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).

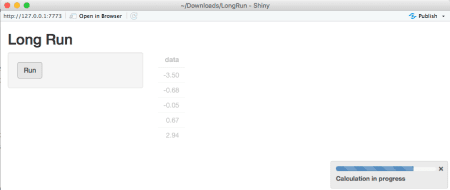
**First Attempt**

Shiny operates in a [reactive programming framework](https://shiny.rstudio.com/articles/reactivity-overview.html). 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](https://shiny.rstudio.com/articles/action-buttons.html) 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](https://shiny.rstudio.com/articles/progress.html) that allow you to do this.

[?](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:

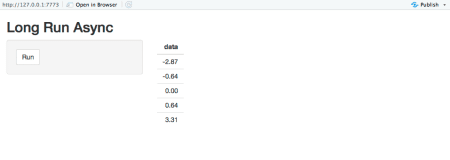
* **If “Run” is clicked multiple times, the analysis is run back to back.** A frustrated user can easily end up having to abort their session because they clicked to many times.
* **There is no way to cancel the calculation.** The session’s UI is locked while the computation takes place. Often a user 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.

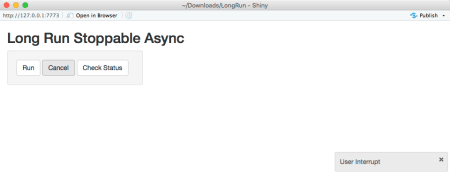
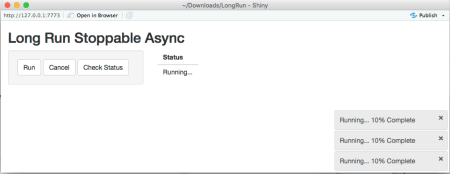
**Third Time Is the Charm**

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.

**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.