Introduction: server.R Slows Down Shiny

It is easy to fall in love with R Shiny's capacity for reactivity. The ability to parametrize every item in the server.R file can be particularly tempting for those with R programming rather than a web app development background. As always, with great power comes great responsibility, and overusing this feature may lead to significant application performance issues. This can lead to a long initial loading time, where elements slowly pop onto the screen one by one. Users might also be forced to wait a few seconds after each click. **Slow performance is a primary reason that users lose interest in an application.** No matter how great your application is, it needs to be fast to be a success!

This huge negative impact on performance might not become apparent until the product reaches maturity and is hard to refactor because the codebase is large and complex. **Adhering to the rules and tips discussed in this article from the very start of your project is the key to developing efficient R Shiny apps whilst taking advantage of the reactivity feature.**

Speed Up R Shiny Performance with updateInput

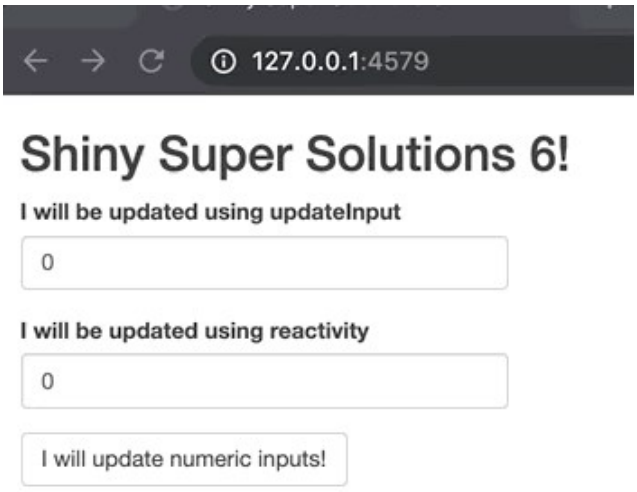
Anyone who has built their first Old Faithful Geyser R Shiny app is familiar with how a Shiny app is split into two parts – *ui* and *server*. Typically, application development begins with having these two worlds separated, i.e. widgets (UI elements) and logic are kept separate in the ui.R and server.R files respectively. However, the developer eventually gets to a point where the UI itself is dependent on the behavior of other widgets, user input, actions, and clicks. There is a temptation to pack everything into the renderUI function and let reactivity do the work. **While tempting, relying too much on the renderUI function will slow down performance.** The developer can speed up the R Shiny application significantly with just a bit more code, as I present in the following example.

Consider these two strategies to deploy two equivalent numeric inputs updated by a button:

```
library(shiny)

ui <- fluidPage(
  titlePanel("Shiny Super Solutions 6!"),
  numericInput(
    "update_input",
    label = "I will be updated using updateInput",
    value = 0,
    min = 0,
    max = 10
  ),
```

```
  uiOutput("render_input"),

  actionButton(
   "click_button",
   label = "I will update numeric inputs!"
  )
)

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

  output$render_input <- renderUI({
    numericInput(
      "render_input",
      label = "I will be updated using reactivity",
      value = if(is.null(input$click_button)) 0 else input$click_button,
      min = 0,
      max = 10
    )
  })

  observeEvent(input$click_button, {
    updateNumericInput(
      session,
      "update_input",
      value = input$click_button
    )
  })
}

shinyApp(ui, server)
```

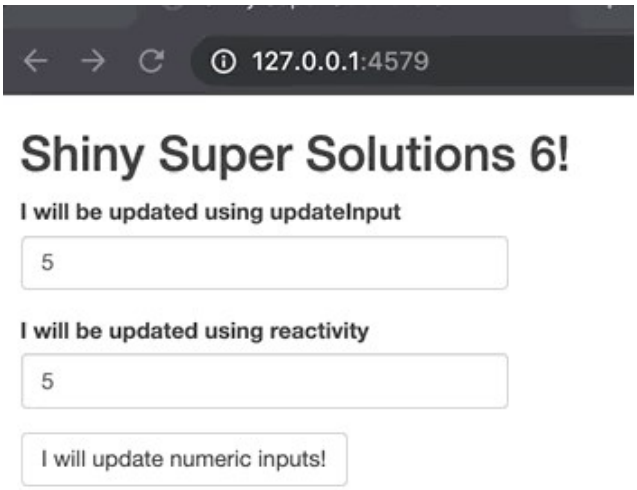
At first glance their resulting behavior is identical:



As long as the application is small, it is difficult to spot any difference in performance between *updateInput* and *renderUI*. Therefore, one may think there is no risk to using *renderUI* in the initial stages of app development…at least until the app grows in size, when performance issues become apparent.

Think about the problem this way: when Shiny spots a difference in the *input*, it goes through all the elements that react to it. The render solution recreates the entire widget, including the label, border and other features. In most cases we are only interested in modifying the value inside, so we should focus on this task only.

Also note what happens when the app starts (I'm refreshing the page in this graphic):



Do you see how the reactivity section pops in and out of existence? **The flickering of the input created by reactivity occurs because the rendering is done on the server rather than in the browser.** First a Shiny app first goes through the UI portion, creates all the features coded there, and then reaches out to the server components. In a real app, handling UI via the server will result in poor UX: after opening the app, the user will see mostly blank spaces that slowly fill in with content one by one, causing a lot of distraction and potentially even confusion. Instead, all widgets should already be there, waiting for the user, and only small portions (such as a value inside of a field) should be left open to modification on the fly.

Enhance R Shiny UI with CSS Classes

It is possible (and even simple) to make use of JavaScript and CSS rules to style the content within an R Shiny app. Once the app is running in the browser, it should be looked at as any other web page from the UI perspective even if there is R code involved. **Under the hood, an R Shiny app is in fact R code transformed into HTML.** Therefore, R Shiny apps can be modified with CSS just like web apps. Using CSS allows you to enhance the style of your app beyond, for instance, the basic “primary-blue, warning-orange, danger-red” options in stock Shiny applications.

> Here’s an open secret: under the hood, an R Shiny app is in fact R code transformed into HTML.

The most efficient way to assign styles are CSS classes. You can use them to e.g., wrap your content in `div(class = ‘myclass’, …)`. Using IDs for the various elements can help with constructing selectors – instructions for the browser that indicate which elements you are interested in modifying.

The ability to simply switch classes further streamlines the style modification process. If you are an R user I would recommend that you check out the excellent *shinyjs* package by Dean Attali. It will help you with web modifications starting at the code level, so the transition into frontend coding will be smooth.

Remember that you need to define classes before you start triggering them on and off. There is a comprehensive tutorial by R Studio on how to implement CSS within a Shiny app, but I would encourage you to go a step further and use SASS. Whilst SASS is beyond the scope of this article, I can recommend a nice overview of SASS by my colleague Pedro Silva. Pedro also has a great primer on getting started with CSS and R Shiny.

Using JavaScript Actions with R Shiny

Because Shiny applications can be treated like any other webpage once they are in the browser, we can also make use of JavaScript code to enhance them. Using JavaScript allows for keeping an action’s logic inside the browser rather than sending the action trigger to the server and slowing down the app. This is beneficial, because there is a resource overhead for passing information between the ui and the server. Therefore, it is best to avoid communicating with the server if there is no need to go beyond the UI code. Consider this equivalent buttons example:

```
library(shiny)
library(shinyjs)
ui <- fluidPage(
  useShinyjs(),
  titlePanel("Shiny Super Solutions 6!"),

  actionButton(
    "js_update",
    label = "I will be updated using javascript",
    icon = icon("arrow-up")
  ),

  uiOutput("shiny_update"),

  actionButton(
    "click_button",
    label = "I will update icons!",
    onclick = "
    $('#js_update > i').toggleClass('fa-arrow-up');
    $('#js_update > i').toggleClass('fa-arrow-down');
    "
  )
)

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

  output$shiny_update <- renderUI({
    icon_name <- if(is.null(input$click_button) || input$click_button %% 2
== 0) {
      "arrow-up"
    } else {
      "arrow-down"
    }

    actionButton(
      "shiny_update",
      label = "I will be updated using reactivity",
      icon = icon(icon_name)
    )
  })
}

shinyApp(ui, server)
```

Both buttons start with the *arrow-up* icon, but they are modified in a very different way.