As Shiny applications grow larger and more complicated, we use modules to manage the growing complexity of Shiny application code.

Functions are *the* fundamental unit of abstraction in R, and we designed Shiny to work with them. You can write UI-generating functions and call them from your app, and you can write functions to be used in the server function that define outputs and create reactive expressions.

In practice, though, functions alone don't address the problem of organising and managing large and complex app code completely. Input and output IDs in Shiny apps share a global namespace, meaning, each ID must be unique across the entire app. If you're using functions to generate UI, and those functions generate inputs and outputs, then you need to ensure that none of the IDs collide.

In computer science, the traditional solution to the problem of name collisions is *namespaces*. As long as names are unique *within* a namespace, and no two namespaces have the same name, then each namespace/name combination is guaranteed to be unique. Many systems will let you nest namespaces, so a namespace doesn't need a name that's globally unique, just unique within its parent namespace.

Shiny modules address the namespacing problem in Shiny UI and server logic, adding a level of abstraction beyond functions.

Note: Prior to Shiny 1.5.0, we recommended using `callModule()` to invoke modules. From 1.5.0 onward, we recommend using `moduleServer()` instead, because it has simpler syntax for using the module. Additionally, new-style modules can be tested with the `testServer()` function, also introduced in Shiny 1.5.0. If you would like to see how to migrate from old to new-style modules and see how they compare, see the Migrating from `callModule` to `moduleServer` section, below.

A Simple Module

Here's a small application demonstrating a simple "counter" module:

```
library(shiny)

counterButton <- function(id, label = "Counter") {
  ns <- NS(id)
  tagList(
    actionButton(ns("button"), label = label),
    verbatimTextOutput(ns("out"))
  )
}

counterServer <- function(id) {
  moduleServer(
    id,
    function(input, output, session) {
      count <- reactiveVal(0)
      observeEvent(input$button, {
        count(count() + 1)
      })
      output$out <- renderText({
        count()
      })
      count
    }
  )
}

ui <- fluidPage(
  counterButton("counter1", "Counter #1")
)

server <- function(input, output, session) {
  counterServer("counter1")
}

shinyApp(ui, server)
```

1. `NS()` is used within `counterButton()` to encapsulate the module's UI.
2. `counterServer()` is a function which calls `moduleServer()`.
3. The call to `moduleServer()` is passed the `id`, as well as a **module function**.
4. In this particular example, the module function returns a reactive value, but it could return any value.
5. Inside of the applications `server` function, `counterServer()` is called to initialize the module.

If that all makes sense to you, great! Otherwise, read on for more details.

Introducing Shiny Modules

A Shiny module is a piece of a Shiny app. It can't be directly run, as a Shiny app can. Instead, it is included as part of a larger app (or as part of a larger Shiny module – they are composable).

Modules can represent input, output, or both. They can be as simple as a single output, or as complicated as a multi-tabbed interface festooned with controls/outputs driven by multiple reactive expressions and observers.

Once created, a Shiny module can be easily reused – whether across different apps, or multiple times in a single app (like a set of controls that needs to appear on multiple tabs of a complex app). One possible motivation for building modules is to bundle them into R packages to be used by other Shiny authors. Another possible motivation is to break up a complicated Shiny app into separate modules that can each be reasoned about independently; such modules likely have no potential for reuse but they serve an important purpose within an app.

Creating Shiny Modules

A module is composed of two functions that represent 1) a piece of UI, and 2) a fragment of server logic that uses that UI – similar to the way that Shiny apps are split into UI and server logic.

Indeed, the contents of your UI and server functions will look a lot like normal Shiny UI/server logic. But the packaging needs to differ in a few important ways.

Creating UI

A module's UI function should be given a name that is suffixed with `Input`, `Output`, or `UI`; for example, `csvFileUI`, `zoomableChoroplethOutput`, or `tabOneUI`.

The first argument to a UI function should always be `id`. This is the namespace for the module. (Note that the namespace for the module is decided by the *caller* at the time the module is *used*. This will make more sense later, when we talk about how modules are invoked.)

Here's an example for a CSV file input module:

```
# Module UI function
csvFileUI <- function(id, label = "CSV file") {
  # `NS(id)` returns a namespace function, which was save as `ns`
and will
  # invoke later.
  ns <- NS(id)

  tagList(
    fileInput(ns("file"), label),
    checkboxInput(ns("heading"), "Has heading"),
    selectInput(ns("quote"), "Quote", c(
      "None" = "",
      "Double quote" = "\"",
      "Single quote" = "'"
    ))
  )
}
```

The body of this function looks quite similar to the UI code for a Shiny app. The main differences are:

1. The function body starts with the statement `ns <- NS(id)`. All UI function bodies should start with this line. It takes the string `id` and creates a *namespace function*.
2. All input and output IDs that appear in the function body needs to be wrapped in a call to `ns()`. This example shows `inputId` arguments being wrapped in `ns()`, e.g. `ns("file")`). If we happened to have a `plotOutput` in our UI, we would also want to use `ns()` when declaring its `outputId` or `brush` ID, for example.
3. The results are wrapped in `tagList`, instead of `fluidPage`, `pageWithSidebar`, etc. You only need to use `tagList` if you want to return a UI fragment that consists of multiple UI objects; if you were just returning a `div` or a single input, you could skip `tagList`.

Admittedly, the `ns()` mechanism isn't very elegant, but what it buys us makes it worth it. Thanks to the namespacing, we only need to make sure that the IDs `"file"`, `"heading"`, and `"quote"` are *unique within this function*, rather than *unique across the entire app*.

Writing server functions

Now that we've got some UI, we can turn our attention to the server logic. The

server logic is encapsulated in a single function that we'll call the module server function.

Module server functions should be named like their corresponding module UI functions, but with a `server` suffix instead of a `Input` / `Output` / `UI` suffix. Since our UI function was called `csvFileUI`, we'll call our server function `csvFileServer`.

Inside of `csvFileServer`, there is a call to `moduleServer()`, to which two things are passed. One is the `id`, and the second is the **module function**:

```
# Module server function
csvFileServer <- function(id, stringsAsFactors) {
  moduleServer(
    id,
    ## Below is the module function
    function(input, output, session) {
      # The selected file, if any
      userFile <- reactive({
        # If no file is selected, don't do anything
        validate(need(input$file, message = FALSE))
        input$file
      })

      # The user's data, parsed into a data frame
      dataframe <- reactive({
        read.csv(userFile()$datapath,
          header = input$heading,
          quote = input$quote,
          stringsAsFactors = stringsAsFactors)
      })

      # We can run observers in here if we want to
      observe({
        msg <- sprintf("File %s was uploaded", userFile()$name)
        cat(msg, "\n")
      })

      # Return the reactive that yields the data frame
      return(dataframe)
    }
  )
}
```

The outer function, `csvFileServer()`, takes `id` as its first parameter. You can define the function so that it takes any number of additional parameters, including `...`, so that whoever uses the module can customize what the

module does. In this case, there's one extra parameter, `stringsAsFactors`, so the application that uses this module can decide whether or not to convert strings to factors when reading in the data.

Inside of `csvFileServer()` is a call to the `moduleServer()` function. This function is passed the `id` variable, as well as the **module function**. You may notice a lot of similarities between the module function and a regular Shiny server function. Its three parameters – `input`, `output`, and `session` – should be familiar: every module function must take those three parameters. The `moduleServer()` function invokes the module function in a special way that creates special `input`, `output`, and `session` objects that are aware of the `id`.

Inside of the module function, it can use parameters from the outside function. In this example, this outside function is `csvFileServer()`, and its `stringsAsFactors` parameter is used inside the module function. You can have as many or as few additional parameters as you want, including `...` if it makes sense, and you can use them for whatever you want inside the function body.

Inside the module function, we can use `input$file` to refer to the `ns("file")` component in the UI function. If this example had outputs, we could similarly match up `ns("plot")` with `output$plot`, for example. The `input`, `output`, and `session` objects we're provided with are special, in that they use the `id` to scope them to the specific namespace that matches up with our UI function.

On the flip side, the `input`, `output`, and `session` cannot be used to access inputs/outputs that are outside of the namespace, nor can they directly access reactive expressions and reactive values from elsewhere in the application.

These restrictions are by design, and they are important. The goal is not to prevent modules from interacting with their containing apps, but rather, to make these interactions *explicit*. **If a module needs to use a reactive expression, the outer function should take the reactive expression as a parameter. If a module wants to return reactive expressions to the calling app, then return a list of reactive expressions from the function.**

If a module needs to access an input that *isn't* part of the module, the containing app should pass the input value wrapped in a reactive expression (i.e. `reactive(...)`):

```
  myModule("myModule1", reactive(input$checkbox1))
```

Using modules

Assuming the above `csvFileUI` and `csvFileServer` functions are loaded (more on that in a moment), this is how you'd use them in a Shiny app:

```
library(shiny)

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      csvFileUI("datafile", "User data (.csv format)")
    ),
    mainPanel(
      dataTableOutput("table")
    )
  )
)

server <- function(input, output, session) {
  datafile <- csvFileServer("datafile", stringsAsFactors = FALSE)

  output$table <- renderDataTable({
    datafile()
  })
}

shinyApp(ui, server)
```

The UI function `csvFileUI` is called directly, using `"datafile"` as the `id`. In this case, we're inserting the generated UI into the sidebar.

The module server function is called with `csvFileServer()`, with the `id` that we will use as the namespace; this must be exactly the same as the `id` argument we passed to `csvFileUI`. The call to the module server function also is passed the parameter `stringsAsFactors = FALSE`.

Like all Shiny modules, `csvFileUI` can be embedded in a single app more than once. Each call must be passed a unique `id`, and each call must have a corresponding call to `csvFileServer()` on the server side with that same `id`.

Output example

Here's an example of a module that consists of two linked scatterplots

(selecting an area on one plot will highlight observations on both plots).

```
library(shiny)
library(ggplot2)
```

First we'll make the module UI function. We want two plots, `plot1` and `plot2`, side-by-side with a common brush ID of `brush`. (Notice that the brush ID needs to be wrapped in `ns()`, just like the plotOutput IDs.)

```
linkedScatterUI <- function(id) {
  ns <- NS(id)

  fluidRow(
    column(6, plotOutput(ns("plot1"), brush = ns("brush"))),
    column(6, plotOutput(ns("plot2"), brush = ns("brush")))
  )
}
```

The module server function comes next. Besides the mandatory `input`, `output`, and `session` parameters, we need to know the data frame to plot (`data`), and the column names that should be used as x and y for each plot (`left` and `right`).

To allow the data frame and columns to change in response to user actions, the `data`, `left`, and `right` must all be reactive expressions. These parameters are passed to `linkedScatterServer`, and they can be used in the module function defined inside.

```
linkedScatterServer <- function(id, data, left, right) {
  moduleServer(
    id,
    function(input, output, session) {
      # Yields the data frame with an additional column "selected_"
      # that indicates whether that observation is brushed
      dataWithSelection <- reactive({
        brushedPoints(data(), input$brush, allRows = TRUE)
      })

      output$plot1 <- renderPlot({
        scatterPlot(dataWithSelection(), left())
      })

      output$plot2 <- renderPlot({
        scatterPlot(dataWithSelection(), right())
      })

      return(dataWithSelection)
    }
  )
}
```

Notice that the module function inside of `linkedScatterServer()` returns the `dataWithSelection` reactive. This means that the caller of this module can make use of the brushed data as well, such as showing it in a table below the plots, for example.

For clarity and ease of testing, let's put the plotting code in a standalone function. The `scale_color_manual` call sets the colors of unselected vs. selected points, and `guide = FALSE` hides the legend.

```
scatterPlot <- function(data, cols) {
  ggplot(data, aes_string(x = cols[1], y = cols[2])) +
    geom_point(aes(color = selected_)) +
    scale_color_manual(values = c("black", "#66D65C"), guide =
FALSE)
}
```

To see this module in action, click here (http://shiny.rstudio.com/gallery/module-example.html).

Nesting modules

Modules can use other modules. When doing so, when the outer module's UI function calls the inner module's UI function, ensure that the `id` is wrapped

in `ns()`. In the following example, when `outerUI` calls `innerUI`, notice that the `id` argument is `ns("inner1")`:

```
innerUI <- function(id) {
  ns <- NS(id)
  "This is the inner UI"
}

outerUI <- function(id) {
  ns <- NS(id)
  wellPanel(
    innerUI(ns("inner1"))
  )
}
```

As for the module server functions, just ensure that the call to `callModule` for the inner module happens inside the outer module's server function. There's generally no need to use `ns()`.

```
innerServer <- function(id) {
  moduleServer(
    id,
    function(input, output, session) {
      # inner logic here
    }
  )
}

outerServer <- function(id) {
  moduleServer(
    id,
    function(input, output, session) {
      innerResult <- innerServer("inner1")
      # outer logic here
    }
  )
}
```

Using renderUI within modules

Inside of a module, you may want to use `uiOutput` / `renderUI`. If your `renderUI` block itself contains inputs/outputs, you need to use `ns()` to wrap your ID arguments, just like in the examples above. But those `ns` instances were created using `NS(id)`, and in this case, there's no `id` parameter to use. What to do?

The `session` parameter can provide the `ns` for you; just call `ns <- session$ns`. This will put the ID in the same namespace as the session.

```
columnChooserUI <- function(id) {
  ns <- NS(id)
  uiOutput(ns("controls"))
}

columnChooserServer <- function(id, data) {
  moduleServer(
    id,
    function(input, output, session) {
      output$controls <- renderUI({
        ns <- session$ns
        selectInput(ns("col"), "Columns", names(data), multiple =
TRUE)
      })

      return(reactive({
        validate(need(input$col, FALSE))
        data[,input$col]
      }))
    }
  )
}
```

Packaging modules

The previous examples of using a module assume that the module's UI and server functions are defined and available. But logistically, where should these functions actually be defined, and how should they be loaded into R?

There are several options.

Inline code

Most simply, you can put the UI and server function code directly in your app.

If you're using an app.R style file layout (both app UI and server logic in the same file), then you can just include the code for your module functions right in that file, before the app's UI and server logic.

If you're using a ui.R/server.R style file layout, add a global.R file to your app directory (if you don't already have one) and put the UI and server functions there. The global.R file will be loaded before either ui.R or server.R.

If you have many modules to define, or modules that contain a lot of code, this may result in a bloated global.R/app.R file.

In an R script in the R/ subdirectory

You can create a separate R script (.R file) for the module in the R/ subdirectory of your application. It will automatically be sourced (as of Shiny 1.5.0) when the application is loaded.

This is the recommended method for modules that won't be reused across applications.

In an R script elsewhere in the app directory

You can create a separate R script (.R file) for the module, either directly in the app directory or in a subdirectory. Then call `source("path-to-module.R")` from global.R (if using ui.R/server.R) or app.R. This will add your module functions to the global environment.

In versions of Shiny prior to 1.5.0, this was the recommended method, but with 1.5.0 and later, we recommend the previous method, where the standalone R script is in the R/ subdirectory.

R package

For modules that are intended for reuse across applications, consider building an R package. If you've never done this before, a good resource is Hadley Wickham's book R Packages (http://r-pkgs.had.co.nz/), which is freely available online.

Your R package needs to export and document your module's UI and server functions. You can include more than one module in a package, if you like.

Migrating from `callModule` to `moduleServer`

Prior to Shiny 1.5.0, the only way to use the module function was with `callModule`; as of Shiny 1.5.0, we added the `moduleServer` function and recommend using it instead of `callModule`, because the syntax for the user of the module is more consistent with the UI portion, and somewhat easier to understand. (Note that the UI portion of modules was unchanged.) New-style modules also can be tested with the `testServer` function, also introduced in Shiny 1.5.0.

Here is an example of the older-style module. The part to pay attention to is

the definition of `myModule`, and its corresponding use in the server function, with `callModule`:

```
# Module definition, old method
myModuleUI <- function(id, label = "Input text: ") {
  ns <- NS(id)
  tagList(
    textInput(ns("txt"), label),
    textOutput(ns("result"))
  )
}

myModule <- function(input, output, session, prefix = "") {
  output$result <- renderText({
    paste0(prefix, toupper(input$txt))
  })
}


# Use the module in an application
ui <- fluidPage(
  myModuleUI("myModule1")
)
server <- function(input, output, session) {
  callModule(myModule, "myModule1", prefix = "Converted to
uppercase: ")
}
shinyApp(ui, server)
```

Here is the same application, with the new method using `moduleServer()`. This time, we are creating a function called `myModuleServer`, and in the application's server function, we call that function directly (instead of using `callModule()`.

```
# Module definition, new method
myModuleUI <- function(id, label = "Input text: ") {
  ns <- NS(id)
  tagList(
    textInput(ns("txt"), label),
    textOutput(ns("result"))
  )
}

myModuleServer <- function(id, prefix = "") {
  moduleServer(
    id,
    function(input, output, session) {
      output$result <- renderText({
        paste0(prefix, toupper(input$txt))
      })
    }
  )
}


# Use the module in an application
ui <- fluidPage(
  myModuleUI("myModule1")
)
server <- function(input, output, session) {
  myModuleServer("myModule1", prefix = "Converted to uppercase: ")
}
shinyApp(ui, server)
```

The old and new versions of the application have the exact same behavior. Notice, however, that with the new version, the usage of the module in the application is more consistent, with `myModuleUI("myModule1")` and `myModuleServer("myModule1", prefix = "Converted to uppercase: ")`.

When creating the module server function, here is the old method:

```
# Old-style modules
myModule <- function(input, output, session, prefix = "") {
  output$result <- renderText({
    paste0(prefix, toupper(input$txt))
  })
}
```

And here is the new method:

```
# New-style modules
myModuleServer <- function(id, prefix = "") {
  moduleServer(
    id,
    function(input, output, session) {
      output$result <- renderText({
        paste0(prefix, toupper(input$txt))
      })
    }
  )
}
```

In the old version, the `myModule` function takes `input`, `output`, and `session`, as parameters, along with any additional user parameters – in this case, `prefix`. In the new version, the function just takes `id` and additional user parameters (`prefix`).

In the new version, there is an **inner** module function that takes `input`, `output`, and `session`, and no other parameters. The code from the old-style module is simply moved into this inner module function. Any extra user parameters, like `prefix` (or even `...`) can be accessed inside that function because they are available in the parent environment.

When it comes to **using** the module in the application's server function, here is the old method, which uses Shiny's `callModule` function:

```
# Old-style modules
server <- function(input, output, session) {
  callModule(myModule, "myModule1", prefix = "Converted to
uppercase: ")
}
```

The new method is more straightforward: it simply calls the `myModuleServer` function.

```
# New-style modules
server <- function(input, output, session) {
  myModuleServer("myModule1", prefix = "Converted to uppercase: ")
}
```