Mastering Shiny

Wickham, H. (2021). Mastering shiny. O'Reilly Media, Inc.

1 Your First Shiny App

For a basic app, create file `app.R` and add

1. `library(shiny)` to load the shiny package
2. `ui <- fluidPage("Hello, world!")` to define the user interface (UI)
3. `server <- function(input, output, session)` to define the server function
4. `shinyApp(ui, server)` to construct and start the application

To run the app, source the document (there are other options).

To stop the document, activate the console window and press *Esc* (there are other options).

2 Basic UI

The UI consists of inputs and outputs.

2.1 Inputs

Insert input controls to UI by adding `{type}Input()` functions to `ui`:

- free text inputs with `textInput()`, `passwortInput()`, `textAreaInput()`
- numeric inputs with `numericInput()`, `sliderInput()`
- dates with `dateInput()`, `dateRangeInput()`
- limited choices with `selectInput()`, `radioButtons()`, `checkboxGroupInput()`
- file uploads with `fileInput()`
- action buttons with `actionButton()`, `actionLink()`

All input functions have the same first argument `inputId`. If some input function has ID `name`, than the input can be accessed in the server with `input$name`. Input functions have additional (unique) arguments to adjust their appearance.

2.2 Outputs

Outputs in the UI create placeholders that are later filled by the server function. Like inputs, their first argument is always an ID `outputId`. If some output function has ID `plot`, than the output can be accessed in the server with `output$plot`.

Insert output placeholders to UI by adding `{type}Output()` functions to `ui`. Each `{type}Output()` function is coupled with a `render{Type}` function in `server`:

- text outputs with `textOutput()` (`renderText()`),
- R code output with `verbatimTextOutput()` (`renderPrint()`)
- static tables with `tableOutput()` (`renderTable()`)
- dynamic tables with `dataTableOutput()` (`renderDataTable()`)
- plots with `plotOutput()` (`renderPlot()`)

3 Basic Reactivity

To connect inputs with outputs, Shiny uses a concept called *reactive expressions*.

3.1 Reactive Expressions

Reactive expressions mean: when an input changes, all related outputs automatically update. Shiny knows when the update should be run. The code `output$greeting <- renderText(paste("hi",input$name))` informs Shiny how it could update the greeting if it needs to (e.g., if `input$name` changed). With this concept, code is no longer executed from top to bottom, but follows a graph of dependencies, which describes how inputs and outputs are connected.

3.2 Modularity

However, Shiny updates outputs always as a whole. This can lead to undesired effects:

```
output$plot <- renderPlot({
  x <- rnorm(n = input$n)
  plot(x, xlim = input$range)
})
```

The random vector `x` is drawn again, when `input$range` changes. Better put the computation of `x` into a separate reactive environment. Now the value of `x` must be accessed via `x()`:

```
x <- reactive(rnorm(n = input$n))
output$plot <- renderPlot(plot(x(), xlim = input$range))
```

If `x` should be drawn after an event (e.g., the user clicked a button), but not when `input$n` changes, use `x <- eventReactive(input$simulate, rnorm(n = input$n))`.

4 Case Study: ER Injuries

Development of a richer Shiny app with the concepts seen so far. Demo Source

5 Workflow

> One of the reasons that I've been able to accomplish so much is that I devote time to analysing and improving my workflow. I highly encourage you to do the same! – Hadley Wickham

5.1 Development

- Type *shinyapp* in .R file to insert Shiny app snippet
- Keyboard shortcut to run the app: *Ctrl+Shift+Enter*
- Relaunch app after every save with background job:
  1. add script `shiny-run.R` to folder with `app.R`:

```
options(shiny.autoreload = TRUE)
shiny::runApp()
```

  2. with active `shiny-run.R`, RStudio > Tools > Background Jobs > Start Background Job
  3. copy URL from Jobs pane and run `rstudioapi::viewer("<URL>")`

5.2 Debugging

- Shiny automatically prints the traceback to the console
- Use interactive debugger with `browser()` in source
- Use `message()` (with `glue::glue()`) or `str()` calls to understand when a part of the code is evaluated and to show values
- Getting help: make a reprex (minimal reproducible example)

6 Layout, Themes, HTML

7 Graphics

interactive graphics `plotOutput("id", click = "plot_click")` in ui makes coordinates `input$plot_click` available in server

use `req()` to avoid app action before user input

use `nearPoints(<dataset>, input$plot_click)` to get points near to the click

can also use `dblclick`, `hover`, and `brush` (together with `brushedPoints()` helper) argument

can modify a plot interactively with `reactiveVal()`

8 User Feedback

9 Uploads and Downloads

10 Dynamic UI

11 Bookmarking

12 Tidy Evaluation