Interactive data visualization using Shiny

Partly based on the RStudio Shiny tutorial

Andrzej Oleś

10 June 2015

Contents

1 Getting to know Shiny **1**

1.1 Structure of a Shiny App . . . 1

1.2 Running an App . . . 3

1.3 Your turn . . . 3

2 Building a user interface **5**

2.1 Layout . . . 5

2.2 HTML Content . . . 6

2.3 Other tags . . . 8

3 Control widgets **9**

3.1 Adding widgets . . . 10

4 Use R scripts and data **12**

5 Interactive MA plot explorer **12**

5.1 Interface layout . . . 13

1 Getting to know Shiny

Shiny is an R package that makes it easy to build interactive web applications (apps) straight from R. The Shiny package comes with eleven built-in examples that each demonstrate how Shiny works. Each example is a self-contained Shiny app.

The **Hello Shiny** example plots a histogram of R's `faithful` dataset with a configurable number of bins. Users can change the number of bins with a slider bar, and the app will immediately respond to their input. You'll use **Hello Shiny** to explore the structure of a Shiny app and to create your first app.

To run **Hello Shiny**, type:

```r
library(shiny)
runExample("01_hello")
```

1.1 Structure of a Shiny App

Shiny apps have two components:

- a user-interface script
- a server script

The user-interface (ui) script controls the layout and appearance of your app. It is defined in a source script named `ui.R`. Below is the `ui.R` script for the **Hello Shiny** example.

```
library(shiny)

# Define UI for application that draws a histogram
shinyUI(fluidPage(

  # Application title
  titlePanel("Hello Shiny!"),

  # Sidebar with a slider input for the number of bins
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),

    # Show a plot of the generated distribution
    mainPanel(
      plotOutput("distPlot")
    )
  )
))
```

The `server.R` script contains the instructions that your computer needs to build your app. Here is the `server.R` script for the **Hello Shiny** example.

```
library(shiny)

# Define server logic required to draw a histogram
shinyServer(function(input, output) {

  # Expression that generates a histogram. The expression is
  # wrapped in a call to renderPlot to indicate that:
  #
  #  1) It is "reactive" and therefore should re-execute automatically
  #     when inputs change
  #  2) Its output type is a plot

  output$distPlot <- renderPlot({
    x    <- faithful[, 2]  # Old Faithful Geyser data
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })
})
```

At one level, the **Hello Shiny** `server.R` script is very simple. The script does some calculations and then plots a histogram with the requested number of bins.

However, you'll also notice that most of the script is wrapped in a call to `renderPlot`. The comment above the function explains a bit about this, but if you find it confusing, don't worry. We'll cover this concept in much more detail soon.

Play with the **Hello Shiny** app and review the source code. Try to develop a feel for how the app works.

Your R session will be busy while the **Hello Shiny** app is active, so you will not be able to run any R commands. R is monitoring the app and executing the app's reactions. To get your R session back, hit escape or click the stop sign icon (found in the upper right corner of the RStudio console panel).

1.2 Running an App

Every Shiny app has the same structure: two R scripts saved together in a directory. At a minimum, a Shiny app has `ui.R` and `server.R` files.

You can create a Shiny app by making a new directory and saving a `ui.R` and `server.R` file inside it. Each app will need its own unique directory.

You can run a Shiny app by giving the name of its directory to the function `runApp`. For example, if your Shiny app is in a directory called `my_app`, run it with the following code:

```
library(shiny)
runApp("my_app")
```

Note: The first argument of `runApp` is the filepath from your working directory to the app's directory. The code above assumes that the app directory is in your working directory. In this case, the filepath is just the name of the directory.

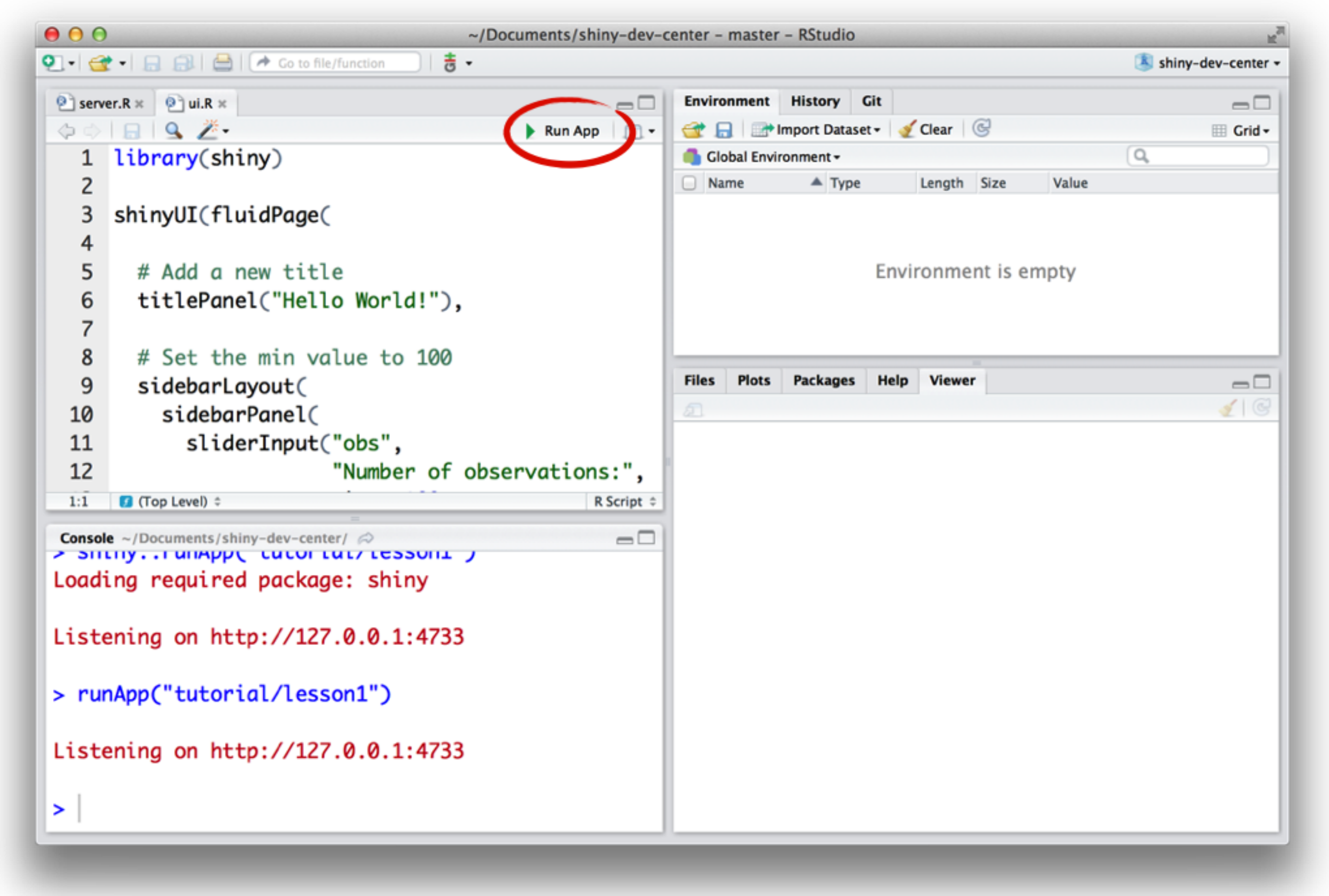
In case you are wondering, the **Hello Shiny** app's files are saved in a special system directory designed to work with the `runExample` call. Alternatively, you can also launch the app using

```
runApp(system.file(package="shiny", "examples", "01_hello"))
```

1.3 Your turn

Create your first Shiny app named `MyApp`. For this, start RStudio and from the **File** menu choose *New Project... > New Directory > Shiny Web Application*. As the directory name enter the desired name `MyApp`. The project will start with `ui.R` and `server.R` containing the familar code from the **Hello Shiny** app.

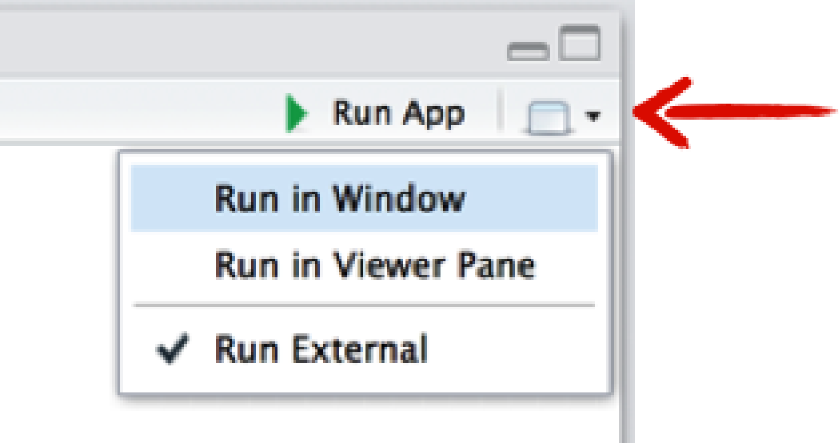
To launch your app execute `runApp("MyApp")` in your R console. Alternatively, you can click the Run App button (at the top of the editor), or or use the keyboard shortcut: `Command+Shift+Enter` on MAC, or `Control+Shift+Enter` on Windows.



```r
library(shiny)

shinyUI(fluidPage(

  # Add a new title
  titlePanel("Hello World!"),

  # Set the min value to 100
  sidebarLayout(
    sidebarPanel(
      sliderInput("obs",
                  "Number of observations:",
```

```
> shiny::runApp('tutorial/lesson1')
Loading required package: shiny

Listening on http://127.0.0.1:4733

> runApp("tutorial/lesson1")

Listening on http://127.0.0.1:4733

>
```

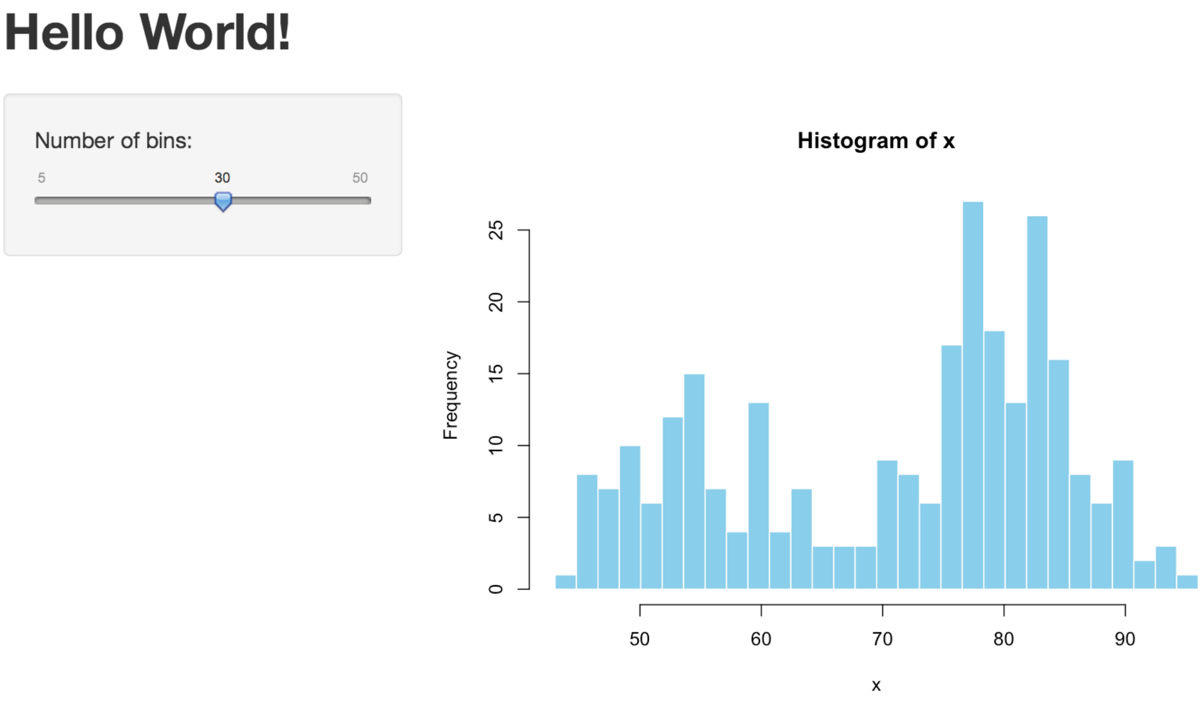
RStudio will launch the app in a new window by default, but you can also choose to have the app launch in a dedicated viewer pane, or in your external web browser. Make your selection by clicking the icon next to Run App.



Now make some changes to your app:

1. Change the title from “Hello Shiny!” to “Hello World!”.
2. Set the minimum value of the slider bar to 5.
3. Change the histogram color from `"darkgray"` to `"skyblue"`.

When you are ready, launch your app again. Your new app should match the image below.



Hello World!

Number of bins: 30

2 Building a user interface

Now that you understand the structure of a Shiny app, it's time to build your first app from scratch. We will start by showing how to build a user interface for your app. You will learn how to lay out the user interface and add HTML elements to your Shiny app.

We'll use the `MyApp` you made before. To get started, open its server.R and ui.R files, and edit the scripts to match the ones below:

ui.R

```
shinyUI(fluidPage(
))
```

server.R

```
shinyServer(function(input, output) {
})
```

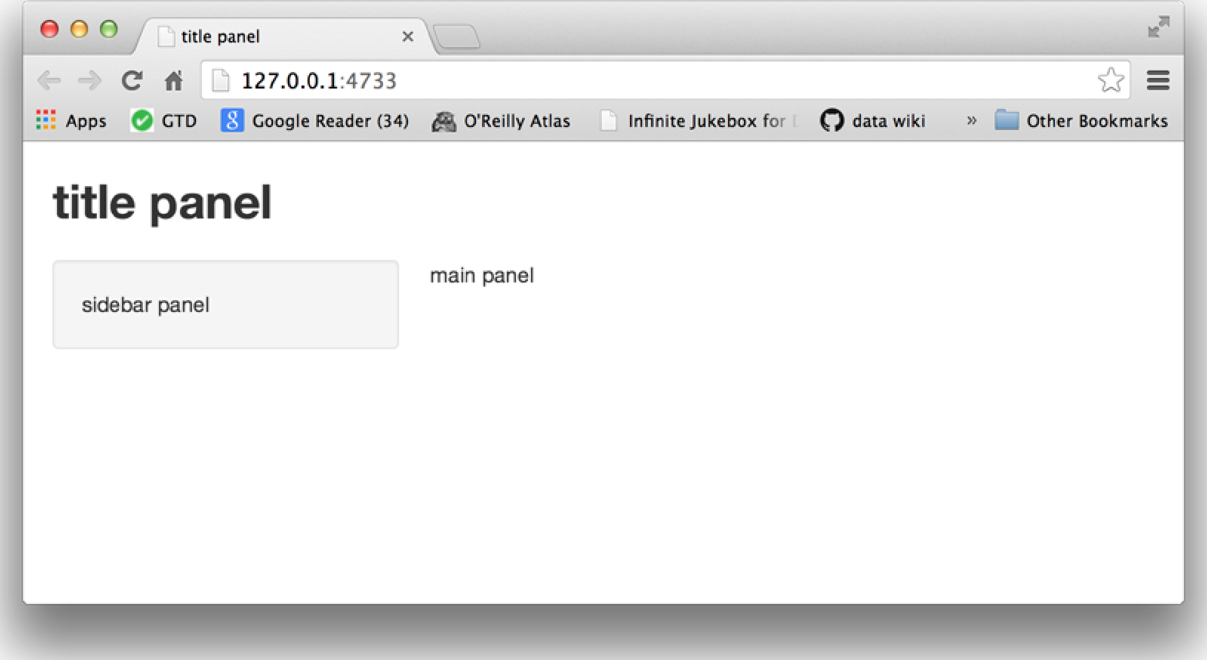
This code is the bare minimum needed to create a Shiny app. The result is an empty app with a blank user interface.

2.1 Layout

Shiny `ui.R` scripts use the function `fluidPage` to create a display that automatically adjusts to the dimensions of your user's browser window. You lay out your app by placing elements in the `fluidPage` function.

For example, the `ui.R` script below creates a user-interface that has a title panel and then a sidebar layout, which includes a sidebar panel and a main panel. Note that these elements are placed within the fluidPage function.

```
# ui.R

shinyUI(fluidPage(
  titlePanel("title panel"),

  sidebarLayout(
    sidebarPanel( "sidebar panel"),
    mainPanel("main panel")
  )
))
```



title panel

sidebar panel

main panel

titlePanel and sidebarLayout are the two most popular elements to add to fluidPage. They create a basic Shiny app with a sidebar. sidebarLayout always takes two arguments:

- sidebarPanel function output
- mainPanel function output

These functions place content in either the sidebar or the main panels.

titlePanel and sidebarLayout create a basic layout for your Shiny app, but you can also create more advanced layouts. For example, navbarPage creates a multi-page user-interface that includes a navigation bar. You can also divide the page into equally spaced regions using flowLayout and inputPanel, or you can use fluidRow and column to build your layout up from a grid system. For an overview, see the *Shiny cheat sheet*.

2.2 HTML Content

You can add content to your Shiny app by placing it inside a *Panel function. For example, the apps above display a character string in each of their panels. The words "sidebar panel" appear in the sidebar panel, because we added the string to the sidebarPanel function, i.e. sidebarPanel("sidebar panel"). The same is true for the text in the title panel and the main panel.

To add more advanced content, use one of Shiny's HTML tag functions. These functions parallel common HTML5 tags. As an example, let's try out creating headers.

2.2.1 Headers

To create a header element:

- select a header function (e.g., `h1` or `h3`)
- give it the text you want to see in the header

For example, you can create a first level header that says "My title" with `h1("My title")`. If you run the command at the command line, you'll notice that it produces HTML code.

```
h1("My title")
## <h1>My title</h1>
```

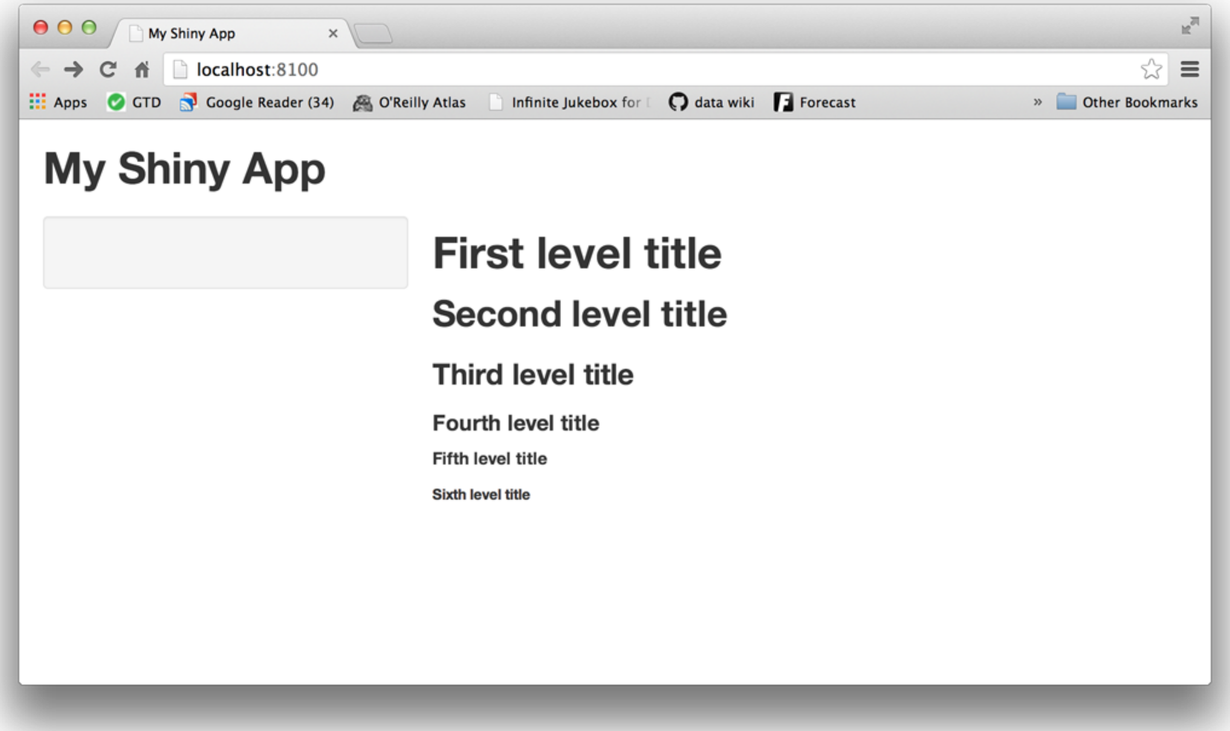
To place the element in your app, pass `h1("My title")` as an argument to `titlePanel`, `sidebarPanel`, or `mainPanel`.

The text will appear in the corresponding panel of your web page. You can place multiple elements in the same panel if you separate them with a comma.

Give this a try. The new script below uses all six levels of headers. Update your `ui.R` to match the following script and then relaunch your app.

```
# ui.R

shinyUI(fluidPage(
  titlePanel("My Shiny App"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel(
      h1("First level title"),
      h2("Second level title"),
      h3("Third level title"),
      h4("Fourth level title"),
      h5("Fifth level title"),
      h6("Sixth level title")
    )
  )
))
```

Now your app should look like this:



My Shiny App

First level title

Second level title

Third level title

Fourth level title

Fifth level title

Sixth level title

2.3 Other tags

We have demonstrated how to use the header tag functions, but there are many more tag functions for you to use. `shiny::tags` contains a list of 110 functions, where each function builds a specific HTML tag. If you are familiar with HTML, you will recognize these tags by their names.

```
names(tags)
##   [1] "a"          "abbr"       "address"    "area"       "article"
##   [6] "aside"      "audio"      "b"          "base"       "bdi"
##  [11] "bdo"        "blockquote" "body"       "br"         "button"
##  [16] "canvas"     "caption"    "cite"       "code"       "col"
##  [21] "colgroup"   "command"    "data"       "datalist"   "dd"
##  [26] "del"        "details"    "dfn"        "div"        "dl"
##  [31] "dt"         "em"         "embed"      "eventsource" "fieldset"
##  [36] "figcaption" "figure"     "footer"     "form"       "h1"
##  [41] "h2"         "h3"         "h4"         "h5"         "h6"
##  [46] "head"       "header"     "hgroup"     "hr"         "html"
##  [51] "i"          "iframe"     "img"        "input"      "ins"
##  [56] "kbd"        "keygen"     "label"      "legend"     "li"
##  [61] "link"       "mark"       "map"        "menu"       "meta"
##  [66] "meter"      "nav"        "noscript"   "object"     "ol"
##  [71] "optgroup"   "option"     "output"     "p"          "param"
##  [76] "pre"        "progress"   "q"          "ruby"       "rp"
##  [81] "rt"         "s"          "samp"       "script"     "section"
##  [86] "select"     "small"      "source"     "span"       "strong"
##  [91] "style"      "sub"        "summary"    "sup"        "table"
##  [96] "tbody"      "td"         "textarea"   "tfoot"      "th"
## [101] "thead"       "time"        "title"       "tr"          "track"
## [106] "u"           "ul"          "var"         "video"       "wbr"
```

To create a tag, run an element of tags as a function. For instance, to create a div tag, run:

```
tags$div()
##<div></div>
```

You can call some of the most popular tags with helper functions that makes accessing them easier. For example, the code function is a wrapper for tags$code and creates text formatted as computer code. Other available helper functions are: a, br, code, div, em, h1, h2, h3, h4, h5, h6, hr, img, p, pre, span, and strong. The names of other tags functions conflict with the names of native R functions, so you will need to call them with the tags$ syntax.

3 Control widgets

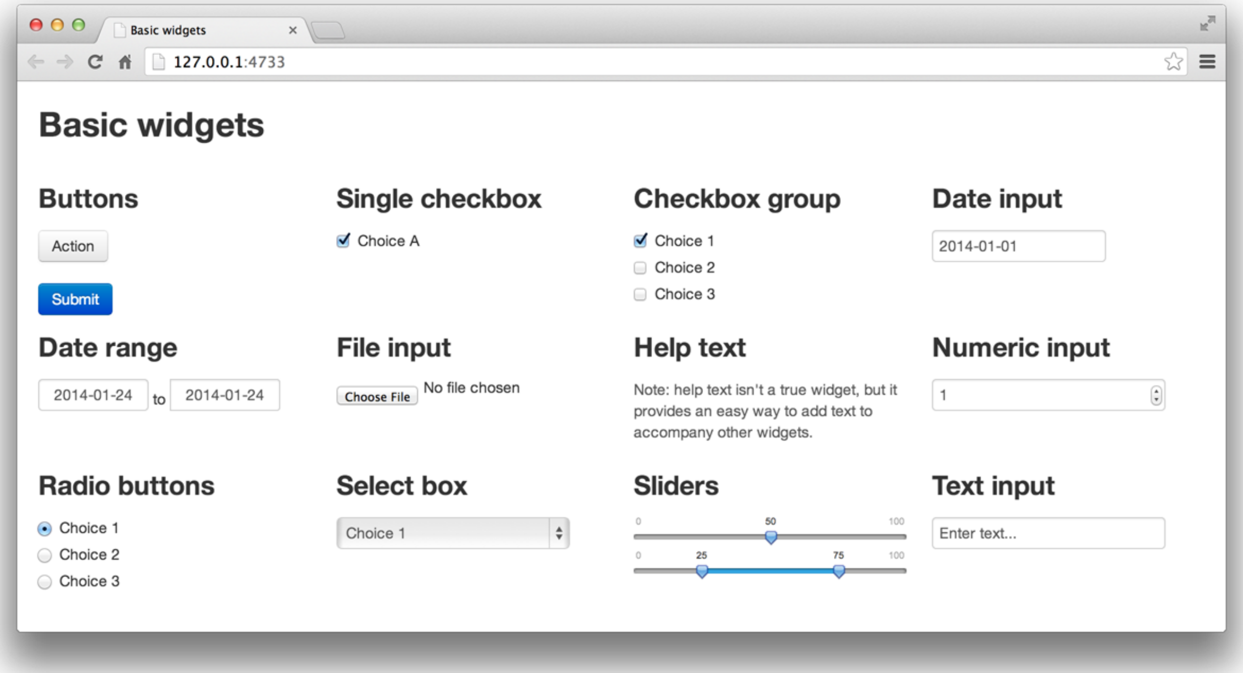
Now that you know how to create the layout of the user interface, it's time to add control widgets to your Shiny app. A widget is a web element that users can interact with. Widgets provide a way to send messages to the Shiny app.

Shiny widgets collect a value from the user. When the user changes the widget, the value will change as well.

Shiny comes with a family of pre-built widgets, each created with a transparently named R function. For example, Shiny provides a function namessliderInput that creates a slider bar.

The standard Shiny widgets are:

function	widget
actionButton	Action Button
checkboxGroupInput	A group of check boxes
checkboxInput	A single check box
dateInput	A calendar to aid date selection
dateRangeInput	A pair of calendars for selecting a date range
fileInput	A file upload control wizard
helpText	Help text that can be added to an input form
numericInput	A field to enter numbers
radioButtons	A set of radio buttons
selectInput	A box with choices to select from
sliderInput	A slider bar
submitButton	A submit button
textInput	A field to enter text



Basic widgets

Buttons
Action
Submit

Single checkbox
☑ Choice A

Checkbox group
☑ Choice 1
□ Choice 2
□ Choice 3

Date input
2014-01-01

Date range
2014-01-24 to 2014-01-24

File input
Choose File No file chosen

Help text
Note: help text isn't a true widget, but it provides an easy way to add text to accompany other widgets.

Numeric input
1

Radio buttons
● Choice 1
○ Choice 2
○ Choice 3

Select box
Choice 1

Sliders

Text input
Enter text...

Some of these widgets are built using the Twitter Bootstrap project, a popular open source framework for building user interfaces.

3.1 Adding widgets

You can add widgets to your web page in the same way that you added other types of HTML content by placing a widget function in `sidebarPanel` or `mainPanel` in your `ui.R` file.

Each widget function requires several arguments. The first two arguments for each widget are

- A Name for the widget. The user will not see this name, but you can use it to access the widget's value. The name should be a character string.
- A label. This label will appear with the widget in your app. It should be a character string, but it can be an empty string `""`.

In this example, the name is "action" and the label is "Action": `actionButton("action", label = "Action")`

The remaining arguments vary from widget to widget, depending on what the widget needs to do its job, like initial values, ranges, and increments. You can find the exact arguments needed by a widget on the widget function's help page, e.g., `?selectInput`.

The code below makes the app pictured above. Change the `ui.R` script of `MyApp` to match it, and then relaunch the app.

Play with each widget to get a feel for what it does. Experiment with changing the values of the widget functions and observe the effects.

```
# ui.R

shinyUI(fluidPage(
  titlePanel("Basic widgets"),

  fluidRow(
    column(3,
      h3("Buttons"),
      actionButton("action", label = "Action"),
      br(),
      br(),
      submitButton("Submit")),

    column(3,
      h3("Single checkbox"),
      checkboxInput("checkbox", label = "Choice A", value = TRUE)),

    column(3,
      checkboxGroupInput("checkGroup",
        label = h3("Checkbox group"),
        choices = list("Choice 1" = 1,
           "Choice 2" = 2, "Choice 3" = 3),
        selected = 1)),

    column(3,
      dateInput("date",
        label = h3("Date input"),
        value = "2014-01-01"))
  ),

  fluidRow(

    column(3,
      dateRangeInput("dates", label = h3("Date range"))),

    column(3,
      fileInput("file", label = h3("File input"))),

    column(3,
      h3("Help text"),
      helpText("Note: help text isn't a true widget,",
        "but it provides an easy way to add text to",
        "accompany other widgets.")),

    column(3,
      numericInput("num",
        label = h3("Numeric input"),
        value = 1))
  ),

  fluidRow(

    column(3,
      radioButtons("radio", label = h3("Radio buttons"),
        choices = list("Choice 1" = 1, "Choice 2" = 2,
                       "Choice 3" = 3),selected = 1)),

    column(3,
      selectInput("select", label = h3("Select box"),
        choices = list("Choice 1" = 1, "Choice 2" = 2,
                       "Choice 3" = 3), selected = 1)),

    column(3,
      sliderInput("slider1", label = h3("Sliders"),
        min = 0, max = 100, value = 50),
      sliderInput("slider2", "",
        min = 0, max = 100, value = c(25, 75))
      ),

    column(3,
      textInput("text", label = h3("Text input"),
        value = "Enter text..."))
  )

))
```

4 Use R scripts and data

We will now show you how to load other scripts and packages to use in your Shiny apps. Our ultimate goal will be building an interactive MA plot explorer.

5 Interactive MA plot explorer

We will now build an app that creates an interactive MA plot which will allow you to select a gene by clicking on it and displaying a plot counts of the selected gene.

We will use the *pasilla* data package contaning per-exon and per-gene read counts of RNA-seq samples of Pasilla knock-down by Brooks et al., Genome Research 2011.

Let's start by loading the data and analyzing differential expression between the treated and untreated samples.

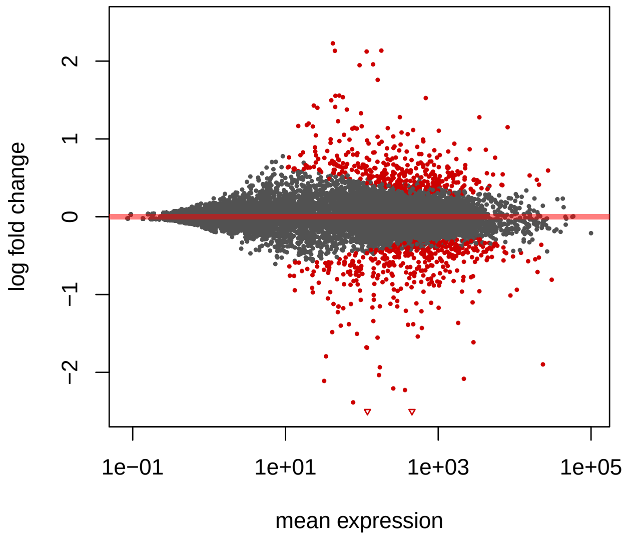
```
library(pasilla)
library(DESeq2)
library(Biobase)
```

```
data(pasillaGenes)

# DE analysis of the pasilla dataset from Bioconductor
dds <- DESeqDataSetFromMatrix(counts(pasillaGenes),
                              pData(pasillaGenes)[,2:3],
                              ~ condition)

# compare treated vs untreated
dds$condition <- relevel(dds$condition, "untreated")
dds <- dds[rowSums(counts(dds)) > 0, ]

# run DESeq
dds <- DESeq(dds)
## estimating size factors
## estimating dispersions
## gene-wise dispersion estimates
## mean-dispersion relationship
## final dispersion estimates
## fitting model and testing
```

```
res <- results(dds)
```

We can now plot the results.

```
ymax <- 2.5
plotMA( res, ylim=c( -ymax, ymax ) )
```



For the plot to appear in our Shiny app, we need to . . .

5.1 Interface layout

We would like to display two plots side by side: the MA plot on the left and the counts for the selected gene on the right. We will divide the page into two equally sized parts using `flowLayout`, and set the width and height of the plots to 400 pixels.

```
# ui.R

shinyUI(fluidPage(
  titlePanel("MA-plot + counts plot with shiny"),

  # flow layout fills out left to right then down
  flowLayout(

    # the MA-plot
    plotOutput("plotma", click="plotma_click", width=400, height=400),

    # the counts plot for a single gene
    plotOutput("plotcounts", width=400, height=400),
    # needed for proper page layout
    cellArgs = list(style="width: 400px;")
  )

))
```

We have used the argument cellArgs to ensure the correct size of the layout elements containing the plots.

Note the additional click argument to the plotOutput function of the MA plot. If set to a value like plotma_click, the plot will send coordinates to the server whenever it is clicked. This value will be accessible via input$plotma_click and will contain a named list with x and y elements indicating the mouse position.

We will use the input$plotma_click coordinates to resolve the nearest gene in the MA plot.