Execution scheduling

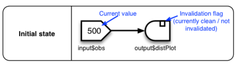
At the core of Shiny is its reactive engine: this is how Shiny knows when to re-execute each component of an application. We'll trace into some examples to get a better understanding of how it works.

A simple example

At an abstract level, we can describe the `01_hello` example as containing one source and one endpoint. When we talk about it more concretely, we can describe it as having one reactive value, `input$obs`, and one reactive observer, `output$distPlot`.

```
shinyServer(function(input, output) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })
})
```

As shown in the diagram below, a reactive value has a value. A reactive observer, on the other hand, doesn't have a value. Instead, it contains an R expression which, when executed, has some side effect (in most cases, this involves sending data to the web browser). But the observer doesn't return a value. Reactive observers have another property: they have a flag that indicates whether they have been *invalidated*. We'll see what that means shortly.

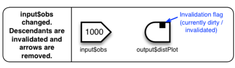


After you load this application in a web page, it be in the state shown above, with `input$obs` having the value 500 (this is set in the `ui.r` file, which isn't shown here). The arrow represents the direction that invalidations will flow. If you change the value to 1000, it triggers a series of events that result in a new image being sent to your browser.

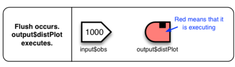
When the value of `input$obs` changes, two things happen:

- All of its descendants in the graph are invalidated. Sometimes for brevity we'll say that an observer is *dirty*, meaning that it is invalidated, or *clean*, meaning that it is *not* invalidated.
- The arrows that have been followed are removed; they are no longer considered descendants, and changing the reactive value again won't have any effect on them. Notice that the arrows are dynamic, not static.

In this case, the only descendant is `output$distPlot`:



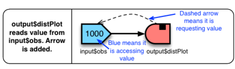
Once all the descendants are invalidated, a *flush* occurs. When this happens, all invalidated observers re-execute.



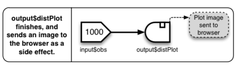
Remember that the code we assigned to `output$distPlot` makes use of `input$obs`:

```
output$distPlot <- renderPlot({
  hist(rnorm(input$obs))
})
```

As `output$distPlot` re-executes, it accesses the reactive value `input$obs`. When it does this, it becomes a dependent of that value, represented by the arrow . When `input$obs` changes, it invalidates all of its children; in this case, that's just`output$distPlot`.



As it finishes executing, `output$distPlot` creates a PNG image file, which is sent to the browser, and finally it is marked as clean (not invalidated).



Now the cycle is complete, and the application is ready to accept input again.

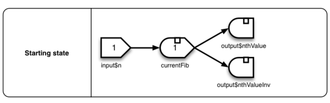
When someone first starts a session with a Shiny application, all of the endpoints start out invalidated, triggering this series of events.

An app with reactive conductors

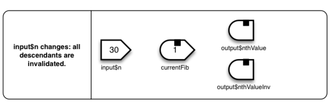
Here's the code for our Fibonacci program:

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

shinyServer(function(input, output) {
  currentFib         <- reactive({ fib(as.numeric(input$n)) })

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

Here's the structure. It's shown in its state after the initial run, with the values and invalidation flags (the starting value for `input$n` is set in `ui.r`, which isn't displayed).

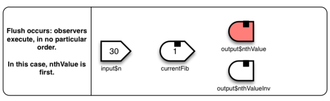


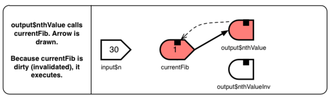
Suppose the user sets `input$n` to 30. This is a new value, so it immediately invalidates its children, `currentFib`, which in turn invalidates its children, `output$nthValue` and `output$nthValueInv`. As the invalidations are made, the invalidation arrows are removed:

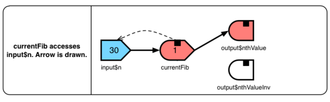


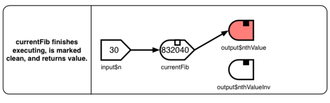
After the invalidations finish, the reactive environment is flushed, so the endpoints re-execute. If a flush occurs when multiple endpoints are invalidated, there isn't a guaranteed order that the endpoints will execute, so `nthValue` may run before `nthValueInv`, or vice versa. The execution order of endpoints will not affect the results, as long as they don't modify and read non-reactive variables (which aren't part of the reactive graph).

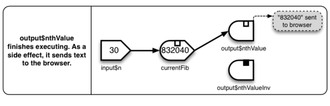
Suppose in this case that `nthValue()` executes first. The next several steps are straightforward:

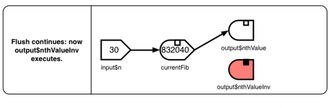








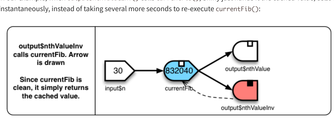




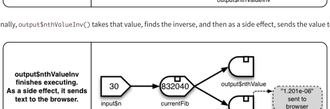
As `output$nthValueInv()` executes, it calls `currentFib()`. If `currentFib()` were an ordinary R expression, it would simply re-execute, taking another several seconds. But it's not an ordinary expression; it's a reactive expression, and it now happens to be marked clean. Because it is clean, Shiny knows that all of `currentFib`'s reactive parents have not changed values since the previous run `currentFib()`. This means that running the function again would simply return the same value as the previous run. (Shiny assumes that the non-reactive objects used by `currentFib()` also have not changed. If, for example, it called `Sys.time()`, then a second run of `currentFib()` could return a different value. If you wanted the changing values of `Sys.time()` to be able to invalidate `currentFib()`, it would have to be wrapped up in an object that acted as a reactive source. If you were to do this, that object would also be added as a node on the reactive graph.)

Acting on this assumption. that clean reactive expressions will return the same value as they did the previous run, Shiny caches the return value when reactive expressions are executed. On subsequent calls to the reactive expression, it simply returns the cached value, without re-executing the expression, as long as it remains clean.

In our example, when `output$nthValueInv()` calls `currentFib()`, Shiny just hands it the cached value, 832040. This happens almost instantaneously, instead of taking several more seconds to re-execute `currentFib()`:



Finally, `output$nthValueInv()` takes that value, finds the inverse, and then as a side effect, sends the value to the browser.



Summary

In this section we've learned about:

- **Invalidation flags**: reactive expressions and observers are invalidated (marked dirty) when their parents change or are invalidated, and they are marked as clean after they re-execute.
- **Arrow creation and removal**: After a parent object follows invalidates its children, the arrows will be removed. New arrows will be created when a reactive object accesses another reactive object.
- **Flush events** trigger the execution of endpoints. Flush events occur whenever the browser sends data to the server.

← Previous Next →