Interactive Visualization

Shiny 2

Dave Armstrong

May 27, 2020

debugging apps

Normally, we could debug functions with `debug()` or `debugonce()` for those who know they're going to get it right the second time.

- NB: if you're not familiar with these functions, they're super helpful for figuring out where things go wrong.

Apps are difficult to debug.

- They can't really be stepped through in the same way a "normal" functions can. You can, however put this in your ui (with the other inputs):

```
actionButton("browser", "browser")
```

and this other piece in your server:

```
  observeEvent(input$browser,{
      browser()
  })
```

debugging apps 2

When you click the "Browser" button in your app, you will be returned to Rstudio inside the app.

- you can see all of the values of the inputs
- you can see all of the contents of reactive objects.
- you can manipulate objects and make new ones.

Let's look at an example.

debugging inputs

Sometimes we meight want to make sure that the inputs are working the way we expect.

- We could do this by printing the values of inputs.
- Look at the `input_debug.r` app.

`reactive()` and `eventReactive()`

- Make sure that you're doing as few calculations as possible.
- `reactive()` re-calculates whenever any of its inputs change.
- `eventReactive(expression, {event})` updates `{event}` only when the expression updates, even if there are inputs in `{event}`.

`isolate()`

The `isolate()` function prevents otherwise reactive elements from updating

- updates when either `x` or `y` update

```
r1 <- reactive({
  input$x*input$y
})
```

- updates when `x` updates, but not when `y` updates

```
r1 <- reactive({
  input$x*isolate(input$y)
})
```

- updates when `y` updates, but not when `x` updates

```
r1 <- reactive({
  input$x*isolate(input$y)
})
```

timing

The `invalidateLater()` function inside a reactive expression will force an update to code that doesn't update itself either automatically or as a result of user inputs.

- in `tweet_app.r` we could use this to update tweets every certain amount of time.

```
rts <- reactive({
  invalidateLater(5000)
  search_tweets(q = "#rstats", n=100, include_rts=FALSE)
})
```

See `tweet_app.r`

timing2

The `debounce()` and `throttle()` functions help limit the rate at which functions update

- `fast_reactive <- reactive({})` updates as often as every 50 ms.
- `throttle(fast_reactive, 2000)` would only update every 2000ms (or 2s).
- `debounce(fast_reactive, 2000)` would only update after `fast_reactive` has stopped updating for 2000ms (or 2s)

action buttons for updates

You could also use an action button to force an update.

- ui:

```
actionButton("updateTweets", "Update Tweets")
```

- server

```
observeEvent(input$updateTweets,
  rts <- search_tweets(q = "#rstats", n=100, include_rts=FALSE)
)
```

see `tweet_app2.r`

walk through

The `{rintrojs}` package gives you the ability to make a walk-through for your app.

- ui

```
actionButton("btn","Walk Me Through")
```

- server

```
 steps <- reactive({
   data.frame(
     element = c("#rowvar + .selectize-control", "#Xtab"),
     intro = c(
       "Choose a variable for the rows of the contingency table - pick one to see what happens. This would be your dependent variable.",
       "When there is no column variable, you see frequency (counts) and relative frequency (percentages) distributions of the selected va
     position = c("right", "right"), stringsAsFactors=FALSE
   )
 })
observeEvent(input$btn,
              introjs(session, options = list(steps=steps())))
```

Make a walk-through for the country app

styling apps

If you're using a shinydashboard setup, they have built-in "skins" - see them here

- there are also some other neat themes here

You could also use the `{bootstraplib}` package to theme any app (dashbord or otherwise). See that documentation here

Try these out on the country app

shiny widgets

The `{shinyWidgets}` package has some "fancier" input controls that you can use.

- They function like normal inputs for the most part.
- See the documentation here

Try these out on the country app

deploying apps

You can deploy your apps to `shinyapps.io`.

- Work through the Getting Started Guide to get your shinyapps.io account set up and configured appropriately.
- deploy one of the apps you've written over the past few days.

You could also deploy your apps to AWS

- You could walk through The Ultimate Guide to Deploying Shiny Apps on AWS
  - benefits - maybe cost, more flexible scaling.

packaging

Pros:

- all dependencies are explicitly stated
- the data travels with the package.
- functions that you write are documented.
- can check code for consistency easily.

Cons:

- Time.