Making interactive web-apps in R using Shiny

Stuart Lacy

2022-03-02

Introduction

- Allow others to explore the output of your R code through a web browser!
- Learning objective: understand what Shiny is, the potential uses it has for research, and be able to create a basic app after this talk
- Assumes you have some familiarity with R, but no experience with web development is required
- Follow along with the examples by cloning this repository within RStudio:
- File -> New Project -> Version Control -> Git -> Repository URL: `https://github.com/stulacy/shiny-introduction.git`

What is Shiny?

- An R package for creating web-apps
- Developed by the RStudio/tidyverse team
- Designed with a focus on data visualisation

Why would I want to use it?

- Very quick development time for initial proof of concepts
- Provide means for non-R users to access your work
- Don't need to know any HTML/CSS/JS, but can use those skills for customisation
- Clean and modern looking default appearance
- Have access to R's extensive libraries
- Simple hosting options

Possible use cases

- Interactively visualize and explore datasets (rather than generating PDFs with hundreds of figures)
- Monitor live data streams
- Provide tools to accompany published research (either methods, models, or data)
- Establish an online presence
- Provide visual interface to R code (i.e. automatic report generation)
- Examples:
  - COVID-19 dashboard
  - Genome browser
  - MRI Visualization
  - Lego Mosaic Creator

Shiny code layout

What does Shiny code look like?

- Shiny apps are organised into 2 files:
- *ui.R* (**U**ser **I**nterface)
  - Defines the appearance of the app to end users (frontend)
  - Every UI is composed of multiple "widgets"
  - Widgets can be outputs such as plots, tables, text
  - And inputs such as buttons, sliders, text fields, etc...

What does Shiny code look like?

- Shiny apps are organised into 2 files:
- *ui.R* (**U**ser **I**nterface)
  - Defines the appearance of the app to end users (frontend)
  - Every UI is composed of multiple "widgets"
  - Widgets can be outputs such as plots, tables, text
  - And inputs such as buttons, sliders, text fields, etc...
- *server.R*
  - Defines all backend logic needed to process incoming inputs and generate the required outputs
  - Majority of your R code lives here

What does Shiny code look like?

- Shiny apps are organised into 2 files:
- *ui.R* (**U**ser **I**nterface)
  - Defines the appearance of the app to end users (frontend)
  - Every UI is composed of multiple "widgets"
  - Widgets can be outputs such as plots, tables, text
  - And inputs such as buttons, sliders, text fields, etc...
- *server.R*
  - Defines all backend logic needed to process incoming inputs and generate the required outputs
  - Majority of your R code lives here
- Input widgets are defined by `xInput` function calls in the UI
- Output widgets are referenced in both the UI (`xOutput` functions) and in the server (`renderX`)

Example 1: Basic working example

- `examples/1_basic/`
- Example application with a single plot and table
- No interactivity at all
- Can look at the code either on GitHub or can clone it to your PC
  https://github.com/stulacy/shiny-introduction
- Can run the app locally either by opening the project in RStudio, or can simply run `shiny::runGitHub(repo="shiny-introduction", username="stulacy", subdir="examples/1_basic")`

Example 2: Interactive output elements

- There are many options for interactive output widgets:
  - `plotly`: general plotting library with controls
  - `DataTables`: interactive tables
  - `leaflet`: maps
  - `networkD3`: network diagrams
  - `diagrammeR`: graphs and flowcharts
  - `r2d3`: Interface to D3 Javascript plotting library
- `examples/2_interactive_widgets`
- The same layout as example 1, but with an interactive plot and table

Reactivity

Example 3: Reactive user inputs

- The main form of adding interactivity in Shiny apps is through reactive connections between the UI and Server
- Input UI elements are referenced in `server.R` through the `input` object, e.g. the current value of a dropdown menu
- Whenever this value changes due to user input, any output widgets that reference it will re-evaluate and pass their new output (plot, table etc…) back to the UI
- `examples/3_reactivity`

Example 4: Multiple reactive dependencies

- There is a many:many relationship between input and outputs. Each `input` can be used in multiple outputs (as seen in previous example), and likewise each `output` can reference multiple `inputs`
- There are lots of possible input widgets:
  - Buttons
  - Checkboxes
  - Date selection/range
  - File upload
  - File download
  - Text input
  - Radio buttons
  - Dropdown menu
  - Sliders
- `examples/4_multiple_reactives`

Example 5: Custom reactive objects

- So far we've seen that `input` objects are reactive and will trigger a change in any downstream output that uses them
- You can also create your own reactive elements in a variety of ways
- Be careful when there are lots of reactive elements, it can make your code hard to follow and your app possibly slow!
- `examples/5_custom_reactivity`

Reactivity chaining

Example 4

- User selects transform or variable
- Simultaneously:
  - Scatter plot transforms selected variable and is redrawn
  - Density plot transforms selected variable and is redrawn

Example 5

- User selects transform or variable
- *transformed_df* transforms selected variable
- Simultaneously:
  - Scatter plot is redrawn
  - Density plot is redrawn

More reactive objects

- Alternatively can use the `observe` and `reactiveValues` paradigm, useful when have multiple reactive items or when the reactive object has stateful characteristics
- This can be a complex topic, so refer to the Shiny documentation for more details

```
updated_vals <- reactiveValues()
observe({
  # Observe is triggered when any reactive expression contained within triggers
  # Unlike reactive, it doesn't return anything
  raw_val <- iris[[input$xvar]]
  funcs <- list("None"=identity, "Squared"=function(x) x**2, "Log"=log)

  transformed <- funcs[[input$transform]](raw_val)
  label <- sprintf("Displaying variable %s with transform %s", input$xvar, input$transform)
  updated_vals$data <- transformed
  updated_vals$label <- label
})
```

Customising your app's appearance

Structuring UI

- By default all Shiny apps are responsive, i.e. they adapt their layout to the size of the viewing device
- To add more structure to an app, you can use a `sidebarLayout` to separate inputs from outputs
- You can partition your app further with tabbed output using `tablistPanel`
- You can create entirely independent pages with `navbarPage`

Example 6: UI Sidebar

- `examples/6_ui_sidebar`
- Shows an example of adding a sidebar to an app
- Useful for separating user controls from the resultant outputs

Example 7: UI Tabs

- `examples/7_ui_tabs`
- Shows how separate UIs can be accessed through tabs

Example 8: UI Pages

- `examples/8_ui_pages`
- When a stronger distinction between different UIs is required, pages can be used instead of tabs

Further visual customisation

- You can organise elements into rows (`fluidRow`) and columns (`column`)
- The bslib package provides different themes if you get bored of the default
- shinydashboard package provides new UI elements for creating dashboard style displays
- Can add CSS to have fine control over each element's appearance
- Add flourishes with Javascript (`shinyjs` package provides some useful features such as loading spinners)

Example 9: Shinydashboard

- `examples/9_shinydashboard`
- Shows how the example 8 app looks under the `shinydashboard` package
- The UI elements provided by `shinydashboard` are designed to get a quick dashboard up and running in a more structured manner than using the default UI

Other considerations

Organising larger apps

- You can dynamically create UI elements in the server end using `renderUI` and `uiOutput`
- Code can start to get disorganised with larger multi-page apps - recommend putting each page into its own files and using `source` to load them in (example here) or using modules
- If your program is running slower than expected, have a look at the reactive dependencies and see if there is any redundancy
- Bear in mind that each time a plot is updated through user interaction, a new plot needs to be generated and downloaded from the server - can be CPU intensive if done very frequently or a big dataset!

Hosting:

- University provide free hosting at shiny.york.ac.uk
- 2 methods:
  - Managed: provide access to a GitHub repo and it will automatically update whenever there is a change to the main branch
  - Self-hosted: you are given access to a folder where you can put your app and any dependencies/data you need
- Email `itsupport@york.ac.uk` for access
- NB: not designed for a large number of concurrent users
- Shinyapps.io has free hosting for up to 5 apps provided by RStudio team

Data storage:

- Several options for storing data
- CSV/other structured files
  - On same machine
  - Or remote (through AWS, Google Drive, or even Google Sheets)
- Database on same machine (SQLite) or remote (Postgres/MySQL/SQLServer etc…)
- Databases are useful for big datasets as they only load the requested subsets into memory, rather than having to read the full file
- Always worth doing as much pre-processing as possible offline!

Similar alternatives

- Dash (Python)
- Observable (Javascript/D3)
- Tableau (drag and drop, commercial)
- Grafana (drag and drop, open-source)

Conclusions

- Biggest strength: Shiny makes it easy to get apps up and running with a wide range of features to cover most use cases
- Biggest weakness: For larger apps, the code can become hard to navigate, particularly if you start adding in custom JS/HTML/CSS
- If you need support come along to a Research Coding Club drop-in session or ask on our slack channel `#research-coding-club`. Join the mailing list at https://researchcodingclub.github.io/