One of the great additions to the R ecosystem in recent years is RStudio’s Shiny package. With it, you can easily whip up and share a user interface for a new statistical method in just a few hours. Today I want to share some of the methods and challenges that come up when the actual computation of a result takes a non-trivial amount of time (e.g >5 seconds).

**First Attempt**

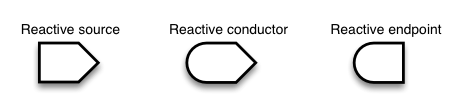
Shiny operates in a reactive programming framework. Fundamentally this means that any time any UI element that affects the result changes, so does the result. This happens automatically, with your analysis code running every time a widget is changed. In a lot of cases, this is exactly what you want and it makes Shiny programs concise and easy to make; however in the case of long running processes, this can lead to frozen UI elements and a frustrating user experience.

The easiest solution is to use an Action Button and only run the analysis code when the action button is clicked. Another important component is to provide your user with feedback as to how long the analysis is going to take. Shiny has nice built in progress indicators that allow you to do this.

Reactive programming Framework

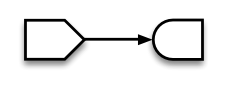
It’s easy to build interactive applications with Shiny, but to get the most out of it, you’ll need to understand the reactive programming model used by Shiny.

In Shiny, there are three kinds of objects in reactive programming: reactive sources, reactive conductors, and reactive endpoints, which are represented with these symbols:



Reactive sources and endpoints

The simplest structure of a reactive program involves just a source and an endpoint:



In a Shiny application, the source typically is user input through a browser interface. For example, when the user selects an item, types input, or clicks on a button, these actions will set values that are reactive sources. A reactive endpoint is usually something that appears in the user’s browser window, such as a plot or a table of values.

In a simple Shiny application, reactive sources are accessible through the input object, and reactive endpoints are accessible through the output object. (Actually, there are other possible kinds of sources and endpoints, which we’ll talk about later, but for now we’ll just talk about input and output.)

This simple structure, with one source and one endpoint, is used by the 01\_hello example. The server function code for that example looks something like this:

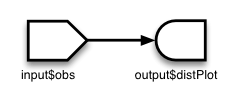
server <- **function**(input, output) {

output$distPlot <- renderPlot({

hist(rnorm(input$obs))

})

}



The output$distPlot object is a reactive endpoint, and it uses the reactive source input$obs. Whenever input$obs changes, output$distPlot is notified that it needs to re-execute. In traditional program with an interactive user interface, this might involve setting up event handlers and writing code to read values and transfer data. Shiny does all these things for you behind the scenes, so that you can simply write code that looks like regular R code.

A reactive source can be connected to multiple endpoints, and vice versa. Here is the server function of a slightly more complex Shiny application:

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

output$plotOut <- renderPlot({

hist(faithful$eruptions, breaks = as.numeric(input$nBreaks))

**if** (input$individualObs)

rug(faithful$eruptions)

})

output$tableOut <- renderTable({

**if** (input$individualObs)

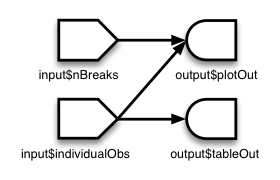
faithful

**else**

NULL

})

}



In a Shiny application, there’s no need to explictly describe each of these relationships and tell R what to do when each input component changes; Shiny automatically handles these details for you.

In an app with the structure above, whenever the value of the input$nBreaks changes, the expression that generates the plot will automatically re-execute. Whenever the value of the input$individualObs changes, the plot and table functions will automatically re-execute. (In a Shiny application, most endpoint functions have their results automatically wrapped up and sent to the web browser.)

Reactive conductors

So far we’ve seen reactive sources and reactive endpoints, and most simple examples use just these two components, wiring up sources directly to endpoints. It’s also possible to put reactive components in between the sources and endpoints. These components are called *reactive conductors*.

A conductor can both be a dependent and have dependents. In other words, it can be both a parent and child in a graph of the reactive structure. Sources can only be parents (they can have dependents), and endpoints can only be children (they can be dependents) in the reactive graph.

Reactive conductors can be useful for encapsulating slow or computationally expensive operations. For example, imagine that you have this application that takes a value input$n and prints the \_n\_th value in the Fibonacci sequence, as well as the inverse of \_n\_th value in the sequence plus one (note the code in these examples is condensed to illustrate reactive concepts, and doesn’t necessarily represent coding best practices):

*# Calculate nth number in Fibonacci sequence*

fib <- **function**(n) ifelse(n<3, 1, fib(n-1)+fib(n-2))

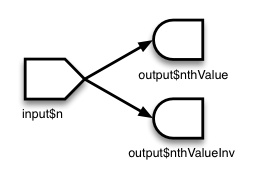
server <- **function**(input, output) {

output$nthValue <- renderText({ fib(as.numeric(input$n)) })

output$nthValueInv <- renderText({ 1 / fib(as.numeric(input$n)) })

}

The graph structure of this app is:



The fib() algorithm is very inefficient, so we don’t want to run it more times than is absolutely necessary. But in this app, we’re running it twice! On a reasonably fast modern machine, setting input$n to 30 takes about 15 seconds to calculate the answer, largely because fib() is run twice.

The amount of computation can be reduced by adding a reactive conductor in between the source and endpoints:

fib <- **function**(n) ifelse(n<3, 1, fib(n-1)+fib(n-2))

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

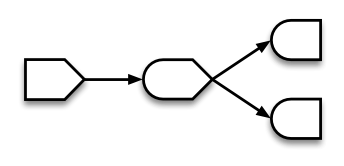
currentFib <- reactive({ fib(as.numeric(input$n)) })

output$nthValue <- renderText({ currentFib() })

output$nthValueInv <- renderText({ 1 / currentFib() })

}

Here is the new graph structure:



Keep in mind that if your application tries to access reactive values or expressions from outside a reactive context — that is, outside of a reactive expression or observer — then it will result in an error. You can think of there being a reactive “world” which can see and change the non-reactive world, but the non-reactive world can’t do the same to the reactive world. Code like this will not work, because the call to fib() is not in the reactive world (it’s not in a reactive() or renderXX() call) but it tries to access something that is, the reactive value input$n:

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

*# Will give error*

currentFib <- fib(as.numeric(input$n))

output$nthValue <- renderText({ currentFib })

}

On the other hand, if currentFib is a function that accesses a reactive value, and that function is called within the reactive world, then it will work:

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

*# OK, as long as this is called from the reactive world:*

currentFib <- **function**() {

fib(as.numeric(input$n))

}

output$nthValue <- renderText({ currentFib() })

}

Summary

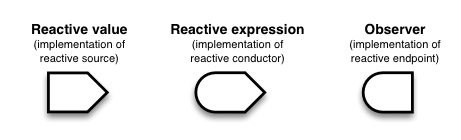
In this section, we’ve learned about:

* **Reactive sources** can signal objects downstream that they need to re-execute.
* **Reactive conductors** are placed somewhere in between sources and endpoints on the reactive graph. They are typically used for encapsulating slow operations.
* **Reactive endpoints** can be told to re-execute by the reactive environment, and can request *upstream* objects to execute.
* **Invalidation arrows** diagram the flow of invalidation events. It can also be said that the child node is a **dependent of** or **takes a dependency on** the parent node.

Implementations of sources, conductors, and endpoints: values, expressions, and observers

We’ve discussed reactive sources, conductors, and endpoints. These are general terms for parts that play a particular role in a reactive program. Presently, Shiny has one class of objects that act as reactive sources, one class of objects that act as reactive conductors, and one class of objects that act as reactive endpoints, but in principle there could be other classes that implement these roles.

* **Reactive values** are an implementation of Reactive sources; that is, they are an implementation of that role.
* **Reactive expressions** are an implementation of Reactive conductors. They can access reactive values or other reactive expressions, and they return a value.
* **Observers** are an implementation of Reactive endpoints. They can access reactive sources and reactive expressions, and they don’t return a value; they are used for their side effects.



All of the examples use these three implementations, as there are presently no other implementations of the source, conductor, and endpoint roles.

Reactive values

Reactive values contain values (not surprisingly), which can be read by other reactive objects. The input object is a ReactiveValues object, which looks something like a list, and it contains many individual reactive values. The values in input are set by input from the web browser.

Reactive expressions

We’ve seen reactive expressions in action, with the Fibonacci example above. They cache their return values, to make the app run more efficiently. Note that, abstractly speaking, *reactive conductors* do not necessarily cache return values, but in this implementation, *reactive expressions*, they do.

A reactive expressions can be useful for caching the results of any procedure that happens in response to user input, including:

* accessing a database
* reading data from a file
* downloading data over the network
* performing an expensive computation

Observers

Observers are similar to reactive expressions, but with a few important differences. Like reactive expressions, they can access reactive values and reactive expressions. However, they do not return any values, and therefore do not cache their return values. Instead of returning values, they have side effects – typically, this involves sending data to the web browser.

The output object looks something like a list, and it can contain many individual observers.

If you look at the code for renderText() and friends, you’ll see that they each return a function which returns a value. They’re typically used like this:

output$number <- renderText({ as.numeric(input$n) + 1 })

This might lead you to think that the observers *do* return values. However, this isn’t the whole story. The function returned by renderText() is actually not an observer/endpoint. When it is assigned to output$x, the function returned by renderText() gets automatically wrapped into another function, which is an observer. The wrapper function is used because it needs to do special things to send the data to the browser.

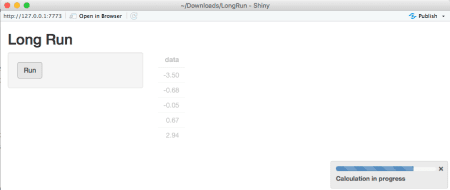
Differences between reactive expressions and observers

Reactive expressions and observers are similar in that they store expressions that can be executed, but they have some fundamental differences.

* Observers (and endpoints in general) respond to reactive *flush* events, but reactive expressions (and conductors in general) do not. We’ll learn more about flush events in the next section. If you want a reactive expression to execute, it must have an observer as a descendant on the reactive dependency graph.
* Reactive expressions return values, but observers don’t.

[?](http://www.ericbess.com/ericblog/2008/03/03/wp-codebox/#examples)View Code R

|  |
| --- |
| library(shiny)    # Define UI for application that draws a histogram  ui <- fluidPage(    # Application title  titlePanel("Long Run"),    # Sidebar with a slider input for number of bins  sidebarLayout(  sidebarPanel(  actionButton('run', 'Run')  ),    # Show a plot of the generated distribution  mainPanel(  tableOutput("result")  )  )  )    server <- function(input, output) {  N <- 10    result\_val <- reactiveVal()  observeEvent(input$run,{  result\_val(NULL)  withProgress(message = 'Calculation in progress', {  for(i in 1:N){    # Long Running Task  Sys.sleep(1)    # Update progress  incProgress(1/N)  }  result\_val(quantile(rnorm(1000)))  })  })  output$result <- renderTable({  result\_val()  })  }    # Run the application  shinyApp(ui = ui, server = server) |

[](https://i2.wp.com/blog.fellstat.com/wp-content/uploads/2018/07/Screen-Shot-2018-07-30-at-10.50.37-AM.png)

The above implementation has some of the things we want out of our interface:

* The long running analysis is only executed when “Run” is clicked.
* Progress is clearly displayed to the user.

It does have some serious downsides though:

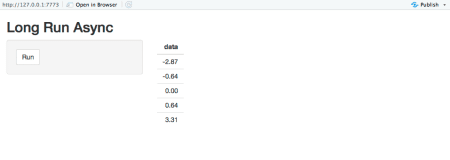
* **If “Run” is clicked multiple times, the analysis is run back to back.** A frustrated user can easily end up having to abort their session because they clicked to many times.
* **There is no way to cancel the calculation.** The session’s UI is locked while the computation takes place. Often a user will realize that some of the options they’ve selected are incorrect and will want to restart the computation. With this interface, they will have to wait however long the computation takes before they can fix the issue.
* **The whole server is blocked while the computation takes place.** If multiple users are working with the app, the UIs of all users are frozen while any one user has an analysis in progress.

**A Second Attempt With Shiny Async**

Having the whole server blocked is a big issue if you want to have your app scale beyond a single concurrent user. Fortunately, Shiny’s new support of asynchronous processing can be used to remove this behavior. Instead of assigning a value to the reactive value ‘result\_val’, we will instead create a promise to execute the analysis in the future (using the future function) and when it is done assign it to result\_val (using %…>%).

[?](http://www.ericbess.com/ericblog/2008/03/03/wp-codebox/#examples)View Code R

|  |
| --- |
| library(shiny)  library(promises)  library(future)  plan(multiprocess)    # Define UI for application that draws a histogram  ui <- fluidPage(    # Application title  titlePanel("Long Run Async"),    # Sidebar with a slider input for number of bins  sidebarLayout(  sidebarPanel(  actionButton('run', 'Run')  ),    # Show a plot of the generated distribution  mainPanel(  tableOutput("result")  )  )  )    server <- function(input, output) {  N <- 10    result\_val <- reactiveVal()  observeEvent(input$run,{  result\_val(NULL)  future({  print("Running...")  for(i in 1:N){  Sys.sleep(1)  }  quantile(rnorm(1000))  }) %...>% result\_val()  })  output$result <- renderTable({  req(result\_val())  })  }    # Run the application  shinyApp(ui = ui, server = server) |

[](https://i0.wp.com/blog.fellstat.com/wp-content/uploads/2018/07/Screen-Shot-2018-07-30-at-11.20.04-AM.png)

When “Run” is clicked, the UI is now blocked only for the individual performing the analysis. Other users will be able to perform analyses of there own concurrently. That said, we still have some undesirable properties:

* **If “Run” is clicked multiple times, the analysis is run back to back.**
* **There is no way to cancel the calculation.**
* **The user cannot monitor progress**. Shiny’s progress bar updates do not support calling them from within future, so we’ve had to remove the progress bar from the UI. This is not a huge problem for tasks that take a few seconds, but for those that take minutes or hours, not knowing how long until the results show up can be very frustrating.

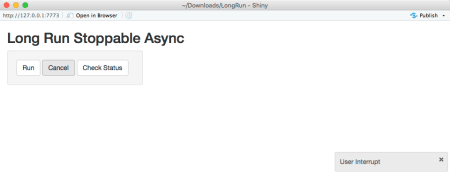
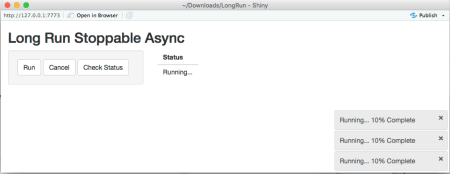
**Third Time Is the Charm**

In order to solve the cancel and monitoring problems, we need to be able to communicate between the app and the inside of the promise. This can be accomplished with the use of a file, where progress and interrupt requests are read and written. If the user clicks the cancel button, “interrupt” is written to the file. During the course of the computation the analysis code checks whether interrupt has been signaled and if so, throws an error. If no interrupt has been requested, the analysis code writes its progress to the file. If Status is clicked, Shiny reads the file and notifies the user of its contents.

The last addition to the code is to create a reactive value nclicks that prevents the Run button from triggering multiple analyses.

[?](http://www.ericbess.com/ericblog/2008/03/03/wp-codebox/#examples)View Code R

|  |
| --- |
| library(shiny)  library(promises)  library(future)  plan(multiprocess)    ui <- fluidPage(  titlePanel("Long Run Stoppable Async"),  sidebarLayout(  sidebarPanel(  actionButton('run', 'Run'),  actionButton('cancel', 'Cancel'),  actionButton('status', 'Check Status')  ),  mainPanel(  tableOutput("result")  )  )  )    server <- function(input, output) {  N <- 10    # Status File  status\_file <- tempfile()    get\_status <- function(){  scan(status\_file, what = "character",sep="\n")  }    set\_status <- function(msg){  write(msg, status\_file)  }    fire\_interrupt <- function(){  set\_status("interrupt")  }    fire\_ready <- function(){  set\_status("Ready")  }    fire\_running <- function(perc\_complete){  if(missing(perc\_complete))  msg <- "Running..."  else  msg <- paste0("Running... ", perc\_complete, "% Complete")  set\_status(msg)  }    interrupted <- function(){  get\_status() == "interrupt"  }    # Delete file at end of session  onStop(function(){  print(status\_file)  if(file.exists(status\_file))  unlink(status\_file)  })    # Create Status File  fire\_ready()      nclicks <- reactiveVal(0)  result\_val <- reactiveVal()  observeEvent(input$run,{    # Don't do anything if analysis is already being run  if(nclicks() != 0){  showNotification("Already running analysis")  return(NULL)  }    # Increment clicks and prevent concurrent analyses  nclicks(nclicks() + 1)    result\_val(data.frame(Status="Running..."))    fire\_running()    result <- future({  print("Running...")  for(i in 1:N){    # Long Running Task  Sys.sleep(1)    # Check for user interrupts  if(interrupted()){  print("Stopping...")  stop("User Interrupt")  }    # Notify status file of progress  fire\_running(100\*i/N)  }    #Some results  quantile(rnorm(1000))  }) %...>% result\_val()    # Catch inturrupt (or any other error) and notify user  result <- catch(result,  function(e){  result\_val(NULL)  print(e$message)  showNotification(e$message)  })    # After the promise has been evaluated set nclicks to 0 to allow for anlother Run  result <- finally(result,  function(){  fire\_ready()  nclicks(0)  })    # Return something other than the promise so shiny remains responsive  NULL  })    output$result <- renderTable({  req(result\_val())  })    # Register user interrupt  observeEvent(input$cancel,{  print("Cancel")  fire\_interrupt()  })    # Let user get analysis progress  observeEvent(input$status,{  print("Status")  showNotification(get\_status())  })  }    # Run the application  shinyApp(ui = ui, server = server) |

[](https://i0.wp.com/blog.fellstat.com/wp-content/uploads/2018/07/Screen-Shot-2018-07-30-at-11.42.08-AM.png)[](https://i2.wp.com/blog.fellstat.com/wp-content/uploads/2018/07/Screen-Shot-2018-07-30-at-11.41.55-AM.png)

All three of the problems with the original async code have been solved with this implementation. That said, some care should be taken when using async operations like this. It is possible for race conditions to occur, especially if you have multiple “Run” buttons in a single app.

Action Button with Shiny App

## **How action buttons work**

Create an action button with actionButton() and an action link with actionLink(). Each of these functions takes two arguments:

* inputId - the ID of the button or link
* label - the label to display in the button or link

actionButton("button", "An action button")

actionLink("button", "An action link")

An action button appears as a button in your app.

An action link appears as a hyperlink, but behaves in the same way as an action button.

Like all widgets, action buttons have a value. The value is an integer that changes each time a user clicks the button. You can access this value from within your app as input$<inputId> where <inputId> is the ID that you assigned to your action button.

Action buttons are different from other widgets because the value of an action button is almost never meaningful by itself. The value is designed to be observed by one of observeEvent() or eventReactive(). These functions monitor the value, and when it changes they run a block of code.

The patterns below explain this arrangement and illustrate the most popular ways to use an action button or an action link.

## **Pattern 1 - Command**

Use observeEvent() to trigger a command with an action button.

### Example

In the code above, session$setCustomMessage() generates a popup message. tags$head(tags$script(src = "message-handler.js")) supplies the JavaScript that makes this possible. See [this example](http://shiny.rstudio.com/gallery/server-to-client-custom-messages.html) to learn more aboutsendCustomMessage().

### Why the pattern works

Action buttons do not automatically generate actions in Shiny. Like other widgets, action buttons maintain a state (a value). The state changes when a user clicks the button.

observeEvent() observes a reactive value, which is set in the first argument of observeEvent(). Whenever the value changes, observeEvent() will run its second argument, which should be a block of code surrounded in braces.

This pattern uses observeEvent() to connect the change in an action button’s value to the code that the action button should trigger.

### Tips

* observeEvent() isolates the block of code in its second argument with isolate().
* observeEvent() only notices changes in the value of the action button. It does not matter what the actual value of the button is. If your code depends on the value of the action button, it may be mis-written.

Shiny Code

library(shiny)

ui <- fluidPage(

tags$head(tags$script(src = "message-handler.js")),

actionButton("do", "Click Me")

)

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

observeEvent(input$do, {

session$sendCustomMessage(type = 'testmessage',

message = 'Thank you for clicking')

})

}

shinyApp(ui, server)

## **Pattern 2 - Delay reactions**

Use eventReactive() to delay reactions until a user clicks the action button.

### Example

### Why the pattern works

eventReactive() creates a reactive expression that monitors a reactive value, which is set in the first argument of eventReactive(). The expression will be invalidated whenever the value changes, but it will ignore changes in other reactive values.

Complete this pattern by using the reactive expression created by eventReactive() in rendered output. Output that depends on the expression will not update until the expression is invalidated, i.e. until the action button is clicked.

### Tips

* Like observeEvent(), eventReactive() isolates the block of code in its second argument with isolate().
* eventReactive() returns NULL until the action button is clicked. As a result, the graph does not appear until the user asks for it by clicking “Go”.

Shiny Code

library(shiny)

ui <- fluidPage(

actionButton("go", "Go"),

numericInput("n", "n", 50),

plotOutput("plot")

)

server <- function(input, output) {

randomVals <- eventReactive(input$go, {

runif(input$n)

})

output$plot <- renderPlot({

hist(randomVals())

})

}

shinyApp(ui, server)

## **Pattern 3 - Dueling buttons**

To build several action buttons that control the same object, combine observeEvent() calls with reactiveValues().

### Example

### Why the pattern works

reactiveValues() creates a reactive values object, a list of reactive values that you can update and call programmatically. These values are like the values stored in Shiny’s input object with one difference: you can update the values of a reactive values object, but you cannot normally update the values of the input object (those values are reserved for the user to update interactively).

To complete the pattern, monitor each button with its own observeEvent() call. Arrange for the calls to update the object created by reactiveValues(). Reactive values obey reference class semantics, which means that you can update them from within the scope of an observeEvent() function.

Shiny Code

library(shiny)

ui <- fluidPage(

actionButton("runif", "Uniform"),

actionButton("rnorm", "Normal"),

hr(),

plotOutput("plot")

)

server <- function(input, output){

v <- reactiveValues(data = NULL)

observeEvent(input$runif, {

v$data <- runif(100)

})

observeEvent(input$rnorm, {

v$data <- rnorm(100)

})

output$plot <- renderPlot({

if (is.null(v$data)) return()

hist(v$data)

})

}

shinyApp(ui, server)

## **Pattern 4 - Reset buttons**

To create a reset button, use the above pattern to assign NULL to a reactive values object.

### Example

### Why this pattern works

You can apply the previous pattern to reset an element of a reactvie values object to its intial state (NULL). To do this, arrange for a button to assign NULL to the reactive values object with the help of observeEvent().

Shiny Code

library(shiny)

ui <- fluidPage(

actionButton("runif", "Uniform"),

actionButton("reset", "Clear"),

hr(),

plotOutput("plot")

)

server <- function(input, output){

v <- reactiveValues(data = NULL)

observeEvent(input$runif, {

v$data <- runif(100)

})

observeEvent(input$reset, {

v$data <- NULL

})

output$plot <- renderPlot({

if (is.null(v$data)) return()

hist(v$data)

})

}

shinyApp(ui, server)

## **Pattern 5 - Reset on tab change**

Observe the value of a tabsetPanel(), navlistPanel(), or navbarPage() with observeEvent() to rest the value of an object each time your user switches tabs.

### Example

### Why this pattern works

This pattern extends the previous reset pattern. You use observeEvent() to reset an element of a reactive values object. However, instead of observing the value of an action button, you observe the value of a tab function.

tabsetPanel(), navlistPanel(), and navbarPage() each combine multiple tabs (created with tabPanel()) into a single ui object. These functions maintain a reactive value that contains the title of the current tab. When your user navigates to a new tab, this value changes. observeEvent() resets the reactive value to NULL when it does.

As with the patterns above, this pattern requires you to store and manipulate a value created with reactiveValues().

Shiny Code

library(shiny)

ui <- fluidPage(

sidebarLayout(

sidebarPanel(

tabsetPanel(id = "tabset",

tabPanel("Uniform",

numericInput("unifCount", "Count", 100),

sliderInput("unifRange", "Range", min = -100, max = 100, value = c(-10, 10))

),

tabPanel("Normal",

numericInput("normCount", "Count", 100),

numericInput("normMean", "Mean", 0),

numericInput("normSd", "Std Dev", 1)

)

),

actionButton("go", "Plot")

),

mainPanel(

plotOutput("plot")

)

)

)

server <- function(input, output){

v <- reactiveValues(doPlot = FALSE)

observeEvent(input$go, {

*# 0 will be coerced to FALSE*

*# 1+ will be coerced to TRUE*

v$doPlot <- input$go

})

observeEvent(input$tabset, {

v$doPlot <- FALSE

})

output$plot <- renderPlot({

if (v$doPlot == FALSE) return()

isolate({

data <- if (input$tabset == "Uniform") {

runif(input$unifCount, input$unifRange[1], input$unifRange[2])

} else {

rnorm(input$normCount, input$normMean, input$normSd)

}

hist(data)

})

})

}

shinyApp(ui, server)

Shiny Apps Progress Indicator

## **Adding a progress indicator**

The simplest way to add a progress indicator is to put withProgress() inside of the reactive(), observer(), or renderXx() function that contains the long-running computation. In this example, we’ll simulate a long computation by creating an empty data frame and then adding one row to it every 0.1 seconds. (Note that this example is written as a [single-file app](https://shiny.rstudio.com/articles/single-file.html)). To run this, you can copy and paste the code into the R console.)

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

output$plot <- renderPlot({

input$goPlot *# Re-run when button is clicked*

*# Create 0-row data frame which will be used to store data*

dat <- data.frame(x = numeric(0), y = numeric(0))

withProgress(message = 'Making plot', value = 0, {

*# Number of times we'll go through the loop*

n <- 10

**for** (i **in** 1:n) {

*# Each time through the loop, add another row of data. This is*

*# a stand-in for a long-running computation.*

dat <- rbind(dat, data.frame(x = rnorm(1), y = rnorm(1)))

*# Increment the progress bar, and update the detail text.*

incProgress(1/n, detail = paste("Doing part", i))

*# Pause for 0.1 seconds to simulate a long computation.*

Sys.sleep(0.1)

}

})

plot(dat$x, dat$y)

})

}

ui <- shinyUI(basicPage(

plotOutput('plot', width = "300px", height = "300px"),

actionButton('goPlot', 'Go plot')

))

shinyApp(ui = ui, server = server)

This is what will happen:

The withProgress() function is used to start a progress bar, and then the value is incremented with incProgress(). By default, the range of values for the bar goes from 0 to 1, although this can be changed with the min and max arguments.

There are two levels of messages: message, and detail. The message is presented in bold, and the detail is presented in normal-weight text.

In the example above, withProgress() is used inside of renderPlot(), but it could also be used inside of any other render function, like renderTable(), or inside of a reactive().

It’s possible to nest calls to withProgress; if you do this, the second-level progress bar will appear directly under the top-level progress bar, and the second-level text will appear under the top-level text. Further levels of nesting will have a similar pattern.

## **Using a Progress object**

The withProgress() function is a convenient interface around a Progress object. In most cases, it’s simpler and easier to use withProgress, but in some cases, you may need the greater level of control provided by the Progress object. Before we delve into a more complex example, we’ll simply convert the example above from using withProgress to using a Progress object.

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

output$plot <- renderPlot({

input$goPlot *# Re-run when button is clicked*

*# Create 0-row data frame which will be used to store data*

dat <- data.frame(x = numeric(0), y = numeric(0))

*# Create a Progress object*

progress <- shiny::Progress$new()

*# Make sure it closes when we exit this reactive, even if there's an error*

on.exit(progress$close())

progress$set(message = "Making plot", value = 0)

*# Number of times we'll go through the loop*

n <- 10

**for** (i **in** 1:n) {

*# Each time through the loop, add another row of data. This is*

*# a stand-in for a long-running computation.*

dat <- rbind(dat, data.frame(x = rnorm(1), y = rnorm(1)))

*# Increment the progress bar, and update the detail text.*

progress$inc(1/n, detail = paste("Doing part", i))

*# Pause for 0.1 seconds to simulate a long computation.*

Sys.sleep(0.1)

}

plot(dat$x, dat$y)

})

}

ui <- shinyUI(basicPage(

plotOutput('plot', width = "300px", height = "300px"),

actionButton('goPlot', 'Go plot')

))

shinyApp(ui = ui, server = server)

Notice that we need to explicitly create the progress object and make sure that it closes properly, using on.exit().

## **A more complex Progress example**

In the example below, the renderTable() calls out to another function, compute\_data(), to do the long-running computation. If we were to just update the progress indicator before and after compute\_data() were called, then it would only be updated at the beginning, when nothing has been done yet, and at the end, when the computation is completed. In some cases, the best we can do may be to set it to a starting value of, say, 0.3, and then move it to 1 at completion. This may be true if, for example, the function is in an external package.

However, if you do have control over the function doing the computation, you may want to modify it to accept either a Progress object which it will update directly, or to accept a function which it calls each time it does some part of the computation.

In the example below, we’ll take the latter approach. The compute\_data() function accepts an optional updateProgress function, which it calls periodically as it does the computation. The updateProgress function is a closure that captures the Progress object; each time it’s called, it updates the progress indicator.

Again, you can copy and paste this code in your R console to see it in action:

*# This function computes a new data set. It can optionally take a function,*

*# updateProgress, which will be called as each row of data is added.*

compute\_data <- **function**(updateProgress = NULL) {

*# Create 0-row data frame which will be used to store data*

dat <- data.frame(x = numeric(0), y = numeric(0))

**for** (i **in** 1:10) {

Sys.sleep(0.25)

*# Compute new row of data*

new\_row <- data.frame(x = rnorm(1), y = rnorm(1))

*# If we were passed a progress update function, call it*

**if** (is.function(updateProgress)) {

text <- paste0("x:", round(new\_row$x, 2), " y:", round(new\_row$y, 2))

updateProgress(detail = text)

}

*# Add the new row of data*

dat <- rbind(dat, new\_row)

}

dat

}

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

output$table <- renderTable({

input$goTable

*# Create a Progress object*

progress <- shiny::Progress$new()

progress$set(message = "Computing data", value = 0)

*# Close the progress when this reactive exits (even if there's an error)*

on.exit(progress$close())

*# Create a callback function to update progress.*

*# Each time this is called:*

*# - If `value` is NULL, it will move the progress bar 1/5 of the remaining*

*# distance. If non-NULL, it will set the progress to that value.*

*# - It also accepts optional detail text.*

updateProgress <- **function**(value = NULL, detail = NULL) {

**if** (is.null(value)) {

value <- progress$getValue()

value <- value + (progress$getMax() - value) / 5

}

progress$set(value = value, detail = detail)

}

*# Compute the new data, and pass in the updateProgress function so*

*# that it can update the progress indicator.*

compute\_data(updateProgress)

})

}

ui <- shinyUI(basicPage(

tableOutput('table'),

actionButton('goTable', 'Go table')

))

shinyApp(ui = ui, server = server)

It’s possible to use other constructions for the updateProgress function that have different behavior. In the example above, each time updateProgress() is called, the progress bar moves 1/5 of the remaining distance. This tells the user that something is happening, and it’s simple because you don’t need to know ahead of time how many times it’s goingto run. However, it’s not the most accurate representation of progress, since it approaches the end asymptotically, whereas a linear approach would be more accurate.

One alternative is to have the external function call updateProgress() with a specific value. If, for example, the external function knows that it will iterate over the loop 100 times, it could call updateProgress() with value=0.01, then value=0.02, and so on.

Another alternative is to construct a different updateProgress callback, one which increments by a fixed amount each time. To do this, before you call compute\_data(), you must know how many times it will call updateProgress() in the loop. Let’s assume that it will be called 20 times. Then updateProgress could be defined like so:

n <- 20

updateProgress <- **function**(detail = NULL) {

progress$inc(amount = 1/n, detail = detail)

}

Each time this version of updateProgress() is called, it moves the bar 1/20th of the total distance.

## **Old style progress bars**

In Shiny 0.14, the progress bars switched to Shiny’s notification API. However, if you created application that used the old progress bars and had custom styling with CSS, you will need to use the old style output to keep the custom styling. This can be done with by calling withProgress() or Progress$new() with the argument style="old".

**Final Thoughts**

It is great that Shiny now supports Asynchronous programming. It allows applications to be scaled much more easily, especially when long running processes are present. Making use of these features does add some complexity. The final implementation has ~ 3 times more lines of code compared to the first (naive) attempt.

It is less than ideal that the user has to click a button to get the status of the computation. I’d much prefer it if we were able to remove this button and just have a progress bar; however this is currently not possible within Shiny proper, though it might be achievable to inject some kludgy javascript magic to get a progress bar.