ZevRoss
KNOW YOUR DATA

Technical Tidbits From Spatial Analysis & Data Science

R powered web applications with Shiny (a tutorial and cheat sheet with 40 example apps)

Posted on April 19, 2016 by zev@zevross.com · 11 Comments

Shiny enables you to write powerful interactive web applications entirely in R. Using R you create a user interface and server and Shiny compiles your code into the HTML, CSS and JavaScript needed to display your application on the web. What makes a Shiny app particularly powerful is that it can execute R code on the backend so your app can perform any R calculation you can run on your desktop. Perhaps you want your app to slice and dice a dataset based on user inputs. Or maybe you want your web app to run linear models, GAMs or machine learning methods on user-selected data. In either case, Shiny can help.

Creating and running simple web applications is relatively easy and there are great resources for doing this. But when you want more control of the application functionality understanding the key concepts is challenging. To help you navigate the creation of satisfying Shiny applications we've assembled example code below that demonstrates some of the key concepts.

In order to run Shiny and follow the code on this post you should make sure you have RStudio software and the `shiny` R package. In creating this post I'm using version 0.13.1.9002. Additional details on package versions are at the end.

Shiny at its simplest

In its simplest form, a Shiny application requires a server function to do the calculations and a user interface. Below we have the simplest possible shiny app. We create an empty server, a UI with a basic message and then launch the app with the `shinyApp` function. Voila – a functioning web application created entirely in R!

For simplicity, we're creating our Shiny apps in this blog post as **single file apps**. In the past, Shiny required that you create two separate files (server.R and ui.R) but as of version 0.10.2 Shiny supports single file applications with both server and UI code in the same file. With bigger apps, of course, you will likely have far more files and want to use `ui.R` and `server.R` but for the mini-apps in this post we will create single page apps.

A second difference with earlier versions of `shiny` is that the user interface required using a ui handler function (`shinyUI`) but this is no longer required. Instead you would choose the function associated with the type of page you want to use as discussed in the layout guide. In relatively big apps, I use `fluidPage` but for most of this blog post we keep things simple with `basicPage`.

Depending on what you want to do with your app, the `session` argument is not required in the server but since you need it for some tasks a best practice would be to include it.

Back to top

# **Example app:** The simplest possible app

Not much here, but this is a real Shiny app.

Code 1 App 1

```
library(shiny)

server <- function(input, output, session) { } #the server

ui <- basicPage("This is a real Shiny app") # the user interface

shinyApp(ui = ui, server = server) # this launches your app
```

Set up your user interface (UI)

Layout your user interface

Under the hood, Shiny uses Twitter Bootstrap, a commonly used framework for designing webpages, to scaffold and provide some minimal styling. When you run the function `shinyApp()` your R code is essentially compiled to web-friendly languages HTML, JavaScript and CSS. You can take advantage of Shiny page template functions to layout your app or you can essentially design your app from scratch. There is also a brand new option discussed below to use what are called HTML templates but here we will focus on the most common (and simpler) approach.

Using predefined layout functions

As mentioned above, you will choose a layout function depending on the type of app you want. For this example, we will take advantage of a Shiny template using the function `sidebarLayout` which creates a page with a sidebar and a main panel. You can review other layout options in the RStudio layout guide. In this example we're creating a `sidebarPanel` and `mainPanel` and not much else.

Note that in a fluid design your sidebar and other elements may "collapse" if your browser view is not wide enough.

Back to top

# **Example app:** Basic user interface

Code 2 App 2

```
#### server
server <- function(input, output, session) {

}

#### user interface
ui <- fluidPage(

  titlePanel("App with simple layout"),

  sidebarLayout(

    sidebarPanel(
      "Sidebar"
    ), #endsidebarpanel

    mainPanel(
      "This is the main panel"
    )#end mainpanel
  )# end sidebarlayout
)
```

```
shinyApp(ui = ui, server = server)
```

Adding elements with tags$xxx

It's easy to add HTML elements to your Shiny app using HTML tags. There are more than 100 HTML elements you can add to your page using the syntax `tags$OBJECT()` as in `tags$p()` for a paragraph or `tags$blockquote()`. For a limited number of these tags, the most common ones, there are helper functions that allow you to skip the `tags$` so, for example, a first tier header can be created with `h1()` – but be careful, not all tags permit this.

For more detail on generating HTML I recommend you watch this video by Joe Cheng or review the HTML tags page on the RStudio website.

Back to top

# **Example app:** using HTML tags to layout your app

In this app we add HTML tags using, for example, `tags$blockquote` or, for common elements like `h1` we drop the `tags$`.

Code 3 App 3

```
server <- function(input, output, session) {

}

ui <- basicPage(
  h1("Title without tags$"),
  tags$blockquote("But block quote requires tags$ because it is less common than h3, h1 or code")
  h3("H3 is fine without tags and so is code here"),
  code("data.frame(a=1:10, b=1:10)")
)

shinyApp(ui = ui, server = server)
```

Layout your app using Bootstrap's grid system manually

If you're new to CSS creating your layout can be challenging and you would need to read the Bootstrap basics before getting started. When you're comfortable, though, you can use Bootstrap's grid system directly by specifying, rows and columns and column spans. As a general rule, all Bootstrap layouts are 12 columns across and these columns can be grouped to make wider columns.

In the example below, we have a layout with 2 rows. One is offset by one column and the other has a button (one column width) and text (6 column width).

Back to top

# **Example app:** Manual layout using Bootstrap's grid system

In this app we use Bootstrap elements like rows and columns to manually layout the app.

Code 4 App 4

```
server <- function(input, output, session) {

}

ui <- fluidPage(

  fluidRow(
    column(6, offset=1,
           h1("Title in one row")
    )

  ),
  fluidRow(
    column(
      actionButton("button", "Click")

    ),
    column(
      p("Row 2, Column 2 (button is col 1)")
    )

  )
)

shinyApp(ui = ui, server = server)
```

Style your user interface

Now that you have created your app layout following some of the approaches discussed above you are ready to style your app. There are several approaches to doing this from choosing existing themes (like adding a new, pre-defined skin) or styling yourself with CSS.

Use an existing "theme"

Bootstrap offers a number of pre-created themes that allow for a complete change of style with limited coding. The `shinythemes` package from RStudio allows you to take advantage of this capability. The only change you need to make to your app is to add a line of code `theme=shinytheme("cosmo")` to your `fluidPage` or `fixedPage` function and the entire look of you app will change (you cannot apply themes to the `basicPage` because `basicPage` is not based on Bootstrap).

In this particular example, you won't see much difference because the app is so simple, but with bigger apps the changes are significant and you can see examples in the Rstudio documentation for shinythemes or in the free themes page.

Back to top

# **Example app:** Use a pre-existing Bootstrap theme

Note that because this blog post is not very wide and we're using a fluid page, the main panel has collapsed below the sidebar.

Code 5 App 5

```
library(shinythemes)

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

}

ui <- fluidPage(theme=shinytheme("cosmo"),

  titlePanel("Use an existing theme"),

  sidebarLayout(

    sidebarPanel(
      h3("Note the button is black. This is different from the previous app."),
      actionButton("button", "A button")
    ),
    mainPanel(
      tabsetPanel(
        tabPanel("Plot"),
        tabPanel("Summary"),
        tabPanel("Table")
      )
    )
  )
)

shinyApp(ui = ui, server = server)
```

Style yourself with CSS

If you want to use your own CSS you have several options. For the sake of completeness, I will cover them all but the best practice would be to include all of your CSS in an external stylesheet (let's say it's called `style.css`). If you're using a two page app (with a ui.R and server.R file) you can add a www folder at the same level as the UI and server files with your style sheet and this will be read automatically. Otherwise you can read your stylesheet with `includeCSS`. Here is your full list of options.

1. Include your raw CSS directly inline in the head of your app
2. Include CSS within specific HTML tags
3. With a two page app (ui.R and server.R) you can add a folder called www and the app will automatically read any CSS files you've included there.
4. The `shinyjs` package (discussed below) also has a useful function called `inlineCSS` that you can use to add CSS
5. Use the `includeCSS` function to read an external CSS file

As I mention above, best practice for larger apps would be to include all your CSS in a single, external file so that it's easy to find and change settings. For smaller apps including CSS in the head or HTML tags would be acceptable but be careful, if you include styling in the HTML tags it can be difficult to prioritize and organize your styles.

For more detail, see RStudio's page on CSS.

Back to top

# **Example app:** using inline CSS in the HTML head

Note that the actual style code is within an `HTML` function – this prevents Shiny from treating the text as "regular" text and escaping the strings.

Code 6 App 6

```
server <- function(input, output, session) {

}

ui <- basicPage(
  # this is your web page header information
  tags$head(
    # here you include your inline styles
    tags$style(HTML("

      body {
        background-color: cornflowerblue;
        color: #6B1413;
      }

    "))
  ),

  h3("CSS using the HTML tag"),
  p("Some important text")

)

shinyApp(ui = ui, server = server)
```

Back to top

# **Example app:** using `includeCSS` (assumes you have an external file called `style.css`)

In this case my `style.css` file only has one bit of CSS `body{background-color: Maroon;}`.

Code 7 App 7

```
server <- function(input, output, session) {

}

ui <- basicPage(

includeCSS("style.css"),

  h2("CSS by reading an external page"),
  p("Some important text")

)


shinyApp(ui = ui, server = server)
```

Back to top

# Example app: including a style in an individual HTML tag

Usually, this is not a great idea because it's hard to find your styles and it's even harder to be consistent with style. But for small apps it's fine.

Code 8 App 8

```
server <- function(input, output, session) {

}

ui <- basicPage(

  # here we style inline
  h2(style = "color:green; font-family:'Comic Sans MS'",
     "Styles within an HTML tag (not recommended for big apps)"),
  p("Some important text")

)


shinyApp(ui = ui, server = server)
```

Back to top

# Example app: An example of using the `shinyjs` `inlineCSS` function

The `shinyjs` package has a lot of nice add-on functionality and includes the `inlineCSS` function to make it a little easier to include CSS using a syntax that is more R-like. Note that instead of `body{color:DarkBlue}`, which is CSS, we have `list(body = "color:DarkBlue")` which is R code. This will make all the text blue.

Code 9 App 9

```
library(shinyjs)

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

}

ui <- fluidPage(

  # This adds the CSS to the file
  shinyjs::inlineCSS(list(body = "color:DarkBlue")),
  titlePanel("Use the shinyjs package to add styles"),

  sidebarLayout(

    sidebarPanel(
      h3("Sidebar title")
    ),

    mainPanel(
        "Body text"
    )
  )
)

shinyApp(ui = ui, server = server)
```

Add widgets to your user interface (text boxes, sliders etc)

At this point we have scaffolded and added styles to our UI but have not added any elements that will allow our user interface to be interactive. Shiny has a wide array of input widgets (e.g., pull-down menus, checkboxes) that we can use to add this functionality. Take a look at the RStudio widget gallery for a complete list.

For this particular example, I'll want a slider range and text input. Note that both of these, and all widgets, have a unique input id (`inputId`) – the text is `mytext` and the slider is `myslider`. Careful, this often causes trouble – the **input ids must be unique**.

We're still only focusing here on the UI so these widgets "work" but don't actually do anything because we have not included any code in the Shiny server. In this particular example, we are including both a slider and a text input.

Back to top

# Example app: Allow user input

Simple app with widgets, though the widgets are not connected to the server yet. You can enter text and play with the slider but nothing will happen.

Code 10 App 10

```
server <- function(input, output, session) {

}

ui <- basicPage(

  h3("A couple of simple widgets that don't do anything yet"),
  # a slider
  sliderInput(inputId = "myslider", label = "Limit the ", min = 0,
              max = 100, value = c(40, 60)),
  # a text input box
  textInput(inputId = "mytext", label = "Text input", value = "Enter text...")
)



shinyApp(ui = ui, server = server)
```

Set up your server

Input from your user interface

Currently, our users can interact with our application all they want but our server is deaf and dumb. There is no code in the server to tell it to listen or react.

In order to listen, it needs to know what to listen to and this is where our unique input ids come in. Our widgets all have a unique ID that the server will listen for and react to. Each of these input ids is mapped to the `input` argument on our server. So, for example, if you want to access the value of the text box we created (which is called `mytext`) from within the server you would call `input$mytext`. Likewise, for the slider the values are in `input$myslider`. These objects from the UI (`input$myslider`, `input$mytext`) are called reactive values.

But… and this is one of the most challenging concepts in Shiny, you can't simply refer to `input$mytext` in the server to get the current value of the text box. Instead **you need to wrap these reactive values in one of the functions designed to handle interactive widget output**. The functions that can handle the reactive values are `observe`, `reactive` and the suite of `render*` functions all of which are discussed below.

Listen for changes with `observe` or `reactive`, the Shiny hearing aids

The values associated with your UI inputs (like the text box, `input$mytext`) are called reactive values. It is tempting to include `input$mytext` directly in the server. Try running the code below in your own console and you will get an error `operation not allowed without an active reactive context`. This means that to read the reactive value you need to wrap it in a function designed to listen to the reactive elements. Note that if you run this code locally (and get the error) you'll need to click on the "stop" button in the console to stop the app.

Back to top

# **Example app:** Careful, you can't include bare reactive values in the server

This will throw an error because the reactive value is not being read by a function designed to handle it! Instead we need to wrap the values in an `observe`, `reactive` or `render*` function that is designed to handle reactive content. In a console, you will see the error. Here the app will fail to load.

Code 11   App 11 (this will fail to load properly)

```
server <- function(input, output, session) {
  # this will NOT work!!
  print(input$mytext)
}

ui <- basicPage(

  h3("A couple of simple widgets that don't do anything yet"),
  # a slider
  sliderInput(inputId = "myslider", label = "Limit the ", min = 0,
              max = 100, value = c(40, 60)),
  # a text input box
  textInput(inputId = "mytext", label = "Text input", value = "Enter text...")
)

shinyApp(ui = ui, server = server)
```

Before going any further let me introduce the `update*` functions

In the example code above (which doesn't work) you'll notice that I used a `print` statement in the server. This keeps things simple in the example apps but if you're not running these mini-apps locally you won't see what's going on because `print` simply prints to the console and you can't see the result in the online app. Since I want you to see reactions even if you're not running the apps locally, I need to introduce one concept a little out of order.

In particular, I want to introduce you to the suite of `update*` functions that are designed to allow you to update existing Shiny widgets – for example, update a text box or the list of items in a pulldown menu. In the next few apps I will be using a function called `updateTextInput` instead of `print` – essentially I will use a text box as a console and will print results to the text box with `updateTextInput`. Here is an example:

Back to top

# **Example app:** Using `updateTextInput` as a "console" for this blog post

In the app below you can see that I'm "printing" the text typed into `mytext` to `myresults` with the function `updateTextInput`.

Code 12 App 12

```
server <- function(input, output, session) {

  observe({
    txt <- paste("Value above is:", input$mytext)

    # here I'm essentially writing a result to the text box
    # called myresults
    updateTextInput(session, "myresults", value=txt)
  })

}

ui <-   basicPage(
  h3("An example of an update* function"),
  textInput("mytext", "Input goes here"),
  textInput("myresults", "Results will be printed here", "Initial value")
)

shinyApp(ui = ui, server = server)
```

Back to functions to read reactive values: featuring `observe` to generate side effects and no output

Observers will get triggered in response to reactive values. They were designed to listen to reactive elements and respond by causing side effects, like updates to text boxes or pull-downs. Unlike the `reactive` function, which we cover next, they should not be used to return data or values.

There are two flavors of `observe`. With `observe` the code inside will get triggered when any of the reactive values inside change. With `observeEvent` code will only be triggered by specified reactive values. I would suggest that you use `observeEvent` whenever possible because `observeEvent` forces you to think through and specify the reactions you want to see.

So, back to the example from above. Instead of including `input$mytext` alone in the server (which causes an error) we can put the `input$mytext` in an observer and use it to update `myresults`. Any time the user makes a change to `mytext` the observer code will run and `myresults` will be updated.

Back to top

# **Example app:** Use `observe` to react and cause side effects

We use the `observe` function to listen for changes to the reactive value associated with the input text box (`input$mytext`). When `input$mytext` changes the observer code runs and updates `myresults`.

Code 13 App 13

```
server <- function(input, output, session) {
  # this will work, the reactive element is wrapped in an observer
  # it prints the value to the to the results text box
  observe({
    updateTextInput(session, inputId = "myresults", value = input$mytext)

  })
}



ui <-   basicPage(
  h3("The value in the text box gets printed to the results text box."),
  textInput("mytext", "Input goes here"),
  textInput("myresults", "Results will be printed here", "Initial value")
)

shinyApp(ui = ui, server = server)
```

`observeEvent`: use `observeEvent` to prevent unwanted reactions

As mentioned above, the code in an `observe` will run if *any* of the reactives inside change. So in the following code the results text box will update if the user interacts with the input text box **or** with the slider. Even though the slider reactive `input$myslider` is not actually being used in any way. Any reactive value in the `observe` function will trigger all the code in the observe function to run.

Back to top

# **Example app:** All the reactive values in `observe` will trigger the code to run (even if that's not what you want)

Since `input$myslider` and `input$mytext` are in the observer if the user makes a change to *either* the observer code will run.

Note that in the next couple of apps I paste a random number to the text to make it easier to see updates.

Code 14 App 14

```
server <- function(input, output, session) {

  observe({
    # even though the slider is not involved in a calculation, if
    # you change the slider it will run all this code and update the text box
    # changes to the mytext box also will trigger the code to run
    input$myslider
    txt <- paste(input$mytext, sample(1:10000, 1))
    updateTextInput(session, inputId = "myresults", value = txt)

  })

}

ui <- basicPage(
  h3("The results text box gets updated if you change the other text box OR the slider."),
  sliderInput("myslider", "A slider:", min=0, max=1000, value=500),
  textInput("mytext", "Input goes here", value = "Initial value"),
  textInput("myresults", "Results will be printed here")
)

shinyApp(ui = ui, server = server)
```

The `observeEvent` function is designed to address this issue. With `observeEvent` the code inside will only run if the specified reactive value(s) change. So in the following code the update will only execute if the user makes changes to the text box.

Back to top

# **Example app:** Only specified reactive(s) trigger the code to run using `observeEvent`.

Using `observeEvent` instead of `observe` allows you to specify the reactive values to listen for and react to. In this case, we're only listening for one reactive (`input$mytext`) but you can include more than one.

Code 15 App 15

```
server <- function(input, output, session) {

  # Using observeEvent we're telling Shiny only to run this code
  # if mytext gets updated.
  observeEvent(input$mytext, {

    # myslider is a reactive but it does not trigger the code to
    # run here because we're using observeEvent and only specified
    # input$mytext
    input$myslider
    txt <- paste(input$mytext, sample(1:10000, 1))
    updateTextInput(session, inputId = "myresults", value = txt)

  })
}

ui <- basicPage(

  h3("The results text box only updates when you change the top text box (slider interactions do
  sliderInput("myslider", "A slider:", min=0, max=1000, value=500),
  textInput("mytext", "Input goes here", value = "Initial value"),
  textInput("myresults", "Results will be printed here")

)

shinyApp(ui = ui, server = server)
```

Observer priority: which observers run first

With bigger apps, you may have situations where you want one observer to run before others. You can use an `observe` function priority argument to do this. The default priority is 0 and higher numbers mean higher priority (and you can use negative numbers).

Back to top

# **Example app:** No priority specified

The order of execution is not always predictable in Shiny apps. In this example we have two observers and they **both write to the same output text box**. This app uses default priorities and the second observer will run second and will, therefore, write over the updates from the first observer.

Code 16 App 16

```
server <- function(input, output, session) {

  # With no priority specified the second observer will
  # run second and overwrite the first observer
  observe({
    txtA <- paste("First observer", input$mytext)
    updateTextInput(session, inputId = "myresults", value = txtA)
  })

  observe({
    txtB <- paste("Second observer", input$mytext)
    updateTextInput(session, inputId = "myresults", value = txtB)
  })


}

ui <- basicPage(
  h3("Second observer runs second so it overwrites the first observer"),
  textInput("mytext", "Input goes here"),
  textInput("myresults", "Results will be printed here", "Initial value")
)

shinyApp(ui = ui, server = server)
```

Back to top

# **Example app:** Prioritize to control order of execution

Instead of default priorities we will force the first observer to run second so that it writes over the updates from the second observer. In order to do this we have a higher priority for the second observer so that it runs first.

Code 17 App 17

```
server <- function(input, output, session) {

  # first observer has lower priority so it runs second and will
  # overwrite the other observer
  observe({
    txtA <- paste("First observer", input$mytext)
    updateTextInput(session, inputId = "myresults", value = txtA)
  }, priority = 1)

  # second observer has higher priority so it will run first and
  # then be overwritten
  observe({
    txtB <- paste("Second observer", input$mytext)
    updateTextInput(session, inputId = "myresults", value = txtB)
  }, priority = 2)

}

ui <- basicPage(

  h3("Priority is higher for second observer so it runs first and then gets written over by the f
  textInput("mytext", "Input goes here"),
  textInput("myresults", "Results will be printed here", "")
)

shinyApp(ui = ui, server = server)
```

`reactive`: use a `reactive` function to isolate code and generate output with no side effects

A `reactive` function is used in the same way you would use an R function except that it gets triggered by a reactive element. Because using `reactive` creates a function and returns results you generally save a reactive as an object and use it elsewhere in your server as you would use any R function. There is one major distinction from a function, however, the function can only be executed within a “reactive context” (so in another reactive, an observe or a render* function). The reactives are NOT supposed to generate side effects, they should essentially be self-contained.

So in the example, below I use `reactive` to create a self-contained function called `myresults`. Since I want to print the results to the console (a side effect), I run the reactive function from within an observer. Note that since it's a function I'm using `myresults()` rather than just `myresults`.

Back to top

# Example app: Using `reactive` to generate output

In this app we use `reactive` to create a function that listens for the `input$mytext` reactive value. When it hears a change it generates a string as output.

Code 18 App 18

```
server <- function(input, output, session) {

  # this is my reactive function -- I'm using it to
  # isolate reactions related to the text box
  mystring <- reactive({
    paste(input$mytext, " is what the user types")
  })

  observe({
    # The reactive will run each time the textbox changes and
    # print results to the console.
    txt <- mystring()
    updateTextInput(session, inputId = "myresults", value = txt)
  })

}

ui <- basicPage(

    h3("The reactive generates a string output which is added to the results text box"),
    textInput("mytext", "Input goes here"),
    textInput("myresults", "Results will be printed here", "")

)

shinyApp(ui = ui, server = server)
```

`eventReactive`: used to prevent unwanted reactions in a `reactive` function

Similar to the function `observeEvent` which allows you to specify which reactive values trigger a code run, `eventReactive` can do the same for reactive functions.

Back to top

# **Example app:** Using `eventReactive` to prevent unwanted reactions

Sometimes you only want your reactive function to listen for specific reactive values and this is when you use `eventReactive` or `observeEvent`. In this app we have a `reactive` function that responds to both reactive values and an `eventReactive` that only reacts to changes in `input$mytext`.

Code 19 App 19

```
server <- function(input, output, session) {

  # since both mytext and myslider are in the reactive
  # they both trigger the code to run
  myresults <- reactive({
    paste(input$mytext, input$myslider)
  })

  # eventReactive here tells Shiny only to trigger this code
  # when mytext changes
  myresults_lim <- eventReactive(input$mytext, {
    paste(input$mytext, input$myslider)
  })

  observe(updateTextInput(session, "myresults", value = myresults()))
  observe(updateTextInput(session, "myresults_lim", value = myresults_lim()))


}

ui <- basicPage(

    h3("Using eventReactive to limit reactions."),
    sliderInput("myslider", "", min=0, max=1000, value=500),
    textInput("mytext", "Input goes here"),
    textInput("myresults", "Text box + slider (updates when either changes)", "Initial value"),
    textInput("myresults_lim", "Text box + slider (updates when text box changes)", "Initial valu

)

shinyApp(ui = ui, server = server)
```

Understanding and controlling when reactive functions get triggered

In our examples above, we included a reactive value (`input$mytext`) in `observe` or `reactive`. Each time the user changes the text value the server updates the text box. No problem. These rules also apply to the family of `render*` functions which we discuss below.

But, and here is a mini test,:

- What happens if you have no reactive value in an `observe` function?
- What happens if your reactive value *is* in the `observe` function but is not involved in any calculations?

If there is no reactive value the code will run once and never again

If the `observe` (or `reactive` or `render*`) function has no reactive value inside it, it will run when you launch the app but never again. So in this server code the values 1-10 will print to the console once and then never again. That's it.

```
server <- function(input, output, session) {
  observe({
    # will only run when the app loads, never again
    print(1:10)
    })
}
```

If there is a reactive value the code will run on app load and then on each change

Conversely, if you include a reactive value in an `observe` (or `reactive` or `render`) function the code inside will be triggered on app load and then every time the reactive value changes even if that reactive value is not used in any calculations. So in the code below you can see that `input$mytext` is not actually part of the print statement – it's not used in any calculations. Nevertheless, any time your user changes the text box this server will print 1 to 10 to the console. This is a very important concept – **a reactive value in your `observe` or `reactive` functions will trigger that function to run if the user interacts with it even if the reactive element is not part of the calculations.**

```
server <- function(input, output, session) {
  observe({
    input$mytext
    # this will run on app load and then again every time "mytext" changes
    print(1:10)
    })
}
```

Since reactive values trigger reactive functions you should keep reactive values separated whenever possible

As we discussed above, because any reactive value in a reactive function will trigger code to run you want to be careful to modularize your code as much as possible. In other words, keep calculations separated as much as possible. Reactive functions are a good way to do this. You can use reactive functions to isolate code and only run that code when necessary. Take, for example, the following app. I have included the reactive associated with both the slider and the text input in the same observer. As a result, even if the user only changes the slider, **all the code in the observer will get run**, even the pieces associated with the text box.

Back to top

# **Example app:** Reactive values are **not** kept separate (less good)

Both reactive values (`input$mytext` and `input$myslider`) are in the same observer meaning that all the code in that observer runs when either changes.

Code 20 App 20

```
#### server
server <- function(input, output, session) {

  # Notice that even if you only change the text box that the
  # slider code also runs and gets changed. The reverse is also
  # true. You might want to isolate these pieces.
  observe({
    txt <- paste(input$mytext, sample(1:100, 1))
    val <- paste(input$myslider,  sample(1:100, 1), sep="-")

    res <- paste0(txt, " | Slider ", val)
    updateTextInput(session, "myresults", value = res)
  })
}



ui <- basicPage(

  h3("Change to text OR slider changes both parts of results text box"),
  sliderInput("myslider", "A slider:", min=0, max=1000, value=500),
  textInput("mytext", "Input goes here", "Text"),
  textInput("myresults", "Results will be printed here", "Initial value")

)

shinyApp(ui = ui, server = server)
```

Back to top

# **Example app:** Reactive values are kept separate (better)

I refactored the code above so that the slider and text reactive values are in separate reactive functions. This way if the text reactive changes only the relevant code gets run. Likewise for the slider.

Code 21 App 21

```
server <- function(input, output, session) {

  # Now if you change the slider only the slider result changes
  # and the text box result stays the same. This is because
  # we isolated the reactive values in their own reactive function

  txt <- reactive({paste(input$mytext, sample(1:100, 1))})
  val <- reactive({paste(input$myslider, sample(1:100, 1), sep="-")})


  observe({
    res <- paste0(txt(), " | Slider ", val())
    updateTextInput(session, "myresults", value = res)
  })


}

ui <- basicPage(

    h3("Changes to the text box and slider are separated so that a change to the text box will no
    sliderInput("myslider", "A slider:", min=0, max=1000, value=500),
    textInput("mytext", "Input goes here", "Text"),
    textInput("myresults", "Results will be printed here", "Initial value")

)

shinyApp(ui = ui, server = server)
```

`isolate`: An alternative to `observeEvent` or `eventReactive`

Above you saw that `observeEvent` and `eventReactive` can limit reactions to specified reactive values. An alternative to using those functions is to use `isolate` in `observe`, `reactive` or `render*`. Generally, I'd argue in favor of using `observeEvent` and `eventReactive` because `isolate` will be harder to find in your code, but there are times when `isolate` can be useful.

Back to top

# **Example app:** Use isolate to avoid triggering reactions

In this example we have two text boxes that update with the input text. One updates constantly and in the other we've used `isolate` to ensure it only gets triggered with a button click.

Code 22 App 22

```r
server <- function(input, output, session) {

  observe({
    updateTextInput(session, "text_output1", value = input$text_input)
  })


  # instead of observe and isolate, you could instead use observeEvent
  observe({
    input$updateButton
    updateTextInput(session, "text_output2", value = isolate(input$text_input))
  })

}


ui <- basicPage(

    h3("The value in the text box gets printed to the results text box."),
    textInput("text_input", "Type here"),
    textInput("text_output1", "This box is constantly updating"),

    textInput("text_output2", "Updates only with action button click"),
    actionButton("updateButton", "Update list")

)

shinyApp(ui = ui, server = server)
```

Link your user interface with the server to display text, tables and plots (`render*` and `*Output`)

The third type of listener (after `observe` and `reactive`) is designed to create data-related items to add to your user interface. In the previous mini-apps we listened and reacted to reactive values with `observe` or `reactive` but simply updated a text box. Not that exciting. The functions like `renderText` or `renderPlot` and their UI counterparts, `textOutput` and `plotOutput`, enable you to create meaningful output.

An initial example with `renderText` and `textOutput`

In order to return values to the user we need a strategy to (1) grab needed values from the UI in the server; (2) process as necessary and then (3) return the result to the UI. To do this RStudio created a suite of functions that tag-team and circulate the value to and from the server and UI. These are the `render*` functions and the `*Output` functions.

The User-to-Server-back-to-User process in broad strokes:

1. The `renderText` function, in the server, would be used to read the text box reactive value (and process as necessary).
2. Then the `renderText` function would send the result back to the Shiny UI by attaching the results to the `output` object.
3. Shiny's UI would read the output from the server and deliver to the user using the `textOutput` function.

The User-to-Server-back-to-User process in detail:

Using our own app as an example. Let's say that we want to print the value from a text box to the UI. In order to do this we need to "complete the loop" with our UI which amounts to this:

1. Our UI has a text input box called `mytext`.
2. Anytime the text box changes it triggers the `renderText` function in the server (this is because `input$mytext`, a reactive, is in the `renderText` function)
3. All the code in `renderText` runs each time the text box changes
4. The results from `renderText` are attached to an output list, in this case we've named it the output `my_output_text`.
5. Since the `my_output_text` object was generated by the server's `renderText` function we need to use the UI's counterpart function `textOutput` to handle it and send it to the user. Note that unlike reactives in the server that are identified as `input$mytext` the output is named in quotes (so we use `"my_output_text"` rather than `output$my_output_text`).

Back to top

# **Example app:** Using `renderText` and `textOutput` to add to your UI

This app uses `renderText` and `textOutput` to demonstrate how to print text to the UI.

Code 23 App 23

```
server <- function(input, output, session) {

  # input$mytext comes from the UI. my_output_text gets
  # sent back
  output$my_output_text <- renderText({
    return(input$mytext)
    })
}

ui <- basicPage(
    h3("The value in the text box gets printed below with the textOutput function."),
    textInput("mytext", "Input goes here"),

    # my_output_text comes from the server
    "Your value is:",
    textOutput("my_output_text")
)

shinyApp(ui = ui, server = server)
```

Adding a plot with `renderPlot` and `plotOutput`

Similar to the `renderText` and `textOutput`, the `renderPlot` and `plotOutput` functions will create an object, in this case a plot, and then add that object to the user interface. Also similar to the text example, the reactive values move from the UI to the server and back to the UI.

Back to top

# **Example app:** Add a plot to your UI

Code 24 App 24

```
server <- function(input, output, session) {

  output$my_output_text <- renderText({
    init <- "Your value is: "
    return(paste0(init, input$mytext))
  })

  # send plot to the ui as my_output_plot
  output$my_output_plot <- renderPlot({
    plot(1:10, 1:10, pch=16, col=1:10, cex=1:10, main = input$mytext)
  })


}

ui <- basicPage(

  h3("Now we have both text and plot output"),
  textInput("mytext", "Input goes here"),

  # my_output_text comes from the server
  textOutput("my_output_text"),
  # my_output_plot comes from the server
  plotOutput("my_output_plot")

)

shinyApp(ui = ui, server = server)
```

Dynamic UI with `renderUI` and `outputUI`

If you want to return a block of UI elements – say a paragraph, then a text box and a selector – you would use `renderUI` and `outputUI` and instead of returning one object, you return a `list` or `tagList` of objects (these functions can be used interchangeably).

Back to top

# **Example app:** Dynamic user interface

Instead of returning a single object, we are returning a list of objects. Specifically, our `my_output_UI` output is a list of an `h4` header and a `selector`. The selector gets updated when the user clicks on the button (note `observeEvent`). Each time the user clicks the button the `selections` gets updated.

Code 25 App 25

```
server <- function(input, output, session) {

  # return a list of UI elements
  output$my_output_UI <- renderUI({

    list(
      h4(style = "color:blue;", "My selection list"),
      selectInput(inputId = "myselect", label="", choices = selections)
    )
  })

  # initial selections
  selections <- c("New York", "Philadelphia")

  # use observe event to notice when the user clicks the button
  # update the selection list. Note the double assignment <<-
  observeEvent(input$mybutton,{
    selections <<- c(input$mytext, selections)
    updateSelectInput(session, "myselect", choices = selections, selected = selections[1])
  })

}

ui <- basicPage(

    h3("Using renderUI and uiOutput"),
    uiOutput("my_output_UI"),
    textInput("mytext", ""),
    actionButton("mybutton", "Click to add to selections")

)

shinyApp(ui = ui, server = server)
```

Putting the three listeners, `reactive`, `observer` and `render` together in your server

In the example below I'm including one typical use case for each of the three listeners. In a later example, I will link them together in a more meaningful way but there is one more topic I want to cover before I do that.

Back to top

# Example app: `observe`, `reactive` and `render*` in one app

The `reactive` function generates data (an output but no UI side effects). The `observe` is designed to update the text box (a side effect, but does not produce output). The `renderTable` function returns the table to the UI.

Code 26 App 26

```
server <- function(input, output, session) {

  # when the slider changes update the text box
  observe({
    updateTextInput(session, "mytext", value=input$myslider)
  })

  # when the slider changes update the dataset
  dat <- reactive({
    input$myslider
    cars[1:input$myslider,]

  })

  # Since dat() is generated from a reactive that
  # is triggered by input$myslider this table will update
  # any time that input$myslider updates
  output$mytable <- renderTable({
    dat()
  })

}

ui <- fluidPage(

  titlePanel("An app using an observe, reactive and render"),

  sidebarLayout(
    sidebarPanel(
      sliderInput("myslider", "Number of rows to display", min=1, max=50, value=5),
      textInput(inputId = "mytext", label = "Slider value")
    ), # end sidebar panel

    mainPanel(
      tabsetPanel(
        tabPanel("Table", tableOutput("mytable"))
      )
    ) # end main panel
  )
)
```

```
shinyApp(ui = ui, server = server)
```

Deploy your app

Although Shiny apps end up as HTML files you can't simply copy them to your server. They require a Shiny server, a server that can run your R commands.

Running locally

- If you're developing in RStudio, and have a multi-page app you can open either ui.R or server.R and click on Run App in the top right.
- For a single or multi-page app you can use the function `runApp` where you specify the directory your app.R or ui.R/server.R files are housed in.
- For a single page app you can use the function `shinyApp` where you specify the ui and the server as I've done in the examples above.

Running on shinyapps.io

This site is managed by RStudio and is free for small apps with limited visits and scales up in paid versions.

Running your own Shiny server

There is a free, open source version of the Shiny server that you can run on, for example, Amazon Web Services or your own server. This is designed for apps with a relatively low number of visitors.

Running Shiny Server Pro

RStudio also sells a yearly subscription to Shiny Server Pro that provides security, admin and other enhancements when compared to the open source version. You can view a comparison of the open source and pro version here.

Add-on packages

shinyjs

The `shinyjs` package, created by Dean Attali, allows you to use common JavaScript operations in your Shiny applications such as hiding an element, delaying code etc. The package provides more than a dozen useful functions that are described in a page on GitHub. His talk on the package at the 2016 Shiny Developers Conference is also worth watching and will be posted by RStudio in the near future.

In order to use the functionality you need to load the package and then activate it in the UI with the `useShinyjs` function.

Back to top

# Example app: `shinyjs` in action, try the toggle

This is just a tiny example of using the package to run JavaScript code.

Code 27 App 27

```
library(shinyjs)

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

  observeEvent(input$button, {
    toggle("myslider")

  })
}

ui <- fluidPage(

  useShinyjs(),
    h3("A quick demonstration of shinyjs functionality"),

    actionButton("button", "Toggle slider"),
    sliderInput("myslider", "A slider:", min=0, max=1000, value=500)
)


shinyApp(ui = ui, server = server)
```

shinyURL

The `shinyURL` package, created by Andrzej Oleś, allows you to save and restore the state of your shiny apps if you want to share not just the app, but a particular page in the app. See the GitHub page for more details. His talk on the package at the 2016 Shiny Developers Conference is also worth watching.

shinyBS

The shinyBS package, created by Eric Bailey, offers some nice additional Bootstrap-related components like modals, collapses and related. Unfortunately, it's not clear that it's being maintained (last commit in April 2015) and many of the components do not work with the most recent versions of Shiny. The functions can still be useful to help guide the creation of updated versions of the components. See, for example, this discussion where the modal function is altered to work with current Shiny. For now you might hold off on using this one.

Advanced topics

Create re-useable UI elements

What if you have an app that puts the same element in multiple tabs. Perhaps you have a variable select box that needs to be on 10 different tabs. You can create it once and then copy and paste it onto the other tabs but this presents several issues:

1. What if you want to make changes – you'd need to make 10 changes.
2. Shiny requires that your elements all have unique IDs so you'd need to come up with a naming scheme. Perhaps select1, select2 etc…
3. Copy paste is a pain and prone to error!

As an alternative, you can create a function that outputs the UI elements you need and then, instead of repeating the code that creates those elements you can simply call the function. So in the example below, I create a function to generate a title, select box, radio buttons and a check box. Then I call that function three times – one each for the tabs.

(Note that in the newest versions of Shiny there is an alternative approach, modules, discussed below).

Back to top

# **Example app:** Create re-useable UI elements

The `createSelectRadio` function, a function that we create in this app, takes an id as input and then creates and outputs a list of components. The new ID for each component is the component name and then, as a suffix, the id provided when running the function.

Code 28 App 28

```
server <- function(input, output, session) {
  # A function to create a block of UI elements this
  # function can be re-used to create similar blocks
  # of ui elements in different places
  createSelectRadio <- function(id, title){

    selectID <- paste0("myselect", id)
    radioID <- paste0("myradio", id)
    checkID <- paste0("mycheck", id)

    res <- list(
      h2(title),
      selectInput(inputId = selectID, label="", choices = sample(LETTERS, 3)),
      radioButtons(inputId = radioID, label="", choices = sample(letters,3)),
      checkboxInput(inputId = checkID, label="", value=TRUE)
    )

    return(res)
  }

  # here we create our blocks of UI elements by running the function
  output$forPlot    <- renderUI({createSelectRadio(1, "In plot tab")})
  output$forSummary <- renderUI({createSelectRadio(2, "In summary tab")})
  output$forTable   <- renderUI({createSelectRadio(3, "In table tab")})

}

ui <- basicPage(

  h3("All tabs have the same set of components created with a function."),

  tabsetPanel(
    tabPanel("Summary", uiOutput("forSummary")),
    tabPanel("Plot",    uiOutput("forPlot")),
    tabPanel("Table",    uiOutput("forTable"))
  )

)


shinyApp(ui = ui, server = server)
```

Interactive data tables (careful, there are two flavors)

Adding interactive tables to your Shiny app is easier than you think. The original version of the shiny package included a `renderDataTable` function but this is slowly being deprecated in favor of a new `renderDataTable` function from the package `DT`. For the purposes of simply showing a table, the two options are very similar but DT extends the styling and interaction capabilities. The `DT` package, created by Yihui Xie, has some pretty amazing functionality – take a look at the documentation.

Keep in mind that the original `renderDataTable` function and the version in the package `DT` have slightly different syntax which can be confusing when you're looking for example code. I recommend you only use code where `DT` is explicitly loaded (after the `shiny` package) or you see the syntax `DT::renderDataTable` which specifies which version of the function to use.

Back to top

# **Example app:** Using the `shiny` package for an interactive table

This app uses the `renderDataTable` function from Shiny.

Code 29 App 29

```r
server <- function(input, output, session) {

  observe({
    updateTextInput(session, "mytext", value=input$myslider)
  })

  dat <- reactive({
    input$myslider
    mtcars[1:input$myslider, c("mpg", "cyl", "disp")]

  })

  output$mytable <- shiny::renderDataTable(dat())
}

ui <- basicPage(
  h3("Interactive table using the now deprecated Shiny data table renderer"),
  sliderInput("myslider", "Number of rows to display", min=1, max = 32, value = 5),
  shiny::dataTableOutput("mytable")

)


shinyApp(ui = ui, server = server)
```

Back to top

# **Example app:** Using the `DT` package for an interactive table

This app uses the `renderDataTable` function from DT.

Code 30 App 30

```
server <- function(input, output, session) {

  observe({
    updateTextInput(session, "mytext", value=input$myslider)
  })

  dat <- reactive({
    input$myslider
    mtcars[1:input$myslider,c("mpg", "cyl", "disp")]

  })

  # I'm setting paging = FALSE so all rows are shown all the time
  # scrollX adds a scrollbar, filter allows column filtering
  output$mytable <- DT::renderDataTable(dat(),
                                        options = list(paging=FALSE, scrollX = TRUE),
                                        rownames=TRUE,
                                        filter = "top")
}

ui <- basicPage(

    h3("Interactive table using the DT data table renderer"),
    sliderInput("myslider", "Number of rows to display", min=1, max = 32, value = 5),
    DT::dataTableOutput("mytable")

)

shinyApp(ui = ui, server = server)
```

Interactive plots and maps

HTML widgets allow you to use JavaScript visualization libraries to create interactive graphics in R, including Shiny web applications. For details on HTML widgets generally you can visit the htmlwidgets site. Below I show three examples, two examples of interactive plots (`plot.ly` and `highcharts`) and an interactive map with `leaflet` but there are many other widgets available.

Back to top

# **Example app:** Use plot.ly to make a ggplot interactive

Details on using plot.ly can be found at this site. In this example, I'm using plot.ly to make a ggplot interactive.

Code 31 App 31

```
library(plotly)

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

  dat <- reactive(cars[1:input$myslider,])

  output$myplot <- renderPlotly({
    p <- ggplot(dat(), aes(speed, dist)) + geom_point(color="purple")
    p <- ggplotly(p)
    p
  })
}

ui <- basicPage(

  h3("Example of plot.ly, the plot is interactive"),
  sliderInput("myslider", "A slider:", min=1, max=50, value=10),
  plotlyOutput("myplot")

)

shinyApp(ui = ui, server = server)
```

Back to top

# **Example app:** Use highcharts to make an interactive plot

Details on using highcharts can be found at this site. In this example, we create a scatter plot from scratch.

Code 32 App 32

```
library(highcharter)
library(magrittr) # for the pipe %>%

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

  dat <- reactive(cars[1:input$myslider,])

  output$myplot <- renderHighchart({
    highchart() %>%
      hc_title(text = "Scatter chart") %>%
      hc_add_serie_scatter(dat()$speed, dat()$dist)

    # here is the code if you don't want to use the %>% pipe
    # hc_add_serie_scatter(hc_title(highchart(),
    # text = "Scatter chart"), dat()$speed, dat()$dist)
  })
}

ui <-  basicPage(

  h3("Example of highcharter, the plot is interactive"),
  sliderInput("myslider", "A slider:", min=1, max=50, value=10),
  highchartOutput("myplot")
)

shinyApp(ui = ui, server = server)
```

Back to top

# **Example app:** Use leaflet for an interactive map

In this app the slider is linked to a map. Details on using leaflet in R can be found at this site.

Code 33 App 33

```
library(leaflet)
#### server
server <- function(input, output, session) {


  # create random points in the US
  dat <- reactive({
    long <- runif(input$myslider,-121, -77 )
    lat <- runif(input$myslider,33, 48)
    vals <- rpois(input$myslider, 5)
    data.frame(latitude = lat, longitude = long, vals)
  })

  output$mymap <- renderLeaflet({
    leaflet() %>% addProviderTiles("Stamen.TonerLite") %>%
      addCircleMarkers(data = dat(), radius = ~vals ) %>%
      setView(-98, 38.6, zoom=3)
  })
}

#### user interface
ui <- fluidPage(

  titlePanel("Example of leaflet interactive map"),

  sidebarLayout(

    sidebarPanel(
      h3("Slider changes number of map points"),
      sliderInput(inputId = "myslider", label = "Limit the ", min = 0,
                  max = 50, value = c(10))
    ), #endsidebarpanel

    mainPanel(
      tabsetPanel(
        tabPanel("Map", leafletOutput("mymap"))

      )
    )#end mainpanel
  )# end sidebarlayout
)

shinyApp(ui = ui, server = server)
```

Shiny modules

At the Shiny Developers Conference Garrett Grolemund, from RStudio, gave a great presentation on Shiny modules. They are designed, in part, to help solve the issue of re-useability discussed above in the on creating re-useable UI elements. You may have, as an example, a block of UI widgets you want to repeat on multiple pages. Rather than copy and paste code, you can use modules to help manage the pieces. For more detail on Shiny modules check out this write-up by Joe Cheng.

To use a module, in addition to creating the UI and server as we've done above, you would also create a module UI and module server. Here are the four pieces you would need for an application that uses one module

Module UI

This is a function that takes, as input, an id that will end up getting pre-pended to all your HTML element ids. You can actually manually paste together your prefix and the IDs but there is a helper function to make it a little easier called `NS`. Essentially `NS` just creates a new function you can use to do the pasting a little more simply.

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

  list(
    plotOutput(ns("plot1"))
  )

}
```

Module Server

The module server will include the processing needed for your module and looks almost identical to your non-module server function except that you will likely want to allow it to accept additional arguments. So below we have a scatter function that will render the plot and, beyond the usual input, output and session arguments it will accept a color argument.

```
scatter <- function(input, output, session, my_color) {

  output$plot1 <- renderPlot({
    ggplot(mtcars, aes_string(wt, mpg)) + geom_point(color=my_color, size=2)
  })
}
```

App UI

In your app UI you can include any UI elements that are not included in the module **and** you would include your module. Since your module UI is a function you would call this function and feed it a prefix, anything you want.

```
ui <- fluidPage(
  h3("This is not part of the module but the pieces below were created with a module"),
  scatterUI("prefix")
)
```

App Server

Similar to the app UI you would include any non-module related processing in your app server and then you include the module server with the function `callModule`. You feed `callModule` the name of the module server function and the prefix you're using and any additional arguments to your module server.

```
server <- function(input, output,session){

  callModule(scatter, "prefix", "purple")

}
```

Back to top

# **Example app:** simple module example

This example demonstrates the code needed to use a module, but is not a very practical example because the purpose of using modules would be to re-use code. See the next example for a more realistic example.

Code 34 App 34

```
library(ggplot2)
# module UI
scatterUI <- function(id) {
  ns <- NS(id)

  list(
    plotOutput(ns("plot1"))
  )
}

# module server
scatter <- function(input, output, session, my_color) {

  output$plot1 <- renderPlot({
    ggplot(mtcars, aes(wt, mpg)) + geom_point(color=my_color, size=2)
  })
}
```

```
# app ui
ui <- fluidPage(

  h3("This is not part of the module but the plot below was created with a module"),
  scatterUI("prefix")
)

# app server
server <- function(input, output,session){

  callModule(scatter, "prefix", "purple")

}

shinyApp(ui, server)
```

Back to top

# **Example app:** more practical module example

This app is a little harder to follow but a better example of how modules can be useful. In this app the module creates a slider and two plots. As input, the module function takes the name of the data.frame as well as several plot-related variables. This way you can create side-by-side plots with a slider using multiple datasets with slight alterations to the app server.

Code 35 App 35

```
library(ggplot2)

# MODULE UI
scatterUI <- function(id) {
  ns <- NS(id)

  list(
    div(sliderInput(ns("slider1"), label = "Limit points", min = 0, max = 32, value = 15)),
    div(style="display: inline-block; height:220px;", plotOutput(ns("plot1"))),
    div(style="display: inline-block; height:220px;", plotOutput(ns("plot2")))
  )
}

# MODULE Server
scatter <- function(input, output, session, datname, var1, var2, ptshape, col1, col2) {

  dat <- eval(as.name(datname))
  dat <- dat[order(dat[[var1]]),]

  resultdata <- reactive({
    dat[1:input$slider1,]
  })

  output$plot1 <- renderPlot({
    plot(1:10)
    ggplot(resultdata(), aes_string(var1, var2)) + geom_point(color=col1, shape=ptshape, size=3)+
      ggtitle(paste("Using the", datname, "data.frame"))
  }, width=200, height=200)

  output$plot2 <- renderPlot({
    plot(1:10)
    ggplot(resultdata(), aes_string(var1, var2)) + geom_point(color=col2, shape=ptshape, size=3)
      ggtitle(paste("Using the", datname, "data.frame"))
  }, width=200, height=200)
}


# App ui
ui <- fluidPage(
  h3("The module creates two plots and a slider and is called twice below"),
  scatterUI("prefix"),
  scatterUI("prefix2")
)

# App server
server <- function(input, output,session){

  callModule(scatter, "prefix", "cars", "speed", "dist",  1, "red", "blue")
  callModule(scatter, "prefix2", "mtcars", "mpg", "hp", 17, "forestgreen", "purple")
}

shinyApp(ui, server)
```

HTML templates

Some developers feel more comfortable writing HTML code directly rather than writing in R and having Shiny compile code to HTML. To meet this need, RStudio introduced the concept of HTML templates that allow you to write the HTML and include placeholders for Shiny input using curly braces. If you're comfortable with web development, the concept will be familiar (e.g. mustache, handlebars).

RStudio's Winston Chang has written a nice article with significant details so I'll cover it only briefly here.

You need the following items in a Shiny app using HTML templates:

You need an HTML page with references to Shiny components

In addition to your ui.R and server.R (or app.R) you would create an HTML page with the structure you want. Shiny-related components are then included in curly braces. A few key points to keep in mind:

1. You must include Shiny-related references in the header: Shiny apps reference Shiny-related JavaScript libraries and include other required information in the head of a Shiny app's HTML page so you need to include these references in your template. You could manually copy and paste them from another Shiny app, but it's much easier to use the `headContent()` function within curly braces.

2. You can also reference bootstrap in the header with a Shiny function: Including Bootstrap for styling is so common that Shiny includes a function to do it for you called `bootstrapLib`. Again you would use this in curly braces in your header.

3. Other shiny components can be included either directly in the HTML template or via references to the Shiny UI.

Here is what our sample app HTML file looks like:

```
<!DOCTYPE html>
<html>
<!-- this file is called template.html -->
<head>
    <!--This code is necessary to include important shiny-related references -->
    {{headContent()}}
    <!--To use bootstrap for styling this will add the references -->
    {{bootstrapLib()}}
</head>

<body>
    <div class="container">
        <!-- I'm using HTML directly -->
        <div class="jumbotron" style="text-align: center;">
            <h2 class="display-3">HTML templates with Shiny</h2>
            <p class="lead">Details on creating Shiny apps with HTML templates. And a random phot
            <img src="http://lorempixel.com/200/100" />
        </div>

        <h2>Block of controls from the Shiny UI</h2>
        <!-- shinycontrols is an object from the ui -->
        {{shinycontrols}}

        <h2>Slider and radio buttons</h2>
        <!-- justslider is an object from the ui -->
        {{justslider}}
        {{radioButtons("my_radio", "Radio buttons from the HTML template", c("A", "B", "C"), "C")


        <h2>You can do calculations with R directly</h2>

        The average of a sequence of numbers from 1 to 10 is {{mean(1:10)}}!
    </div>
</body>
</html>
```

You need a UI that reads the HTML template

Your server and UI will be very similar to an all-R app except that your UI needs to read the HTML template, and feed the template whatever Shiny components you'll want on the page. This is done with the `htmlTemplate` function.

So in the current example, I am reading the `template.html` file in the same directory as app.R and I am also using the `htmlTemplate` function to feed two objects, `shinycontrols` which is a block of Shiny controls, and `justslider` which is a simple slider to the `template.html`.

```
# here are some elements I want to include as a block
block_of_stuff <- list(
  selectInput("my_select", "Select something please", c("A", "B", "C"), selected="B"),
  textInput("my_text", "A text box", value = "initial value")
)

# user interface
ui <- fluidPage(

  htmlTemplate("template.html",
               shinycontrols =  block_of_stuff,
               justslider = sliderInput("my_slider", "A slider from the Shiny UI", 0, 10, 5)
  )
)
```

Back to top

# **Example app:** An app using HTML templates

This app reads my file, `template.html` and then feeds it a block of controls called `shinycontrols` and a slider called `justslider`.

Code 36 App 36

```
server <- function(input, output, session) {

}

# here are some elements I want to include as a block
block_of_stuff <- list(
  selectInput("my_select", "Select something please", c("A", "B", "C"), selected="B"),
  textInput("my_text", "A text box", value = "initial value")
)

# user interface
ui <- htmlTemplate("template.html",
                   shinycontrols =  block_of_stuff,
                   justslider = sliderInput("my_slider", "A slider from the Shiny UI", 0, 10, 5)
)


shinyApp(ui = ui, server = server)
```

Brush, click and hover on plots (and an example of using global variables)

In mid-2015 RStudio added plot interactivity to Shiny's arsenal – this is a little different, though, than the HTML widgets mentioned above. This interactivity allows users to interact with "regular" base plots and ggplot2 plots. Specifically, these functions allow your users to click, hover or brush a base or ggplot to get more information on the elements clicked etc. Adding this type of functionality can be done simply by adding an argument to the `plotOutput` function and using that output in your server.

Back to top

# **Example app:** Brush and click on a ggplot

As an example of interactive plots, consider this reasonably simple app. You'll notice that in the `plotOutput` function I added two arguments, one for click and one for brush. And then in the reactive I use both of these. The actual resulting input variables (`input$user_brush` and `input$user_click`) provide details like the x and y coordinates clicked. To use this information to filter your table you can use the corresponding `brushedPoints` and `nearPoints` functions.

A note about this particular example: remember that any reactives in your reactive function will trigger the code to run. As a result, if the user either brushes or clicks the `dat` reactive will run. But… it won't know which reactive variable triggered the code to run. To get around this, I've created a global variable called `interaction_type` that stores the current type of interaction (i.e., brush or click). There are two `observeEvent` functions that will listen and update as needed. Pay special attention to the double assignment operator (`<<-`), this is needed to change the global variable.

Code 37 App 37

```
library(ggplot2)


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

  # global variable, what type of plot interaction
  interaction_type <- "click"

  # observe for user interaction and change the global interaction_type
  # variable
  observeEvent(input$user_click, interaction_type <<- "click")
  observeEvent(input$user_brush, interaction_type <<- "brush")

  output$plot <- renderPlot({
    ggplot(mtcars, aes(wt, mpg)) + geom_point()
  })

  # generate the data to put in the table
  dat <- reactive({

    user_brush <- input$user_brush
    user_click <- input$user_click

    if(interaction_type == "brush") res <- brushedPoints(mtcars, user_brush)
    if(interaction_type == "click") res <- nearPoints(mtcars, user_click, threshold = 10, maxpoir

    return(res)

  })

  output$table <- DT::renderDataTable(DT::datatable(dat()[,c("mpg", "cyl", "disp")]))

}
```

```
ui <- fluidPage(

  h3("Click or brush the plot and it will filter the table"),
  plotOutput("plot", click = "user_click", brush = "user_brush"),
  DT::dataTableOutput("table")

)

shinyApp(ui = ui, server = server)
```

Back to top

# Example app: Brush with map and plot

This app is a slightly more complex version of the brush functionality. In this example, the user selections update in both a plot and on a leaflet map. The approach is very similar, though, to the previous app. We include a `brush` argument in the `plotOutput` and we use a brush to select points with the `brushedPoints` function.

This app example was inspired by an app from Kyle Walker that displays the demographics of Texas Counties. He generously posted the code used to create it.

Code 38 App 38

```
library(leaflet)
library(ggplot2)



#### server
server <- function(input, output, session) {

  dat<-reactive({
    long <- runif(input$myslider, -121, -77 )
    lat <- runif(input$myslider, 33, 48)
    val1 <- runif(input$myslider, 1, 50)
    val2 <- rnorm(input$myslider, 1, 50)
    data.frame(latitude = lat, longitude = long, val1, val2)
  })

  br <- reactive({
    brushedPoints(dat(), input$plot_brush)
  })

  output$plot <- renderPlot({
    ggplot(dat(), aes(val1, val2, color=val2)) + geom_point() +
      ggtitle("Select points by brushing graph")
  })

  output$mymap <- renderLeaflet({

   # if nothing has been selected use the data itself, otherwise
   # use a brushed version
   ifelse(is.null(input$plot_brush), dat <- dat(), dat <- br() )

    leaflet(data = dat) %>%
      addProviderTiles("CartoDB.Positron",
                       options = providerTileOptions(noWrap = TRUE)) %>%
      addMarkers(~longitude, ~latitude) %>%
      setView(-98, 38.6, zoom=3)
  })



  output$table_brushedpoints <- renderTable({
    br()[,c("val1", "val2")]
  })
}


#### user interface
ui <- fluidPage(

  titlePanel("Example of leaflet interactive map using brushing"),

  sidebarLayout(

    sidebarPanel(
      h3("Slider changes number of points"),
      sliderInput(inputId = "myslider", "Number of points:",
                  min = 5, max = 20, value = 10, step = 1),
      plotOutput("plot", height = 250,
                 brush = brushOpts(id = "plot_brush", resetOnNew = TRUE,
                                   fill = "red", stroke = "#036", opacity = 0.3)),
      h4("Brushed points appear here"),
      tableOutput("table_brushedpoints")
    ), #endsidebarpanel

    mainPanel(
      tabsetPanel(
        tabPanel("Map", leafletOutput("mymap"))

      )
    )#end mainpanel
  )# end sidebarlayout
)


shinyApp(ui = ui, server = server)
```

Including custom JavaScript in your app

Occasionally you may want to include some JavaScript that you've written in your Shiny app. Similar to the discussion of CSS above you can include the JS by referencing an external file or you can include JS inline in the header.

1. `tags$head(includeScript("google-analytics.js"))`
2. `tags$head(HTML("<script type='text/javascript'></script>"))`

Back to top

# **Example app:** inline JavaScript code

Here we add inline JavaScript (jQuery) to fade out, then in the title and finally make it green.

Code 39 App 39

```
server <- function(input, output, session) {

}

ui <- basicPage(

    h1(id = "thetitle", "Inline JavaScript to change the title style"),
    actionButton("thebutton", "Click to change title color and transparency"),

    # here we're including inline JavaScript code (jQuery)
    tags$footer(HTML("<script type='text/javascript'>
                      $( document ).ready(function() {
                      var $thetitle = $('#thetitle');
                      var $thebutton  = $('#thebutton');
                      $thebutton.click(function(){
                      $thetitle.fadeOut(2000);
                      $thetitle.fadeIn(2000);
                      $thetitle.css('color', 'green');
                      })

                      });
                      </script>"))
)


shinyApp(ui = ui, server = server)
```

Shiny dashboards

With an R backend for processing and strong plotting capabilities Shiny is well-suited to creating dashboards. Dashboards can be created manually using the techniques discussed above, but creating them is simpler with the Rstudio package `shinydashboard`. In order to create a dashboard you need a header, sidebar and body. Within the body you create a series `tabItems` and fill them with the controls you want to see included. For more detail you should refer directly to the shiny dashboard page on GitHub.

Back to top

# **Example app:** A simple dashboard

This code was stolen, in a large part, from the shiny dashboard get started page.

Code 40 App 40

```
library(ggplot2)
library(shinydashboard)

ui <- dashboardPage(
  dashboardHeader(title = "Basic dashboard"),
  dashboardSidebar(
     sidebarMenu(
      menuItem("Dashboard", tabName = "dashboard", icon = icon("dashboard")),
      menuItem("Widgets", tabName = "widgets", icon = icon("th"))
    )
  ),
  dashboardBody(
    tabItems(
      # First tab content
      tabItem(tabName = "dashboard",
        fluidRow(
          box(plotOutput("plot1", height = 250)),
          box(plotOutput("plot2", height = 250)),

          box(
            title = "Controls",
            sliderInput("myslider", "Number of observations:", 1, 50, 15)
          )
        )
      ),

      # Second tab content
      tabItem(tabName = "widgets",
        h2("Widgets tab content")
      )
    )
  )
)

server <- function(input, output) {
  histdata <- rnorm(500)

  output$plot1 <- renderPlot({
    dat <- mtcars[1:input$myslider,]
    ggplot(dat, aes(mpg)) + geom_histogram(fill="cadetblue")
  })
   output$plot2 <- renderPlot({
    dat <- mtcars[1:input$myslider,]
    ggplot(dat, aes(mpg, wt)) + geom_point() + stat_smooth()
  })
}

shinyApp(ui, server)
```

More on reactive programming

Joe Cheng, the primary creator of Shiny, gave a great talk on reactive programming at the Shiny Developers Conference in January 2016. The videos are now online and I highly recommend you watch both part 1 and part 2.

Notes on the creation of this post

The output of `sessionInfo()` is below in case you run into trouble related to versions of Shiny or other packages discussed. RStudio is generously hosting the 40 mini-apps in the post and they are shown here using iFrames. Browsers limit the number of connections (iFrames) on one page so I could not load all 40 apps at once. I've done my best to speed things up by pre-loading apps when users click on certain links but in some case you need to wait for an app load or reload an app and I apologize if this is slow.

```
sessionInfo()
```

```
## R version 3.2.3 (2015-12-10)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 7 x64 (build 7601) Service Pack 1
##
## locale:
## [1] LC_COLLATE=English_United States.1252
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] htmlwidgets_0.6    leaflet_1.0.1.9002 DT_0.1.55
## [4] shiny_0.13.1.9002
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.4     digest_0.6.9    mime_0.4        R6_2.1.2
##  [5] xtable_1.8-0    formatR_1.2.1   magrittr_1.5    evaluate_0.8
##  [9] stringi_1.0-1   rmarkdown_0.9.5 tools_3.2.3     stringr_1.0.0
## [13] httpuv_1.3.3    yaml_2.1.13     htmltools_0.3.5 knitr_1.12.3
```

Posted in R, Shiny

11 responses

Matt
April 21, 2016 at 6:57 pm

Hi Zev,

Great website!

I'm new to Shiny and was wondering how long it would take to learn how to use Shiny if one has only 6 months of general programming experience?

Best wishes,
Matt

zev@zevross.com
April 22, 2016 at 8:29 am

Shiny can be tricky but it depends on your motivation and your goal. With some previous R background you can learn to make simple Shiny apps in a few hours. Ultimately it depends on what you want to accomplish.

Allen
April 23, 2016 at 7:21 am

Hi – thanks for all the hard work on this article! I am in the same boat as the previous poster. Have some familiarity with R but no exposure to Shiny. Can Shiny import CSV file with data and do analysis as can be done with R in desktop environment. That is my main interest. If you could direct me to the right place to study this (importing csv file) I would greatly appreciate it. Thanks.

zev@zevross.com
April 27, 2016 at 9:39 am

If you just type r shiny read.csv into Google you'll get all the examples you need. The RStudio example is http://shiny.rstudio.com/gallery/upload-file.html

lincoln
April 24, 2016 at 3:26 pm

Extremely helpful. Thank you very much.

Joseph Pang
May 27, 2016 at 4:27 am

Can we create multiple webpages/workflows using Shiny? If so, how? I only want to run it locally in my machine at the beginning.

zev@zevross.com
May 27, 2016 at 9:13 am

It depends what you mean. One app will be one "webpage" essentially, but you can have tabs or other pages attached. See, for example, the app I created with the World Health Organization: https://whoequity.shinyapps.io/HEAT/ which has several pages under "about" and then different tabs and pages associated with the navbar. I'm not sure what you mean by workflows.

Emily
June 8, 2016 at 3:24 am

Hello,

Great website.

So I have made a few apps in the past and they were all working fine and deployed no problem to the server. However, I recently updated my RStudio to 0.99.902 and now my apps won't run due to the following error:

"(TypeError) : undefined is not an object (evaluating 'c.b')"

Any ideas?

zev@zevross.com
June 8, 2016 at 8:26 am

Not without seeing code, why don't you put a reproducible chunk of code on stackoverflow and send me the link.

satheesh

June 17, 2016 at 4:10 am

Hi Zev,

Great website. I'm new to Rstudio. I'm using 3.2.3 version. I have installed Shiny package. While running the app I'm getting a message saying UPDATED VERSION OF SHINY PACKAGES REQUIRED and if I click on yes, installing shiny and I'm not getting any output.

Where I'm doing wrong?

Let me know if you need any more information.

Your inputs are much appreciated.

Thank you
Satheesh K

zev@zevross.com

June 17, 2016 at 8:15 am

Please post a question to stackoverflow with an exact, reproducible, example and send me the link