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.

9 Uploads and Downloads

To upload files, use `fileInput("id")` in the UI, which makes the data frame `input$id` of file information available in `server()` (it is initialized with `NULL`, so use `req(input$id)` to make sure the code waits for the first upload).

To download files, use `downloadButton("id")` in the UI. Unlike other outputs, it is not paired with a render function, but with `downloadHandler()`. There is the option to create downloadable reports via (parameterized) RMarkdown documents via using `rmarkdown::render()` for the `content` argument of `downloadHandler()`.

10 Dynamic UI

Every input control `{type}Input()` is paired with an update function `updateTextInput()` that allows to modify any UI control from `server()` after it has been created, e.g., to reset parameters back to their initial values, or to hierarchically create select boxes. There are two issues:

- Modifications of UI take some time, use `freezeReactiveValue()` to tell all downstream calculations that an input value is currently stale to avoid temporal bad states.
- Be aware of circular references.

To selectively show and hide parts of the UI from the server, use `tabsetPanel("id", type = "hidden")` (together with `updateTabsetPanel("id")`). This makes a *wizard interface* possible.

Even more flexible, the UI can be completely recreated in response to user action using `uiOutput("id")` in `ui` paired with `output$id <- renderUI()` in `server()`. This is most effective combined with functional programming, e.g., with {purrr}.

11 Bookmarking

Bookmarks allow to save and share the state of an app. For this, do:

1. add `bookmarkButton()` to `ui`
2. turn `ui` into a function (`ui <- function(request) {<ui code>}`)
3. add `enableBookmarking = "url"` to `shinyApp()` call
4. *optional:* make random process reproducible with `repeatable()`
5. *optional:* add `setBookmarkExclude()` to avoid bookmarking certain inputs

The generated URL stores the input states. Alternatives are:

1. automatically update URL in the browser by adding this chunk to `server()`:

```
observe({
  reactiveValuesToList(input)
  session$doBookmark()
})
onBookmarked(updateQueryString)
```

2. set `enableBookmarking = "server"` to save app state as *.rds* file on the server

12 Tidy Evaluation

There are two issues when using functions from {tidyverse} in (Shiny) programming:

- Data masking: Variables in the applied data frame are called without any extra syntax, hence cannot use (user input) variables directly, but via `data_frame %>% filter(.data[[input$var]] > .env$input$min)`, where `.data` makes clear that `input$var` lives inside `data_frame` and `.env` that `input$var` is an environment variable.
- Tidy selection: Columns in the applied data frame can be selected by position, name, or type. To refer to variables indirectly, use `any_of()`, `across()`, etc.

13 Why Reactivity?

- We need expressions and outputs to update if and only if their inputs change (1. stay in sync and 2. do minimal work).
- Solution: Reactive Expressions (1. are lazy and 2. are cached)
- History: first use in spreadsheats (cells update automatically through formula dependencies), now reactive programming dominates web programming

14 The Reactive Graph

1. Session begins: no connections exist, reactive expressions are invalidated, reactive inputs are ready.
2. Execution begins: Shiny picks randomly an invalidated output and executes it.
3. If output needs value of reactive expression, expression is executed. Expression records connection. If output needs reactive input, the input can be returned immediately.
4. Recursive progress until every reactive expression completes and every output executes. Then session is at rest and waits for input change.
5. If an input changes, invalidate the input, notify the dependencies, and remove the existing connection. (This implies that the reactive graph is dynamic and can change while the app runs, depending on input values which might define connections.) Then execute invalidated outputs again.

The {reactlog} package can draw the reactive graph of an app automatically: - Before starting the app, run `reactlog::reactlog_enable()`. - After the app has closed, run `shiny::reactlogShow()`.

15 Reactive Building Blocks

There are three building blocks of reactive programming: reactive values, reactive expressions, and observers (including outputs).

- Reactive Values:
  - most come from `input` argument to `server()`
  - can create own: `reactiveVal()` (single reactive value) or `reactiveValues()` (list)
  - both types have reference semantics (not copy-on-modify)
- Reactive Expressions:
  - wrapped inside `reactive()`
  - they cache errors (will be displayed in the console and terminate session)
  - reactive expressions behave like functions, so function that only work inside functions can be used (e.g., `on.exit()`)
- Observers and Outputs:
  - terminal nodes in the reactive graph, eager and forgetful
  - created via `observe()` (low-level) or `observeEvent()` (user-friendly)
  - to create output, assign `output$value <- ...`

15.1 Isolating Code

- `isolate()` allows to access the current value of a reactive (value or expression) without taking a dependency on it
- `observeEvent(x, y)` is equivalent to `observe({x; isolate(y)})` (`eventReactive()` is the same for reactives)

15.2 Timed Invalidation

- `invalidateLater(ms)` invalidates any reactive consumer after `ms` milliseconds (time is a minimum and can be larger)
- `reactivePoll()` (or `reactiveFileReader()`) for downloading data every time interval (with check if data actually changed)

16 Escaping the Graph

- one can combine `reactiveVal()` and `observeEvent()` to connect the right-hand side of the reactive graph back to the left-hand side
- technically, when calling an `update*()` function or modifying a reactive, no reactive dependency is created between reactive value and observer
- sometimes these techniques are required to solve advanced problems, but should be used sparsely

17 General Guidelines

- challenge: keeping a complex and growing code-base organized, stable, and maintainable
- code organization: clear variable and function names, comments where needed, no code repetition, code isolation
- testing: automated test plan that grows over time and runs after each change
- dependency management: create reproducible R environment with {renv} or track dependencies with {config}
- source code management: use Git for version control, paired with GitHub for code sharing and collaboration
- continuous integration/deployment: automatic and perpetual validation of code changes (e.g., via GitHub actions)
- code reviews: many benefits from having someone else review code before integration

18 Functions

- breaking app code into functions
  1. reduces duplication (more efficient coding)
  2. makes debugging easier
  3. allows code isolation
- file organization:
  - large functions in `R/{function-name}.R`
  - collect small functions in `R/utils.R`
- Try to keep reactive and nonreactive parts as separate as possible.

19 Shiny Modules

Modules can extract code that spans both UI and server into reusable components to simplify complex apps:

- composed of module UI and module server (both functions)
  - put UI code inside a function that has an `id` argument and wrap each existing ID `var` in a call `NS(id, "var")` (creates module namespace)
  - wrap server function inside another function that has an `id` argument and calls `moduleServer()` with the `id` and a function that looks like a regular server function
  - good practice to write function that uses both modules to create app
- module servers can receive additional arguments (reactive or constant) and can return (list of) values that should be reactive

20 Packages

Shiny Apps can be organized in packages:

1. Put all R code in the `R/` directory. Pulling modules in separate files is useful.
2. Write a function that starts the app.
3. Create a `DESCRIPTION` file.

A package structure provides an easy workflow to accurately reload and relaunch the app.

21 Testing

- Testing non-reactive functions is easily automated via unit testing with {testthat} package.
- To test HTML code from extracted UI code, use snapshot tests.
- Use code coverage to see what proportion of code is covered by tests.
- Test reactives with `shiny::testServer()`. This is a (limited) simulation of an app. Works similar with modules. However, time does not advance automatically. Also, it ignores the UI (e.g., can't test the `update*()` functions because they work with JavaScript). Any code that relies on a real browser running will not work.
- The {shinytest} package allows testing user interaction with an app (downside: quite slow).
- To test visuals, you can take and compare screenshots. However, this is the most fragile technique.

22 Security

Two main things to protect:

1. Your personally identifying information, passwords: never include them in source code, never attempt to roll user authentication yourself, be aware that inputs are validated on the client-side.
2. Your compute resources: combination of `parse()` and `eval()` can make app vulnerable, and it is hidden in many places (model formulas, {glue} package, variable transformations to {dplyr} and {ggplot2}).

23 Performance

- Metaphor: Each customer (user) comes into the restaurant (server) and makes an order (request), which is then prepared by a chef (R process).
- Benchmarking with {shinyloadtest} and `shinycannon` (Java) to check the performance of an app with multiple users.
- Profile code with {profvis} to find bottleneck. It produces a flame graph (x-axis shows elapsed time and y-axis shows depth of call stack) with `profvis::profvis(f())`.
- Use `bindCache()` to cache reactives, {memoise} provides caching for regular R functions.