Shiny Apps allow developers and researchers to easily build interactive web applications within the environment of the statistical software R. Using these apps, R users can interactively communicate their work to a broader audience.

After reading this blog post and engaging with the applied examples, readers should:

* be able to retrace the logic behind the structure of Shiny Apps,
* be able to build their own Shiny App, and
* be able to deploy their Shiny App and run them in the world wide web.

**Shiny Apps and Data Presentation**

Data are usually collected in a raw format and, no matter how well processed and analyzed, results should be presented in a plain and simple way in order to make them accessible for everyone. Unfortunately, the potential of effective data presentation is all too often not fully exploited. Yet, presentation is crucial as it illustrates the actual outcome of research. If not effectively visualized, we waste a great opportunity to build credibility, attract and sustain the interest of readers, and make large amounts of information easily accessible. Here, we introduce Shiny Apps as a powerful and highly flexible tool for interactive data presentation in the world wide web.

Shiny is an R package that offers cost- and programming-free tools for building web applications using R. Since Shiny Apps come with an extensive back-end setup, users do not need extensive web development skills to build and host standalone apps on a homepage. However, for those keen on bringing their apps to perfection, Shiny Apps allows for CSS, HTML and JavaScript extensions.Shiny Apps can be used either for data presentation, as a communication tool for results, or even as an interactive analytical tool.

In what follows, we introduce the Shiny environment and guide readers through the development of Shiny Apps. Using the famous Kaggle titanic data set, we draw the distinction between the front-end ui.R and back-end server.R, which are required to build Shiny apps. Following this, we introduce important concepts and features to build an interactive app, including control widgets, reactivity, and rendering.

**The Structure of Shiny Apps**

Shiny applications have two components. The front-end builds the webpage that is actually shown to the user. As already mentioned, the HTML page is written by Shiny itself and includes layout, appearance, and design features. In Shiny terminology, this is called the ui, which stands for user interface. The ui file contains R functions that are then translated into an HTML file.  
The other component is the back-end, which includes the code for producing the app’s contents (e.g. functions or data import, management, and analysis). Here, we create the objects that are later shown on the front-end. In Shiny terminology this is called the server.

**Ways of Setting Up a Shiny App**

Shiny Apps can be set up in two different ways.

**1. Single-file App:**  
In a single-file app, ui and server are stored in one script. In this case, we create a file named app.R that contains both the server and UI components. This technique is used when developing very simple Shiny Apps and lacks some advantages of the alternative two-files App method.

**2. Two-files App:**  
Here, ui and server are stored in two separate scripts, which implies a clear separation between front-end and back-end. This method is preferable when developing more advanced Shiny Apps. It is **important that the files are named ui.R and server.R and always stored in a separate folder**. In this tutorial, we are going to develop Shiny Apps using the two-files method.

**Developing Shiny Apps**

**Building a Shiny App from Scratch**

To set up, make sure that all required packages are installed and subsequently load the shiny,tidyverse, and plotly packages along with R’s example data set titanic.

Code: R packages used in this tutorial

## Packages

pkgs <- c("shiny",

"tidyverse",

"titanic",

"plotly")

## Install uninstalled packages

lapply(pkgs[!(pkgs %in% installed.packages())], install.packages)

## Load all packages to library

lapply(pkgs, library, character.only = TRUE)

Next, create a new folder with two R scripts, ui.R and server.R:

Code: Load packages and create folder with two R scripts

ui <- fluidPage()

server <- function(input, output) {

}

Subsequently, launch the Shiny App by pressing the “Run App” button in the top right corner of the Source pane. Also, have a look at the the code chunk below, which provides an example of a Shiny App.

Code: Example

runExample("01\_hello")

# To show other Apps, please type runExample(NA) and choose another example

**Building the UI**

When building a Shiny App, one should have in mind what the app should look like. Hence, we build the UI first. In simple Shiny Apps, the whole UI fits in the fluidPage: every new object is passed comma-separated, and text can be passed to the UI simply by entering strings. In order to format text, Shiny uses HTML wrappers, i.e., functions that take text as arguments (plus further style options).

* h1(): top-level header
* h2(): secondary header
* strong(): make text bold
* em(): make text italicized
* br(): add line break
* titlePanel(): adds an official header

Until now, we only have a plain white page. We need a proper layout to make our app look nicer. sidebarlayout is the simplest layout format. It splits your page in two parts: a sidebarPaneland a mainPanel. Inside sidebarLayout you may use sidebarPanel to specify the appearance of the side panel of your page. The mainPanel contains the results of the  
server.

Code: Building the plain UI

ui <- fluidPage(titlePanel("Title of my Shiny App"),

sidebarLayout(

sidebarPanel("My input goes here"),

mainPanel("The results go here")

))

In order to interact with Shiny Apps, we need control widgets, such as buttons, select boxes, or sliders. These allow us to navigate the app. The type of the widget you choose depends on the design of your app. You can either specify inputs, enter text, or select specific dates to create results. All input functions have two arguments: inputId and label.

inputId tells Shiny to refer to a particular input when retrieving values for the back-end. **It is important that the ID names you use are unique**. If you have two IDs with the same name, there will not be a warning or error message. The label option specifies the text displaying the label of the control widget. The control labels go in the sidebarPanel. Here, we specify the possible values, range, and the appearance of the control widget.

Having added the input and the control widgets, we finally need to specify the output elements of our app. These outputs might be plots, tables, text, images, or maps. These will be built in the server file of our two-files app. In the UI, we only build the placeholder, which means that every output function has an outputId argument to identify the output created in server.R. Here, we use plotlyOutput to refer to a plot that will be generated later in server.R. Output elements should always be added to the mainPanel() function. Other types of output elements are:

* dataTableOutput(): data table
* imageOutput(): image
* plotOutput(): plot
* tableOutput(): table
* verbatimTextOutput(): text
* textOutput(): text
* uiOutput(): raw HTML
* htmlOutput(): raw HTML

The code chunk below demonstrates how widget and output elements can be combined.

Code: Finishing the UI

# The most simple structure of the UI:

ui <- fluidPage(

titlePanel("Visualizing the Titanic data set"),

sidebarLayout(

sidebarPanel(

checkboxInput("checkbox",

"Activate all filters",

FALSE),

radioButtons(

"buttons",

"Did the passenger survive?",

choices = c("did not survive" = 0, "did survive" = 1),

selected = 0

),

selectInput("selector",

"Select the class of the passenger",

choices = c(1, 2, 3)),

sliderInput(

"slider",

"Pick a range of fare (in $)",

min = 0,

max = 550,

value = c(10, 50),

pre = "$"

),

selectInput(

"selectVars",

"Select the variable to be displayed",

choices = c(

"Age" = "Age",

"# of Siblings/Spouses on board" = "SibSp",

"# of Parents/Children on board" = "Parch"

)

)

),

mainPanel(plotlyOutput("greatPlot"),

textOutput("goodText"))

)

)

**Implementing the Server Logic**

Having finished the front-end, we now turn to building the server or the back-end. The server logic in Shiny Apps uses an input argument, which contains the values of the input given by the users, as well as an output argument, which contains the plots and tables created as a function of the input arguments. Output objects can be created without any input specification, but always need to be connected with the UI by an outputId. To do so, save the output object into the output list. We then build the object using one of the available render functions. For every object type, there is a unique render function, e.g. renderPlot(). The table below provides a brief overview of which render functions work with which output formats.

| **render()** | **Output()** |
| --- | --- |
| renderDataTable() | dataTableOutput() |
| renderImage() | imageOutput() |
| renderPlot() | plotOutput() |
| renderPrint() | verbatimTextOutput() |
| renderTable() | tableOutput() |
| renderText() | textOutput() |
| renderUI() | uiOutput() and htmlOutput() |

We have to make sure that we accurately assign our IDs to their corresponding render and output functions. Since we named our plotOutput in the UI greatPlot, we have to assign it to this ID. As we use the plotOutput function in the UI, we need the renderPlot function to match it. Furthermore, our plots need a connection to the input and the control widgets. The logic behind the input list is the same as for the output list: As we used the sliderInput function in the UI and named it slider, we now access the respective object. Given the structure of the input, we specify it to fit the output object.

Code: Fitting IDs and output/render functions

output$greatPlot <- renderPlot({

plot(rnorm(input$slider[1]))

})

**Rendering Plots**

There are several ways of increasing the interactivity of plots. For example, one can simply add code to the renderplot() function. Another very simple way of increasing interactivity is filtering. However, when filtering, the whole code will get re-executed every time the user re-specifies anything, which can become very memory-intensive and repetitive when building several plots! We therefore use reactive variables to extract the filtering from the render functions. Reactive variables can now be used for different objects, which change simultaneously with different input. Additionally, we use plotly plots to make it even more interactive. You simply need to replace the plotOutput() with plotlyOutput() in the UI.

Code: Make plots reactive and interactive

# Back-end

# The server function takes input and output as arguments

# input: Input provided by the user to specify a certain appearance, data

# or result

# output: The output created by the functions specified in the back end

server <- function(input, output) {

# Create a reactive variable to use it for different objects interactively

filtered <- reactive({

titanic\_train %>%

filter(

Fare >= input$slider[1],

Fare <= input$slider[2],

Pclass == input$selector,

Survived == input$buttons

)

})

# Plot the variables of the titanic\_train data set

# Using if-conditions to specify the variables being plotted

output$greatPlot <- renderPlotly({

if (input$checkbox == TRUE) {

x <- filtered()[, input$selectVars]

plotHist <- ggplot(filtered(), aes(x)) +

geom\_histogram()

} else {

x <- titanic\_train[, input$selectVars]

plotHist <- ggplot(titanic\_train, aes(x)) +

geom\_histogram()

}

ggplotly(plotHist)

})

output$goodText <- renderText({

if (input$checkbox == TRUE) {

x <- nrow(filtered()[filtered()$Survived == 1, ])

text <-

paste0("Using your filter ", x, " passengers have survived")

return(text)

} else {

x <- nrow(titanic\_train[titanic\_train$Survived == 1, ])

text <- paste0("In total ", x, " passengers have survived")

return(text)

}

})

}

To finally visiualize our plot locally, we simply run shinyApp(ui = ui, server = server). This allows us to see the final Shiny App, including some additional components, which increases the functionality of the app even further.

**Deployment**

**Deployment using** [**shinyapps.io**](http://shinyapps.io)

Until now, we only ran our apps locally on our machine. In order to present an app to a broader audience, we need to deploy it in the world wide web. We have to take care that our app is protected by a firewall and that we have a stable URL. For simple apps, deploy the app using a free account on [shinyapps.io](https://www.shinyapps.io/). If done so, go back to RStudio, press the deploy button in the top right corner of the Source pane, re-enter your credentials, select the correct files, and let the magic happen!

**Deployment using Shiny Server**

For certain apps you might not want to use [shinyapps.io](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/shiny-apps/wwww.shinyapps.io), for instance if:

* the size of the app would require a (very) costly plan, or
* you want full control over the app and host it yourself, or
* you love playing around with Unix code.

A free alternative is Shiny Server, which allows you to host your own app in a controlled environment (e.g. inside your organization). The professional version of Shiny Server (Shiny Server Pro) allows you to deploy password-protected apps and use an administrative dashboard that provides you with usage statistics. However, you need an own server to host the app (e.g. Digital Ocean or Amazon Web Services), which may still be costly.

Another alternative is hosting the app on university servers. For instance, members of the University of Mannheim can use the web services provided by the universities of Baden-Württemberg, the *bwCloud*, for free.

When using Shiny Server, the required steps are a bit more complicated:

1. [Register](https://www.bw-cloud.org/de/erste_schritte) for the bwCloud
2. Log into the bwCloud Dashboard
3. Create an SSH-key pair to connect to your server
4. Install [PuTTY](https://www.putty.org/) (Windows) or start a remote connection in the Shell (MAC) as well as [Filezilla](https://filezilla-project.org/)
5. Set up the SSH-client to access remote connections
6. Build a virtual machine (the server) using your bwCloud dashboard and connect to your virtual machine using SSH with PuTTY

After setting up the server, we need to enter our Unix system and install R, all relevant packages, and Shiny Server using Unix code

1. Install R on your machine

sudo apt-get install r-base

1. Install dependencies to install R packages

sudo apt-get -y install libcurl4-gnutls-dev

sudo apt-get -y install libxml2-dev libssl-dev

1. Install R packages within R, including shiny (easy!)

install.packages("shiny")

1. Install Shiny Server (check for latest version!)

wget <https://download3.rstudio.org/ubuntu-12.04/x86_64/shiny-server-1.5.6.875-amd64.deb>

sudo gdebi shiny-server-1.5.6.875-amd64.deb

1. Open port 3838 and check whether your firewall is set up

sudo ufw allow 3838

1. Test whether Shiny Server runs correctly. Open this in your browser: <http://134.155.108.111:3838/> (replace with your IP-address)
2. Use Filezilla to access your server and upload your app files to the /srv/shiny-server/ folder. It should run when typing in the correct URL associated with your app: <http://134.155.108.111:3838/foldername/projectname/>
3. The above will likely not work immediately. In this case, you need to go back and troubleshoot…

**Conclusion**

Data presentation is crucial for accessible scientific research. Even with accurate and comprehensive data that supports relevant findings, failure to present the data in an easily accessible way can result in wasted opportunities. Shiny Apps are a powerful and free-of-cost tool that allows scholars to provide readers with easy-to-use, interactive, and comprehensive insights into their research. For instance, scholars can use Shiny Apps to program an interactive online appendix and thereby offer readers full control to compare findings under different specifications of measurement and modelling.

However, the advantages of creating a functional web-application exclusively with R has some limitations. Considering the design and the placement of inputs (such as sliders) or outputs (such as graphics and tables), Shiny is limited. When combining Shiny with HTML, CSS, and JavaScript, however, it offers a good way to program professional web-apps. Shiny is particularly great for fast prototyping and fairly easy to use with little experience in programming. It comes with many different charting libraries and captures feedback and comments in a structured manner. It therefore offers an exciting toolbox that can be a valuable addition to scientific research.