**Introduction**

Shiny modules provide a great way to organize and container-ize your code for  
building complex Shiny applications as well as protecting namespace collisions.  
In this post, I am going to cover how to implement modules with  
insertUI/removeUI so that you DRY, clear server-side overhead, and encapsulate  
duplicative-ish shiny input names in their own namespace.

Normally when developing an application, each input provides a unique parameter  
for the output and it is specified by a unique ID. The example below illustrates  
a shiny app that allows the end user to specify the variables for regression.

library(shiny)

data(mtcars)

cols <- sort(unique(names(mtcars)[names(mtcars) != 'mpg']))

ui <- fluidPage(

wellPanel(

fluidRow(

column(4,

tags$h3('Build a Linear Model for MPG'),

selectInput('vars',

'Select dependent variables',

choices = cols,

selected = cols[1:2],

multiple = TRUE)),

column(4, verbatimTextOutput('lmSummary')),

column(4, plotOutput('diagnosticPlot'))

)

)

)

server <- function(input, output) {

lmModel <- reactive({lm(sprintf('mpg ~ %s', paste(input$vars, collapse = '+')),

data = mtcars)})

output$lmSummary <- renderPrint({

summary(lmModel())

})

output$diagnosticPlot <- renderPlot({

par(mfrow = c(2,2))

plot(lmModel())

})

}

shinyApp(ui = ui, server = server)

There is no need to worry about namespace collisions here because you can  
directly control the unique-ness of each input control. Note: If you  
accidentally duplicate the id, Shiny is not going to tell you that from the  
R console. If you open up the browser devtools, you will find an error like  
this:



**Creating insertUI/removeUI Modules**

Now, one linear model is great to start, but I want to try out several  
different models with different variables. I could select the variables,  
take a screenshot of each model, and piece them all together later but I’m  
trying to make it much easier for the end user. Wrapping the app above into a  
module requires a UI function. The NS function is a convenience function to  
create a namespace for the input IDs. In short, when input IDs are created  
later on they will be pre-fixed with lmModel*id*.

lmUI <- function(id) {

ns <- shiny::NS(id)

shiny::uiOutput(ns("lmModel"))

}

This particular module UI may look a bit sparse to other examples. The UI that  
the end user sees is going to be generated later on with renderUI. All  
that this UI needs to do is set the namespace.

Next, the server code needs to be wrapped into a server module. The UI and  
server code is going to be combined for use with renderUI. I also added in a  
delete button that will be the input for removing UI controls. The rendered  
UI is also wrapped with a *div* id. To keep track of each UI Controls, I’m  
using environment(ns)[['namespace']] which is a fancy way to pull out the  
namespace from session$ns. environment gets the environment (space to look  
in for values; similar to namespace) of ns which is storing the namespace id.

lmModelModule <- function(input, output, session) {

lmModel <- reactive({

lm(sprintf('mpg ~ %s',paste(input$vars, collapse = '+')), data = mtcars)

})

output[['lmModel']] <- renderUI({

ns <- session$ns

tags$div(id = environment(ns)[['namespace']],

tagList(

wellPanel(

fluidRow(

column(3,

tags$h3('Build a Linear Model for MPG'),

selectInput(ns('vars'),

'Select dependent variables',

choices = cols,

selected = cols[1:2],

multiple = TRUE)),

column(4,

renderPrint({summary(lmModel())})

),

column(4,

renderPlot({par(mfrow = c(2,2))

plot(lmModel())})

),

column(1,

actionButton(ns('deleteButton'),

'',

icon = shiny::icon('times'),

style = 'float: right')

)

)

)

)

)

})

}

**Dynamic UI/Server Logic**

The modules can be called just as functions can be called. For ease, just place  
the module code at the top of the shiny application script outside of the  
main server/ui functions. The main shiny functions below are even shorter than  
the workhorse module functions. Even in a small application you can start to  
see the benefit of “modularizing” code!

The majority of the code in server is just setting up handling for the module IDs.  
The actionButton increments by 1 each time that it is clicked so I’m using it  
as an ID number. The id tag is doing double duty here: providing the namespace  
for module UI and for the div tag so that we can remove it later on. After  
calling the module, you will want to create an action that will respond to  
deleting the module. The last observeEvent will create that action  
and it will persist with the correct id.

ui <- fluidPage(

br(),

actionButton('addButton', '', icon = icon('plus'))

)

server <- function(input, output) {

observeEvent(input$addButton, {

i <- sprintf('%04d', input$addButton)

id <- sprintf('lmModel%s', i)

insertUI(

selector = '#addButton',

where = "beforeBegin",

ui = lmUI(id)

)

callModule(lmModelModule, id)

observeEvent(input[[paste0(id, '-deleteButton')]], {

removeUI(selector = sprintf('#%s', id))

remove\_shiny\_inputs(id, input)

})

})

}

shinyApp(ui = ui, server = server)

**Cleaning up Server Side**

removeUI will delete the contents on the client side, but the inputs will  
still exist on the server side. Currently, removing the inputs on the server  
side is not implemented in the shiny package.

remove\_shiny\_inputs <- function(id, .input) {

invisible(

lapply(grep(id, names(.input), value = TRUE), function(i) {

.subset2(.input, "impl")$.values$remove(i)

})

)

}

I used the latter to pass that all important id to look up all inputs in  
that namespace and remove them. Inputs are protected from directly using  
input[[inputName]] <- NULL to delete them. For the outputs on the server  
side, I haven’t been able to find that much documentation on what happens to it.  
I know that they still exist as at least a named entry on the server side.  
it is possible to remove them, but it doesn’t appear their name slots go away.  
Debugging and using outputOptions still listed the output, but setting the  
output to NULL will delete them from what the user sees on the shiny  
application.

**Final Thoughts**

Modules, insertUI, and removeUI have added some very impressive features to  
Shiny. It has opened up a much more on-the-fly interface for Shiny  
developers. I’m hoping development around these ideas continue. My first crack at this, I didn’t use the NS framework at all, but  
essentially used the same method so there is more than one way to do this.  
Using NS will save you leg work. Here is an example of my first attempt:

observeEvent(input$add, {

i <- input$add

id <- sprintf('%04d', i)

inputID <- sprintf('input-%s', id)

insertUI(

selector = "#add",

ui = tags$div(id = inputID,

numericInput(id, 'A number')

)

)

})

Possible enhancment: In the above code, I was using integers from the action button increment so that I could easily see that Shiny was doing what I expected it to  
do. An enhancement would be to generate something like a guid so that you  
wouldn’t have to worry about what happens when multiple users in the same  
app are clicking. This might not be needed if the action button increments are  
per user per session. I still have some homework to do on the namespacing in  
Shiny and client/server data persistence.