Foremost in your mind should be the quintessential reality of R: [Everything that happens in R is the result of a function call](https://drdoane.com/three-deep-truths-about-r/). Shiny is no exception.

To write a minimal shiny app, you create an object that describes your app’s user interface, write a function describing runtime behaviors, and pass them into a function that spins up the app you’ve described. By convention, these two objects are associated with the variable names **ui** and **server**.

library(shiny)  
ui <- fluidPage()  
server <- function(input, output, session) {}

This is just R code. You can type it into the Console to execute it line by line and inspect what it does.

If you’re working in RStudio, you can type it into a Source file, then press Control-Enter (Windows) or Command-Return (MacOS) to send each line to the Console for execution.

Checking the Environment—or the structure of these two objects with **str()**—we can see that **ui** is a list of three objects. If we print **ui** to the Console, we see only an empty HTML

element.

The object associated with **server** is simply a function with no body.

To execute this minimal shiny app, we pass the **ui** and **server** objects to the **shinyApp()** function.

shinyApp(ui, server)

The app will be spun up either in RStudio’s Viewer pane, in a Viewer window, or in your default Web browser, depending on your settings in RStudio.

Don’t be surprised: it will be just a blank window, since all that has been defined thus far is an empty

element. The document that opened is an HTML document with some boilerplate CSS and JavaScript. You can inspect it using your Browser’s Developer Tools.

That’s it. That’s shiny. Everything else flows from these core ideas:

* **ui** is a list object representing the HTML UI to be constructed.
* **server** is a function describing the runtime behavior of your app.
* **shinyApp()** takes these two objects and uses them to construct an HTML document that then gets spun up in a browser.

**A Shiny App that does Something**

Let’s take the empty shell and start adding some content and behaviors. We’ll use whitespace to make our code more readable.

library(shiny)  
  
 ui <- fluidPage(  
  **textOutput("output\_area"),  
 actionButton(  
 "next\_num",   
 "Click to generate a random number"  
 )**  
 )  
  
 server <- function(input, output, session) {  
 **observeEvent(input$next\_num, {  
 output$output\_area <- renderText({  
 rnorm(1)  
 })  
 })**  
 }  
  
 shinyApp(ui, server)

Since the elements of **ui** are passed in as arguments to a layout function—**fluidPage**, here—they are separated by commas. Since the content of **server** is just a function definition, its parts are not separated by commas.

It may not be obvious in a short example like this, but the organization of Shiny code can get out of control quickly: you start with 5, 10, or 20 UI blocks, then a set of server behaviors that somehow align with those UI blocks. The UI code and the code that dictates runtime behavior for those UI elements can be hundreds of lines separated from each other. It’s easy to make spaghetti, but we’d rather have ravioli.

**Refactoring Toward Shiny Modules**

Let’s refactor this by extracting to functions the UI elements and server code. To signal my intention that these two parts belong together, I’ll adopt a naming convention of calling them the same thing with the UI elements having \_UI suffixed to it. Since there are multiple items being returned from the \_UI function, I’ll need to bundle them together in a list.

library(shiny)  
  
 **button\_UI <- function() {  
 list(  
 textOutput("output\_area"),  
 actionButton(  
 "next\_num",   
 "Click to generate a random number"  
 )  
 )  
 }**  
 **button <- function() {  
 observeEvent(input$next\_num, {  
 output$output\_area <- renderText({  
 rnorm(1)  
 })  
 })  
 }**  
 ui <- fluidPage(  
 **button\_UI()** )  
  
 server <- function(input, output, session) {  
 **button()** }  
  
 shinyApp(ui, server)

That doesn’t quite work, but it’s close. The immediate problem is with the environment scope of the **input** variable: **button()** wants to know about it, but I haven’t passed them in to **button()**. Let’s fix that and, while we’re at it, pass in output and **session**, too.

library(shiny)  
  
 button\_UI <- function() {  
 list(  
 textOutput("output\_area"),  
 actionButton(  
 "next\_num",   
 "Click to generate a random number"  
 )  
 )  
 }  
  
 button <- function(**input, output, session**) {  
 observeEvent(input$next\_num, {  
 output$output\_area <- renderText({  
 rnorm(1)  
 })  
 })  
 }  
  
 ui <- fluidPage(  
 button\_UI()  
 )  
  
 server <- function(input, output, session) {  
 button(**input, output, session**)  
 }  
  
 shinyApp(ui, server)

It works! This is two-thirds of the way to applying the Shiny Modules design pattern.

Why isn’t this sufficient? What more could we possibly need? At the moment, all of these objects and the element IDs they create are being created at the top level of the running Shiny application. In other words, all the functions I write this way are sharing a single namespace. That’s not ideal, if I want to create a second button and output field. I would need to write two more extracted functions, come up with more globally unique IDs—**output\_area2**, **output\_area3**, … **next\_num2**, **next\_num3**… it all would get messy quickly and totally unmanageable at scale.

The solution is to have those two extracted functions exist in their own namespace so that I could instantiate another pair of them, then another, then another, without needing to concern myself with name collisions.

The Shiny Modules pattern achieves this by having you provide a unique ID each time you instantiate this pair of functions. **shiny::NS()** takes the **id** you provide and returns a function that will pre-pend the **id** onto the individual UI elements’ IDs.

button\_UI <- function(**id**) {  
 **ns = NS(id)**  
 list(  
 textOutput(**ns("output\_area")**),  
 actionButton(  
 **ns("next\_num")**,   
 "Click to generate a random number"  
 )  
 )  
}

To make the module’s server fragment aware of that namespace, you use **shiny::callModule()** in your main server function located in app.R. You pass in the ID of the instance of the UI you want that module to work with.

ui <- fluidPage(  
 button\_UI(**"first"**)  
 )  
  
 server <- function(input, output, session) {  
 callModule(button, **"first"**)  
 }

Let’s look at the complete code:

library(shiny)  
  
 button\_UI <- function(id) {  
 ns = NS(id)  
  
 list(  
 textOutput(ns("output\_area")),  
 actionButton(  
 ns("next\_num"),   
 "Click to generate a random number"  
 )  
 )  
 }  
  
 button <- function(input, output, session) {  
 observeEvent(input$next\_num, {  
 output$output\_area <- renderText({  
 rnorm(1)  
 })  
 })  
 }  
  
 ui <- fluidPage(  
 button\_UI("first")  
 )  
  
 server <- function(input, output, session) {  
 callModule(button, "first")  
 }  
  
 shinyApp(ui, server)

That’s the Shiny Module design pattern!

It’s worth pointing out that the **input**, **output**, and **session** input parameters to a module are *not* the same as the values of **input**, **output**, and **session** that exist in your shiny app as a whole. Within the server fragment of your module, **input**, **output**, and **session** are scoped to your module’s namespace.

**Separate Modules into their own Directories**

For the sake of code organization, you can move the two module functions to their own script file and **source()** them into your app.R and since you might end up having many modules, you might want to create a separate directory to contain them (I call mine **modules/**):

**modules/button\_mod.R**

button\_UI <- function(id) {  
 ns = NS(id)  
  
 list(  
 textOutput(ns("output\_area")),  
 actionButton(  
 ns("next\_num"),   
 "Click to generate a random number"  
 )  
 )  
 }  
  
 button <- function(input, output, session) {  
 observeEvent(input$next\_num, {  
 output$output\_area <- renderText({  
 rnorm(1)  
 })  
 })  
 }

**app.R**

library(shiny)  
  
 **source("modules/button\_mod.R")**  
 ui <- fluidPage(  
 button\_UI("first")  
 )  
  
 server <- function(input, output, session) {  
 callModule(button, "first")  
 }  
  
 shinyApp(ui, server)

Now, in app.R, you can reuse your module freely… simply give each instance its own ID:

**app.R with Module Reuse**

library(shiny)  
  
 source("modules/button\_mod.R")  
  
 ui <- fluidPage(  
 button\_UI("first"),  
 **button\_UI("second")** )  
  
 server <- function(input, output, session) {  
 callModule(button, "first")  
 **callModule(button, "second")** }  
  
 shinyApp(ui, server)

You can click freely on each of the two buttons without it having an effect on the other.

With this pattern you have:

* modularity
* distinct namespaces for each instance (encapsulation)
* reusability

all of which is derived from the fact that these are just R functions.

The usual rules of well-defined functions apply:

* within a module’s server fragment, code must rely only on objects it creates or that are explicitly passed into the module as additional arguments to **callModule().**
* any objects you create within a module that you need access to in other modules must be returned from the producing module as a reactive expression, captured by a variable assignment, and passed into the consuming module(s) where it will be invoked as a function call to that reactive expression.

produceMod <- function(input, output, session) {  
 **reactive(rnorm(1))**  
}  
  
consumeMod <- function(input, output,   
 session, **data**) {  
 output$someOutput <- renderText({  
 **data()** + 100  
 })  
}  
...  
server <- function(input, output, session) {  
 **result <-** callModule(produceMod, "someID")  
 callModule(consumeMod, "", **result**)  
}