Why run Shiny locally

There are a couple of reasons why running Shiny apps locally is necessary. First and foremost, that is how you can **test the app** if you are the developer. Of course testing goes way beyond just opening up the app, read more about best practices for engineering Shiny apps in this excellent book written by ThinkR folks.

The source code for the app can be **shared** with collaborators, clients, and users. They can run the app themselves if they are R savvy enough. When the audiences of a Shiny app are R users, it makes sense to share the app as a Gist, GitHub repository, or a zip file. However, sharing Shiny apps this way leaves installing dependencies up to the user.

Distributing Shiny apps as part of an R package takes care of dependency management. Putting your **Shiny app inside an R package** is especially useful when the app is used to augment the command line capabilities of the package. In such cases, the Shiny apps are often included in functions, i.e. `somepackage::run_app()`.

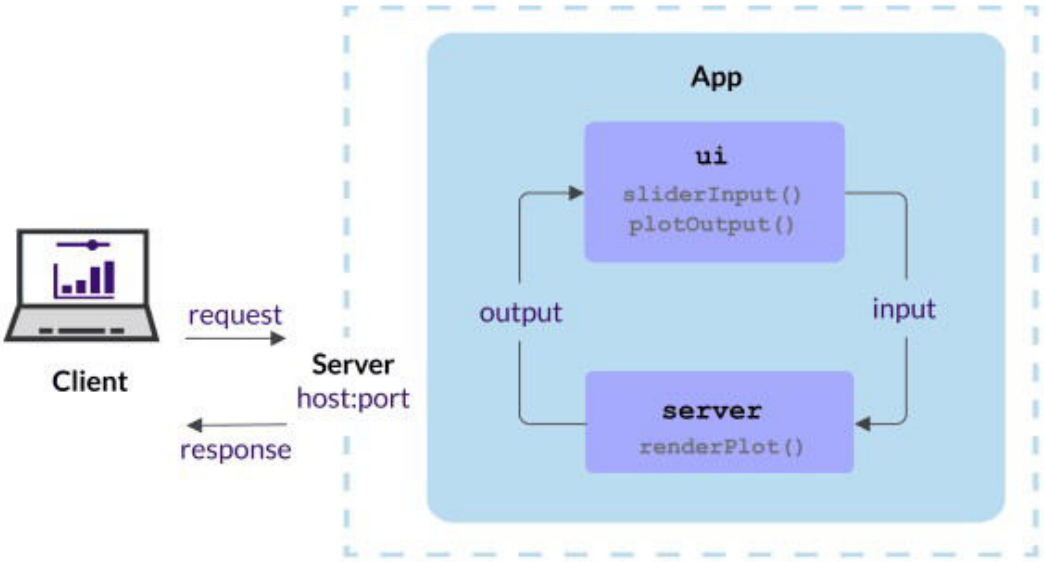
The most important reason for reviewing how to run Shiny apps locally is that the practices that let you run the Shiny app locally are the same practices you can use when **deploying your apps to remote servers**. So let's review the options!

How to run Shiny locally

In a previous post you have seen how a Shiny app is structured. Besides being just an HTML file at the end, it requires the websocket server in the background to constantly update the application state when the user interacts with the web page. Therefore, we cannot just copy the HTML output to a server, like we would with a static website.

> The Anatomy of a Shiny Application
> Shiny lets you quickly build web applications using the R programming language.In this post I will walk you through how a Shiny application is structured. Thegoal is simply to have the most basic and dependency free app that we candeploy. Let’s get going! What is R and ShinyR [https://www.r-project.org/…

Hosting Data AppsPeter Solymos

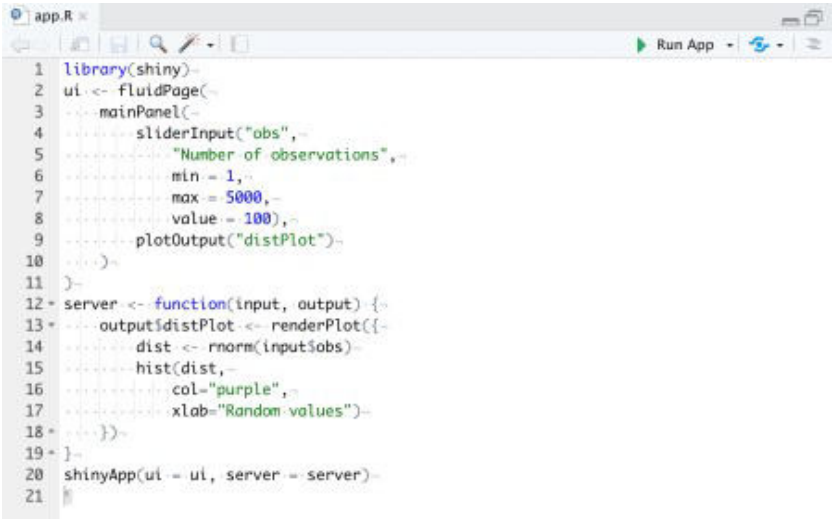


Single file

Let's take the following file named `app.R`:

```
library(shiny)
ui <- fluidPage(
    mainPanel(
        sliderInput("obs",
            "Number of observations",
            min = 1,
            max = 5000,
            value = 100),
        plotOutput("distPlot")
    )
)
server <- function(input, output) {
    output$distPlot <- renderPlot({
        dist <- rnorm(input$obs)
        hist(dist,
            col="purple",
            xlab="Random values")
    })
}
shinyApp(ui = ui, server = server)
```

If you are using the RStudio IDE, you should see the **Run App** button:



Run Shiny app from RStudio IDE

Clicking on the button would run the app. If you inspect the console output, you should see something like this:

```
> runApp('~/your/path')

Listening on http://127.0.0.1:3884
```

What does this mean? Pushing the Run App button led to running the `runApp()` command. This started the local server (127.0.0.1 is also called the local host) listening on port 3884 (your port number might be different). If you visit the http://127.0.0.1:3884 address in your browser, you should see the Shiny app with the slider and the histogram. (Stop the app by closing the app window or using Ctrl+C).

The `runApp` function can take different types of arguments to run the same app. What you saw above was serving the app from a **directory** containing `app.R`. If you name the single file something else, e.g. `my-app.R`, you can provide the **path to a single file** as `runApp('~/your/path/my-app.R')`.

Multiple files

If your app is a bit more complex, you might have **multiple files in the same directory**. You can use the `runApp('~/your/path')` method to point to such a directory that contains `server.R` file and `ui.R` (or a `www` directory that contains the file `index.html`). You can save the parts of the simple app into three files.

The `global.R` file is used to load packages, data sets, set variables, or define functions that are available globally:

```
# global.R
library(shiny)
```

The `server.R` file defines the `server` function which is called once for every R process:

```
# server.R
server <- function(input, output) {
    output$distPlot <- renderPlot({
        dist <- rnorm(input$obs)
        hist(dist,
             col="purple",
             xlab="Random values")
    })
}
```

The `ui.R` file defines the `ui` object:

```
# ui.R
ui <- fluidPage(
    mainPanel(
        sliderInput("obs",
            "Number of observations",
            min = 1,
            max = 5000,
            value = 100),
        plotOutput("distPlot")
    )
)
```

Programmatic cases

If you want to run the Shiny app as **part of another function**, you can supply a **list** with `ui` and `server` components (i.e. `shinyApp(list(ui = ui, server = server))`) or a Shiny **app object** created by the `shinyApp` function (i.e. `shinyApp(shinyApp(ui, server))`).

Note that when `shinyApp` is used at the R console, the Shiny app object is automatically passed to the `print()` function, which runs the app. If `shinyApp` is called in the middle of a function, the value will not be passed to `print()` and the app will not be run. That is why you have to run the app using `runApp()`:

```
run_app <- function() {
  runApp(
    shinyApp(
      ui = fluidPage(
        mainPanel(
          sliderInput("obs",
            "Number of observations",
            min = 1,
            max = 5000,
            value = 100),
          plotOutput("distPlot")
        )
      ),
      server = function(input, output) {
        output$distPlot <- renderPlot({
          dist <- rnorm(input$obs)
          hist(dist,
            col="purple",
            xlab="Random values")
        })
      }
    )
  )
}
```

Start the app by typing `run_app()` into the console.

Which option to choose

The programmatic use case is best when the app is to be served from a function by a user running R interactively. Although it is possible to use such functions in non-interactive mode, for example as part of Docker based deployments, it requires additional work.

The choice between singe vs. multiple files comes down to personal **preference** and the **complexity** of the shiny app. You might start with a single file, but as the file gets larger, you might decide to save the

pieces into their own files. You might also add other files to the `www` folder etc.

Keeping Shiny apps **in their own folder** is generally a good idea irrespective of having single or multiple files in the folder. Changing your mind later won't affect how you run the app.