Q-STEP R 'HOW TO' GUIDES:

BUILDING APPS IN SHINY (R)

Creator: Dr Patrick English

Building applications might seem like a very high-end programming level skill, and something which requires knowledge of expert coding languages such as Java, HTML, or C+. However, the Shiny package for R makes developing and publishing our own applications very simple, and only requires a basic knowledge of R to use.

Of course, the greater one's knowledge in R, the more complex apps they can build! This guide however focuses on creating a simple app which everyone should be able to follow and construct. It is split into the following sections:

1. What are Shiny Apps and why might we want to develop one?
2. Introducing Shiny
3. Constructing an example Shiny App
4. Publishing Shiny Apps on Shinyapp.io

This guide assumes a basic competency in in R from the start – users should already be comfortable with assigning and calling objects, and much of the logic is similar to working with functions (though this is not necessary prerequisite). I work with R Studio throughout this piece.

Q-Step
A step-change in quantitative social science skills
Funded by the Nuffield Foundation, ESRC and HEFCE
Nuffield Foundation
Sheffield Methods Institute.
The University Of Sheffield.

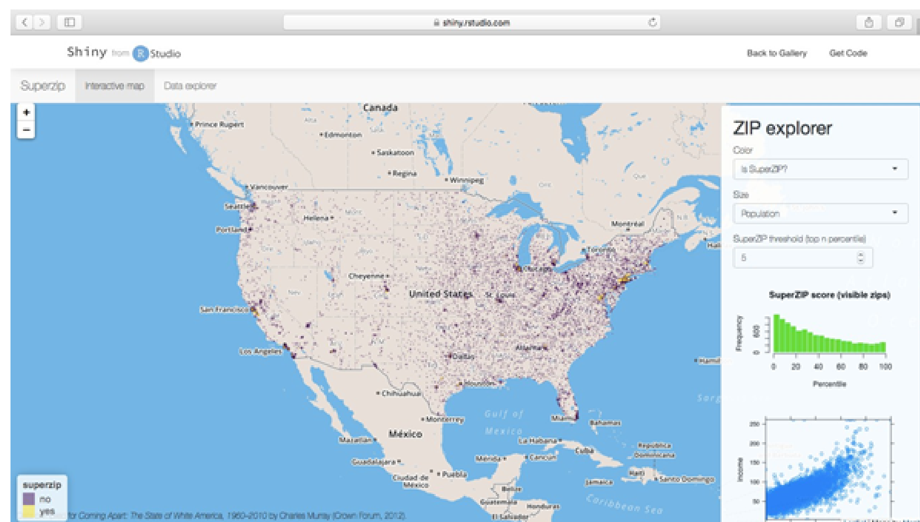
1. What are Shiny Apps and why might we want to develop one?

In the words its developers, Shiny: “is an R package that makes it easy to build interactive web apps straight from R.” Essentially, Shiny allows users to construct and publish good quality web applications (even full websites) using only R coding and knowledge – no need to engage in traditional site building languages such as HTML, Java, or C++.

Shiny uses a three-step process to develop applications. A ‘user interface’ (UI) which sends information to a ‘server’ space, with the two then threaded together into an application using the shinyApp() function. Within the boundaries of the UI and server framework, anything which is possible in R is possible to turn into a web application.

Shiny has been used to develop some truly stunning web applications, making full use of the amazing range of packages and flexibility R offers to produce brilliant results. The best place to see the range of possibilities Shiny can offer is the Shiny Gallery. Below are a few examples which highlight how useful Shiny can be.

i) SuperZip



SuperZip is a Shiny App which allows users to look in detail at a huge range of socio-economic data at the ‘zip’ (postcode) level in the United States. We can look at things such as college education rates, population, and income across different areas of the US and directly compare them using colour and plot-sizes. SuperZip is an example of Shiny in combination of the Leaflet package, which allows for the construction of detailed interactive maps.

ii) Movie explorer

The movie explorer app looks at the relationship between film review scores on Rotten Tomatoes (the 'Tomato Meter') and such outcomes as Dollars taken at Box Office, movie Length, and indeed the year of release. Users can select a variety of options in the side panel which will call variables from a dataset stored in the application, which will then be displayed on the graph in the main panel.

The two examples above are just a flavour of how we can use Shiny to present and explore data in a number of interesting and engaging ways and build applications which can be used by a much wider range of audiences than might otherwise be exposed to data and data analysis in the sciences.

2. Introducing Shiny

First of all, we need to install and load the Shiny package:

```
install.packages("shiny")
library("shiny")
```

Allow the installation to proceed with compilation packages. After Shiny has been attached to the R session, open a new R script and write/copy in the following code:

```
library(shiny)

ui <- fluidPage()
### This is the user interface
### Here we build items for users to interact with and position our outputs.

server <- function(input, output) {}

### This is the server space
### Here we build our outputs as conditional reactive elements to selections
in the user interface

shinyApp(ui = ui, server = server)

### This line of code 'stitches' the UI and Server together into an app
```

As outlined in the script itself, the base of all Shiny Apps is split into three elements: the user interface (UI), a server, and the ShinyApp stitching the two together. The following section outlines the process in more detail, but a summary of what each space is for and how they work with each other follows here.

Everything that we want to user to see and interact with in our application goes into the UI. This of course includes the outputs as well as the selection boxes, information, and any other text or imagery we might want to include. So, while the server function handles exactly what the App does in response to user inputs, we need to specify the type and arrangements of those outputs in the UI.

So, for instance, imagine we want users to be able to select variables from a drop-down menu in order to create and investigate different graphs. In the UI, we would need to position a drop-down menu but also tell R that we want a graphical output to come after the menu and give it a name which is then called in the server space.

In the server space, we then call the output and assign it a render function, followed by a set of arguments (in our case, we would render a plot using renderPlot and then specify a type of graph depending on user-selected variables). This tells R exactly what to generate in the space we created in the UI and how it ought to ‘react’ to user inputs in the UI.

The ‘reactivity’ element of Shiny App building is the most important thing to remember when constructing, testing, and publishing your web applications. Whatever you allow users to specify in the UI, there needs to be a reactive element in the server telling the application what to do with those selections.

9/10 Shiny App problems and errors will be down to reactive elements not quite matching up between the UI and the server. Always follow the logic of **Input -> Response -> Output**.

3. Constructing an example Shiny App

We will begin now to construct an example application. We will build a simple app which will plot graphs based on three variables in a database.

Firstly, run this code to generate our sample database:

```
## create variables along 1-100 vectors person_id <- c(1:100)
chocolate_buttons <- c(1:100)

## generate and add some random noise for the dependent variables
happiness <- jitter(c(1:100), factor=100)
sugar_rush <- jitter(c(1:100), factor=150) health_perception <-
jitter(c(1:100), factor=200)

## create data frame from vectors
chocolate <- as.data.frame(cbind(person_id, chocolate_buttons,
happiness, sugar_rush, health_perception))

## restrict values to between 0 and 100 chocolate[chocolate>100]<-100
chocolate[chocolate<0]<-0
```

The data frame, chocolate, will later be turned into a CSV file to use with the app.

For now, having constructed the data frame, return to the app script from above and we will use the **Input -> Response-> Output** logic to plan our app.

Let's imagine we are designing a web-app so that users can see the results from our fake experiment. Let's say we want them to be able to freely and easily investigate the relationship between our test subjects eating a number of chocolate buttons (captured by thechocolate_buttons vector) and: reporting feeling happy (happiness on a scale of 0 to 100), feeling like they are having a sugar rush (scale of 0 to 100), and feeling like they are doing something healthy (scale of 0 to 100).

The best way to do this would be through a nice and simple plot, with number of chocolate buttons eaten on the x axis, and users able to select y-axis variables from the three options above.

Using the **Input -> Response -> Output** logic framework and the base script, we can quickly note down the outline of our Shiny app:

> Input <- a drop-down menu where users are selecting variables.
>
> Response <- **y-axis** variables pulled from data frame, changing based on user selection.
>
> Output <- a graph which will set selected **y-axis** variable against chocolate_buttons on the **x-axis**.

We can now move to constructing these three components in our Shiny app, using our base script.

i) The User Interface (UI)

The UI represents what users will see when the app loads in front of them – the face of the application. It is essentially the 'front of house' service, while the server operates as the 'back office' of the operation. In the **Input -> Response -> Output** logic, this section covers the 'Input' phase.

The first thing to do is define what kind of 'page' we want the UI to be – a static page where components stay locked in place, a fluid page where elements can stretch or relocate based on browser size (definitely recommended for most circumstances) or a page with some structured arrangements (for example side bars or panels) - in fact we can actually deploy sidebars and panels within other page structures, so making use of fluid pages is more so recommended. We will make use of the fluidPage set up for our app, but see here for alternatives.

The most important part of the UI is the selection module in which the user defines their choice.

For our app, we want users to be able to select columns from our data frame which will then be placed onto the y-axis of a plot. So, the first item in our UI (as specified in the first section of R code "ui <- fluidPage()") needs to be a drop-down selection menu which users pick their variable from. We can generate this by using the following code:

```
ui <- fluidPage(
    selectInput(inputId = "variable",
        label = "Select Variable",
        choices = c("Happiness", "Sugar Rush", "Health Perception"))
)
```

The 'selectInput' function calls a drop-down menu, which we then fill with a vector of 'choices', after having specified an 'inputId' (a token which we then 'bridge' into the server function) and a 'label' (a string of text which appears automatically above the menu).

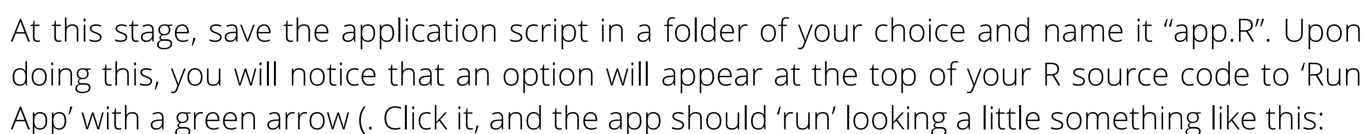
There are many different UI selection components which Shiny offers. To see them all (and get information on how to code them), take a look at this handy 'cheat sheet' from R Studio (inputs are down the right-hand side).

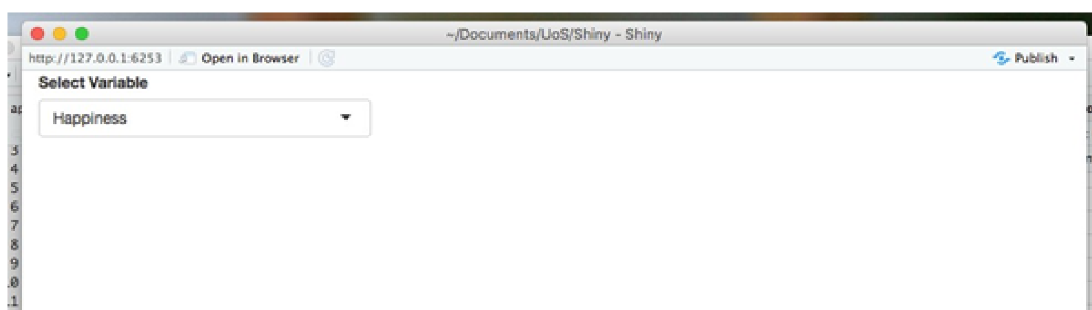
Since the output (but not the response) needs to be seen by the user as part of the interface, we also need to define some space in the UI for the output to appear. Further, we need to build a 'bridge' between the UI and the server which informs the latter as to exactly what the output is.

Since we want a graphical output, we call a 'plotOutput' and give it an appropriate token – "choco_graph":

```
ui <- fluidPage(
    selectInput(inputId = "variable",
        label = "Select Variable",
        choices = c("Happiness", "Sugar Rush", "Health Perception")),
    plotOutput("choco_graph")
)
```

At this stage, save the application script in a folder of your choice and name it "app.R". Upon doing this, you will notice that an option will appear at the top of your R source code to 'Run App' with a green arrow (. Click it, and the app should 'run' looking a little something like this:





We can see that a fully interactive selection box has indeed appeared, but of course without any reactive elements or outputs defined, the selections don't generate anything quite yet!

We should also note that for simple apps such as this, it really isn't that necessary to define a reactive element within the server. It is possible to link the user selection directly into the graph by placing "input$variable" after the "y = " code in the graph function. However, getting used to using the reactive element approach will pay dividends when you move on to developing more complex Shiny apps – such as when there are multiple paths or selections which users can make which could do things such as change databases or subset data frames.

That's that for the UI. We have completed specifying our input and defined some responsive elements (the "variable" and "choco_graph" tokens) to be fed back into our specified (and partially defined) output space. We now move on to server function, where we will tell R exactly what to do with those responsive elements and exactly what kind of output to generate.

ii) The Server

The server is the space in which we develop our responsive arguments and define their outputs. Following the **Input -> Response -> Output** logic, this section focuses on both the 'Response' and 'Output' phase.

We will first focus on the responsive element. There are a few ways to manage responsivity in Shiny Apps, but the cleanest and (hopefully) less error-prone method is to start off by defining a 'reactive' value2. Reactive values quite simply tell the server to produce a certain response based on user selections within an interface element.

In our case, we will be telling the server to create a function called 'yvarInput' which will then be called by the plot to determine which variable will be plotted on the y axis from our "chocolate" data frame. To do this, we write in the following code:

```
server <- function(input, output) { yvarInput <- reactive({
    switch(input$variable,
        "Happiness" = chocolate$happiness, "Sugar Rush" = chocolate$sugar_rush,
        "Health Perception" = chocolate$health_perception)
        })
    }
```

This 'reactive' function will change the value of yvarInput based on the user selection in the interface. So, for example, if the user selects "Happiness" from our selectInput box in the UI, the value of yvarInput will become "chocolate$happiness". Simple, right?

The second phase of the server build is covering the Output from our **Input -> Response -> Output** framework.

Thinking back to the logic of reactivity, we have defined a space for a plot to appear in our UI using the code "plotOutput("choco_graph")". Now, we need to tell Shiny what exactly the plot looks like that we want to appear in that space.

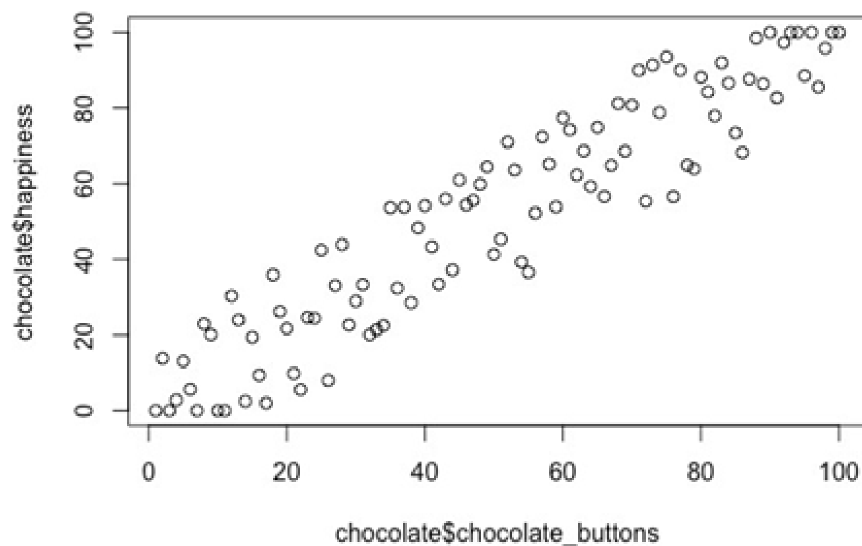
To keep things simple, we will use R's base graphics system to make a nice easy scatter plot for our users to interact with. One of the great things about Shiny though is that, with the right code, it can work with just about any other R package which generates outputs that Shiny is capable of reacting with and displaying (of which there are a lot, including text, pictures, maps, and so on – refer to the earlier links to see just how many). For instance, we could use ggplot in this example to make some nice, professional looking graphs which are more customisable than those in base R.

In base R, to call a scatter plot all we have to do is write the following code:

```
plot(x, y)
```

Let's call variables from the 'chocolate' data frame and write them into a plot with some labels, to get a sense of what exactly it is we would like to end up displaying in the UI of our Shiny app:

```
plot(chocolate$chocolate_buttons, chocolate$happiness)
```



We can see a nice linear, positive relationship between eating chocolate buttons and increasing happiness. To share these stunning findings, and other chocolate button related plots, with the rest of the world through our Shiny app, we need to tell Shiny to generate this plot but with an added reactive element so that users can change what the plot outputs on the y-axis (using the selection box defined in the UI).

To create a plot in the server, we need to inform the server to 'render' a plot in the space allocated for said plot in the UI. To do this we use the 'renderPlot' function. We need to 'render' outputs every time we call one into the UI - once again, the R studio cheat sheet (linked above) has lots of information on different types of outputs which can be generated, and the specific render calls required to load them up into the interface. We also need to 'bridge' the plot to the corresponding space in the UI using the 'inputId' from the selection box. Finally, we want the y variable to be interactive (responding to user selections in the UI) and so instead of assigning specific variables to the y axis in the plot command, we use our 'yvarInput' function defined above:

```
output$choco_graph <- renderPlot({
    plot(chocolate$chocolate_buttons, yvarInput())
    })
```

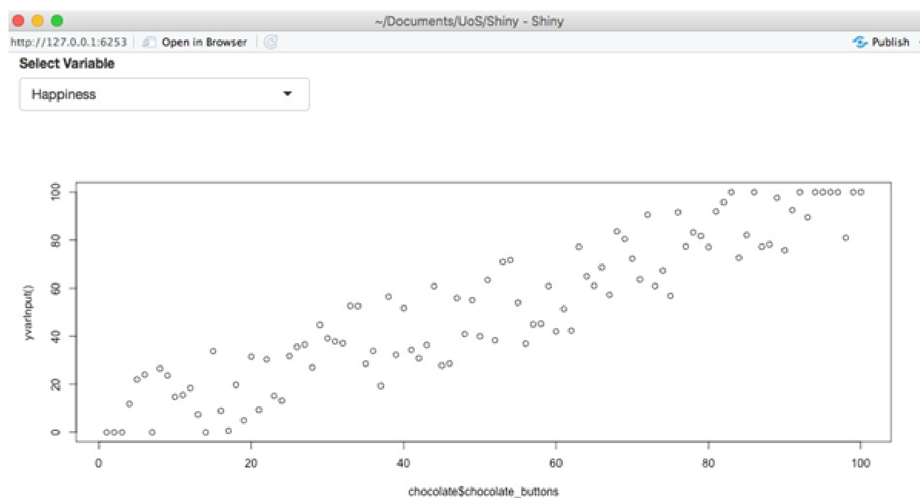
The "output$choco_graph" code 'bridges' to the UI using the "choco_graph" string which we wrote into the UI – "plotOutput("choco_graph)" – and so tells Shiny exactly where this plot should be assigned to in the interface.

Within the plot function, we set the x variable to be "chocolate$chocolate_buttons", and then use our reactive element function – yvarInput() – to assign y variables to the plot in accordance with user selections in the UI.

The final piece of code which stitches everything together into an application is already at the foot of our script:

```
shinyApp(ui = ui, server = server)
```

Save the script again and then run it. Upon doing that, you should be presented with a Shiny application looking a little something like this:



The graph will most likely not look exactly the same, because of the random nature of the variance introduced into the vectors by the 'jitter' function when we were creating the 'chocolate' data frame.

You should notice that changing the variable from the drop-down menu instantly spurs the reactivity chain into loading a new graph with the selected variable plotted on the y axis.

Now, before we finish we ought to tidy up the axis labels and add a title, so that users can fully understand what is going on as they interact with the dataset.

To do this, we simply add reactive label and title arguments into our plot – in exactly the same fashion as we would do if we were constructing a regular (static, non-Shiny) plot using base R.

The only difference is that we can use reactive elements to customise the labelling as the users interact with the app. So instead of a standard script such as:

The only difference is that we can use reactive elementsto customise the labelling as the usersinteract with the app. So instead of a standard script such as:

```
output$choco_graph <- renderPlot({
    plot(chocolate$chocolate_buttons, yvarInput(),
            xlab="Number of Chocolate Buttons Eaten",
            ylab="Happiness",
        main="Relationship between EatingChocolate Buttons and Happiness")
    })
```

We replace the ‘ylab’ and second half of the ‘main’ title with reactive elements, bridged to the UI. Putting the variable name into the ‘ylab’ argument is quite simple – as with calling it to the reactive element above in the server, we just write “input$variable” after “ylab=”.

Replacing the second portion of the main title text requires us to add an additional function – ‘paste’. Pasting allows users to combine arguments into a single text output. In our case, we want to combine “Relationship between Eating Chocolate Buttons and” followed by the name of the user-selected variable.

Again, we ‘bridge’ across to the ‘inputId’ code in the UI to make this happen – “paste(“Relationship betweenEating Chocolate Buttonsand”, input$variable)”. So, to complete the app, replacethe plot scriptfrom above with the following:

```
output$choco_graph <- renderPlot({
    plot(chocolate$chocolate_buttons, yvarInput(),
        xlab="Number of Chocolate Buttons Eaten",
        ylab=input$variable,
    main=paste("Relationship between Eating ChocolateButtons and",
    input$variable))
    })
```

Run the app to see the results!

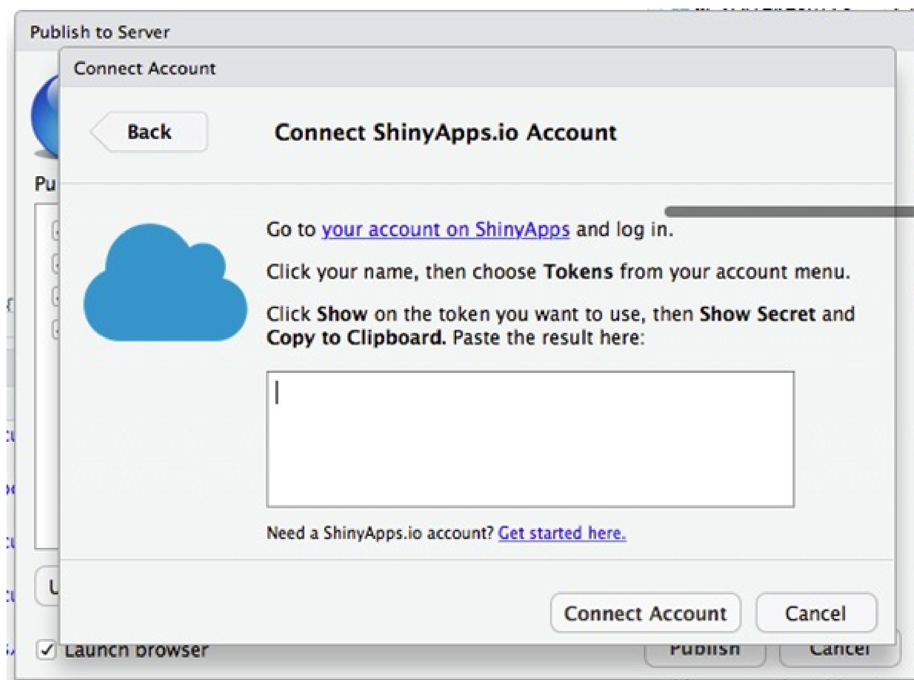
4. Publishing Shiny Apps on Shinyapp.io

The final section of this guide quickly runs through how to publish apps to ShinyApps.io – R Studio’s server for Shiny App development. This is the quickest and easiest way to get your application online and get sharing it with the world.

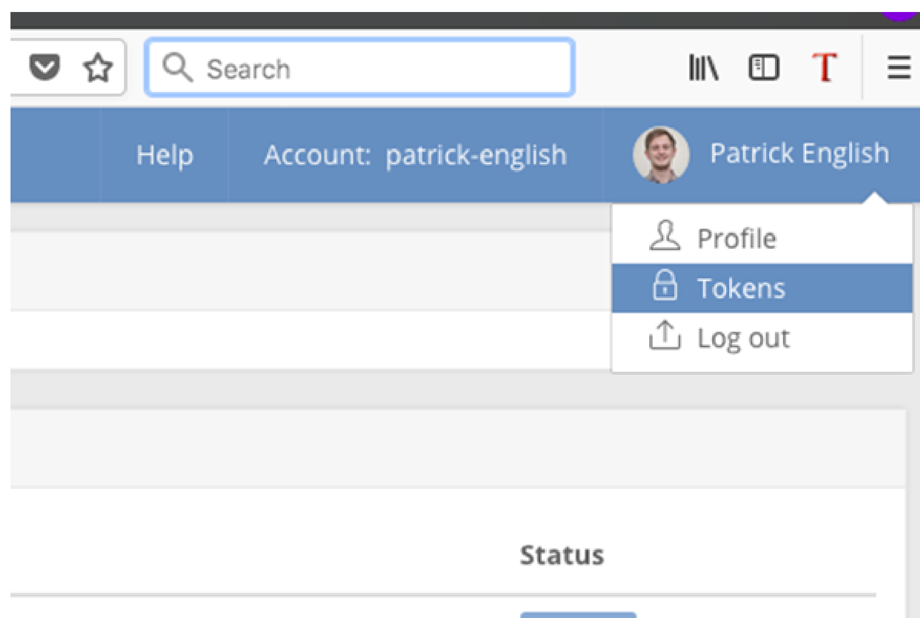
Firstly, it will be necessary to set up an account on Shinyapps.io. Head on over to http://www.shinyapps.io to create an account, via the 'Sign Up' button. Shinyapps.io offers a variety of server-space packages, with prices increasing with required combinations of number of applications to be hosted and 'active user time' on said apps. For now, create a free account (accounts can always be upgraded and downgraded at the user's request).

Once you have created an account, proceed then to install the 'rsconnect' package from the CRAN repository. This package pretty much automates the entire process of publishing Shiny applications to the Shinyapps.io web server. Load the package into the session using the 'library(rsconnect)' command.

You will notice that next to the 'Run App' option at the top of the app script there is a blue icon. This icon is a shortcut to app publication. With rsconnect installed and loaded, upon clicking this icon you will be invited to connect your RStudio session to Shinyapps.io account. Follow this link to be presented with the following screen asking for a 'token':



The 'token' to connect your RStudio session to Shinyapps.io can be found online through your Shinyapps.io dashboard. The second option in the drop-down menu under your account name contains your 'token':



Ask Shinyapps.io to 'Add Token' for you and then copy the code directly from the dialogue box on the website into the dialogue box back in your RStudio session. Once this step is completed, your RStudio session will by synced up to Shinyapps.io and you will be ready to publish your application simply hitting the newly-created 'publish' option next to the blue icon! Fill out an application Title (the name under which the application will be published) and rsconnect will handle the building and uploading of your application. Once this is done, the application will appear in your Shiny dashboard and it is all ready to share around.

And that's it! How to construct and publish Shiny applications.