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()`)
- images with `imageOutput()` (`renderImage()`)

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 and use *Tab* 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 `message()`, `str()` calls to understand when a part of the code is evaluated and to show values.
- To access values of reactive expressions, use interactive debugger with `browser()` in source.
- Getting help: first make a reprex (*minimal reproducible* example).

6 Layout, Themes, HTML

The input and output functions just return HTML, which the developer can create themself, but Shiny offers helper functions. The `fluidPage()` provides basic HTML for the UI (alternatives are `fixedPage()` and `fillPage()`). Other functions modify the visual appearance:

- App with sidebar (most common app design):

```
ui <- fluidPage(
  titlePanel(
    # app title/description
  ),
  sidebarLayout(
    sidebarPanel(
      # inputs
    ),
    mainPanel(
      # outputs
    )
  )
)
```

- Multiple rows (number of columns should add up to 12):

```
ui <- fluidPage(
  fluidRow(
    column(4,
      ...
    ),
    column(8,
      ...
    )
  )
)
```

- Tabsets (for multipage layout, alternatives are Navlists and Navbars):

```
ui <- fluidPage(
  tabsetPanel(
    tabPanel("Import data",
      ...
    ),
    tabPanel("Visualize data",
      ...
    )
  )
)
```

Shiny uses the Bootstrap CSS and javascript style framework. The style can be modified via `theme = bslib::bs_theme()` as argument to `fluidPage()`. Use `bslib::bs_theme_preview(theme)` to preview the theme. Use `thematic::thematic_shiny()` inside `server()` adapt the theme to plots.

7 Graphics

`plotOutput()` can serve as an input that tracks pointer events, leading to interactive plots. Add the following arguments:

- click: `plotOutput("id", click = "plot_click")` makes coordinates `input$plot_click` available in `server()` (e.g., use `nearPoints(<dataset>, input$plot_click)` to get data points near to the click)
- double-click: `plotOutput("id", dblclick = "plot_click")` (similar to `click`)
- hover: `plotOutput("id", hover = "plot_click")` (similar to `click`)
- rectangular selection tool: `plotOutput("id", brush = "plot_brush")` makes coordinates `input$plot_brush` available (e.g., use `brushedPoints(<dataset>, input$plot_brush)` to get the selected data points)

8 User Feedback

Thoughtful communication about what's happening with the app can have a huge impact on the quality of the user experience.

8.1 Validation

Use the {shinyFeedback} package to put feedback next to an invalid input:

```
ui <- fluidPage(
  shiny::useShinyFeedback(),
  numericInput("n", "n", value = 10)
)
server <- function(input, output, session) {
  square_root <- reactive({
    shinyFeedback::feedbackWarning("n", input$n < 0, "Please select a positive number.")
    req(input$n >= 0)
    sqrt(input$n)
  })
}
```

The function `req()` stops bad or missing inputs from triggering reactive changes. Alternatives to `feedbackWarning()` are `feedback()`, `feedbackDanger()`, and `feedbackSuccess()`. Alternative to {shinyFeedback} is the Shiny built-in function `validate()`.

8.2 Notifications

To let the user know what's happening, display a notification via `showNotification("<text>")` inside `server()`. It can be transient, removed on completion, or progressively updated.

8.3 Progress Bars

Progress bars help the user to estimate the time to completion. Inside `server()`:

```
eventReactive(input$go, {
  withProgress(message = "Computing random number", {
    for (i in 1:steps) {
      x <- function_that_takes_a_long_time(x)
      incProgress(1 / steps)
    }
  })
})
```

The `eventReactive()` environment is not required, but it is good practice to allow the user to control when the time consuming task starts. The {waiter} package offers more visual options (like, e.g., spinners).

8.4 Dialoges

Create dialog boxes with `modalDialog()` to get, e.g., explicit confirmation from user for a potentially destructive action. Use `showModal()` and `removeModal()` to show and remove the dialog, respectively.