As I said in the prior, UI section I consider modularizing Shiny server code to be the more complicated of the two parts of an applet, so I will do my best to present the process and format in an understandable way. I also said the UI and server portions will need to work together, so here is what we ended the previous section with:

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

exampleUI <- function(name) {

ns <- NS(name)

tagList(

fluidRow(

column(2, numericInput(inputId = ns('randCount'), label = paste0("Count ", name), value = 3, min = 0)),

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

),

tableOutput(ns('TAB')),

)

}

ui <- fluidPage(

titlePanel(NULL, "Shiny Modularization Example"),

fluidRow(

column(2, numericInput(inputId = 'modCount', label = "Module Count", value = 2, min = 1)),

column(2, numericInput(inputId = 'roundTerm', label = "Rounding Term", value = 2)),

),

uiOutput('modules'),

)

```

While we do need to be concerned about all of these elements the four most important are \*\*randCount\*\*, \*\*TAB\*\*, \*\*modCount\*\*, and \*\*modules\*\*; two inputs and two outputs, as well as two within a namespace and two outside of it. We are going to start with the input and output that are within the namespace, as these will also need to be in the server module. There are two other inputs we want to work with the module, but first things first.

Creating a server module is kind of weird looking, but it is not too bad and opens up some possibilities I will explore in the Advanced sections. I also want to point out that while the official guide states you must use a suffix of “Server” for server modules, this is not necessary, at least not with the version of Shiny (1.7) I have been working with.

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

exampleServer <- function(name) {moduleServer(name, function(input, output, session) {

…

})}

```

As I said, it is weird looking as we have two calls of \*\*function\*\* and one for this new \*\*moduleServer\*\* function. The first \*\*function\*\* call is just like the single call we did for the UI module, in that it is no different from when we normally create a custom function, but we do need an argument, here \*\*name\*\*, for identifying the namespace. If you want to use additional arguments, you may and very likely will, but that is not necessary right here.

The \*\*moduleServer\*\* function is the key to server modules for Shiny as it does a couple things. The first is evident by its first argument, as it expects the name of the appropriate namespace that we have passed to the outer \*\*function\*\* call. If you always knew the value of \*\*name\*\* you could actually just call \*\*moduleServer\*\* directly without the outer \*\*function\*\*. That will not satisfy the goals I have stated of showing multiple tables (or repeating the module multiple times), but does work. I can show you what this looks like, but first I want to get to the code within \*\*exampleServer\*\*, and before that I want to explain the second call to \*\*function\*\*.

Depending on how much time you have spent investigating Shiny, this second \*\*function\*\* may look familiar. It is the generic call for creating the server function for a Shiny applet. This also gives us an idea of what is actually happening when \*\*moduleServer\*\* is used. It is creating a separate Shiny server from the normal one, with this new one being within the identified namespace. This will allow it to access the \*\*input\*\* and \*\*output\*\* objects of that namespace. Importantly, it will apply this namespace without having to explicitly call \*\*NS\*\*, like we did for the UI module earlier. With that, we can look at the code I have placed within \*\*exampleServer\*\*.

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

exampleServer <- function(name) {moduleServer(name, function(input, output, session) {

exTAB <- reactive(exFUN(input$randCount))

output$TAB <- renderTable(exTAB())

})

}

```

If we were to take these two lines I added and use them as the server code to compliment the UI code in the previous section, when I first created \*\*exampleUI\*\* without the namespace stuff, the Shiny applet could be called and would work. Of course \*\*modCount\*\* will not do anything, but I have included an \*app\_non\_modular.r\* script that shows what I just described is functional.

Looking at those two lines as normal lines for the server section of a Shiny applet, we see I am creating the reactive \*\*exTAB\*\* to store the output from \*\*exFUN\*\*. It being a reactive is necessary to ensure it updates with \*\*randCount\*\* and so \*\*randCount\*\* can even be used, as it is a reactive. You may be wondering why I am not just calling \*\*exFUN\*\* within \*\*renderTable\*\*, as that would achieve a similar result, but there is a difference that matters to me. Remembering back to the UI code we have already covered, there are a couple inputs that are meant to control the table here; I want the ability to change those inputs without changing the contents of the table. This can only be achieved by creating the table’s contents outside of \*\*renderTable\*\*.

The next step is to apply \*\*exampleServer\*\* so that multiple instances of the UI and server modules are created, which is actually pretty easy, but there is something I want to do first to clean up the code some.

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

modList <- reactive( as.character(1:input$modCount) )

```

Essentially what \*\*modList\*\* does is store a list of the namespace names, which, because \*\*modCount\*\* is numeric, must be made into strings with \*\*as.character\*\*. It is a \*\*reactive\*\* so that changing \*\*modCount\*\* updates this as well. With this done though, we can get to solving the issue of having multiple instances of the server and UI modules called.

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

observe({

lapply(modList(), exampleServer)

output$modules <- renderUI( lapply(modList(), exampleUI) )

})

```

With \*\*lapply\*\* each value from \*\*modList\*\* is passed to the supplied function, which first is \*\*exampleServer\*\*. Though it does provide an output like any function, we do not care about \*\*exampleServer\*\*’s output, and so just passing it to \*\*lapply\*\* is enough.

Next we create the \*\*modules\*\* output, which requires a call to \*\*renderUI\*\*, as that will have Shiny just render whatever UI elements are passed to it. In this case those are generated by another call to \*\*lapply\*\* but this time it is given \*\*exampleUI\*\* as the function.

To be absolutely clear, by \*\*lapply\*\* working through the values from \*\*modList\*\* in the ways we see here, both \*\*exampleServer\*\* and \*\*exampleUI\*\* are being called with their \*\*name\*\* arguments being those values. Though I am not going to do it just now, it is possible to combine \*\*exampleServer\*\* and \*\*exampleUI\*\* so \*\*lapply\*\* needs only be used once here.

The next thing I want to do within this section is set up some control over \*\*renderTable\*\* within \*\*exampleServer\*\*. That will wrap up this section as what remains to cover will be in the Advanced sections.

The \*\*striping\*\* control is within the UI module, so calling it will be easy, but not quite as easy as just passing \*\*input$striping\*\* to the appropriate argument for \*\*renderTable\*\*.

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

exampleServer <- function(name) {

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

exTAB <- reactive(exFUN(input$randCount))

output$TAB <- renderTable(exTAB(), striped = reactive(input$striping))

})

}

```

As you can see I have had to surround the input with \*\*reactive\*\*. This is because the options for \*\*renderTable\*\* do not natively accept reactive values; only the table expression itself can be a reactive without stating as much. Now when the input changes, the table will be forced to update to whatever the new value is for \*\*striping\*\*. This updating would also trigger new random numbers to be generated if not for the use of \*\*exTAB\*\*, as that is not bound to this input.