("There are ", sum(COUNT), " random numbers")

})

})

```

As you can see, we are still using \*\*renderUI\*\* and \*\*lapply\*\*, so we need \*\*exampleUI\*\* to be returned as it holds the UI elements.

Just a couple final notes here, with the first being that the 3 at the end of calling \*\*lapply\*\* is not necessary, but it is passed to \*\*UIcreate\*\* as its \*\*VAL\*\* argument. If you prefer the modules start with a different count of random numbers, this is the value to change, as it will propagate to the appropriate places from there.

The other note is I chose to use the pipe here because I do believe it makes the code look cleaner in this case (surrounding the whole block of \*\*moduleServer\*\* is not ideal) and it is a decent example of using it with the placeholder.

That is it; the last advanced trick I have for modularizing Shiny code is to recognize \*\*moduleServer\*\* does return an object, and that can then be taken advantage of.