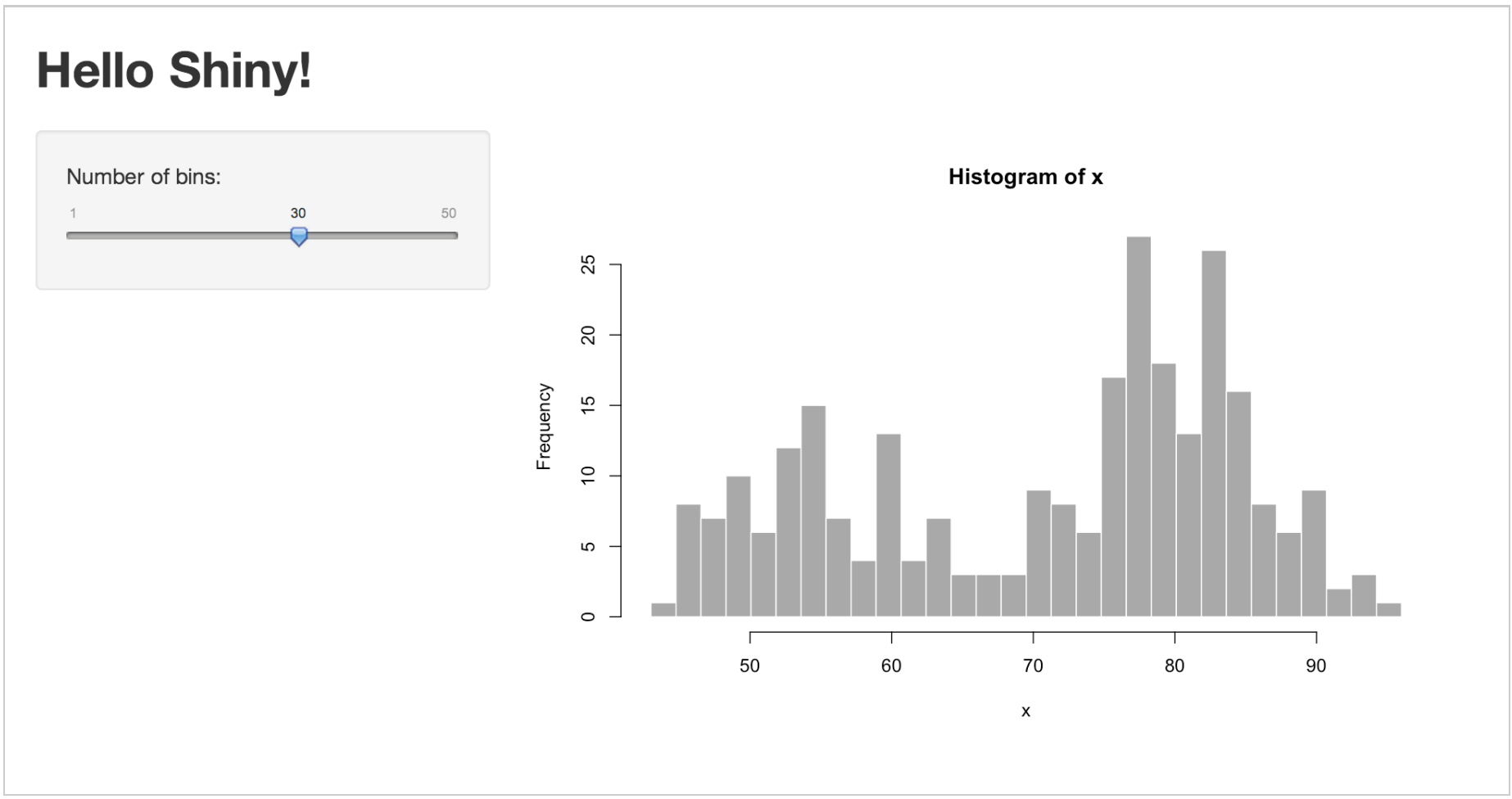
The basic parts of a Shiny app

ADDED: 06 JAN 2014

The Shiny package comes with ten built-in examples that demonstrate how Shiny works. This article reviews the first three examples, which demonstrate the basic structure of a Shiny app.

Example 1: Hello Shiny



The Hello Shiny example is a simple application that plots R's built-in `faithful` dataset with a configurable number of bins. To run the example, type:

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

Shiny applications have two components: a user-interface definition and a server script. The source code for both of these components is listed below.

In subsequent sections of the article we'll break down Shiny code in detail and explain the use of "reactive" expressions for generating output. For now, though, just try playing with the sample application and reviewing the source code to get an initial feel for things. Be sure to read the comments carefully.

The user interface is defined in a source file named ui.R:

ui.R

```
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")
    )
  )
))
```

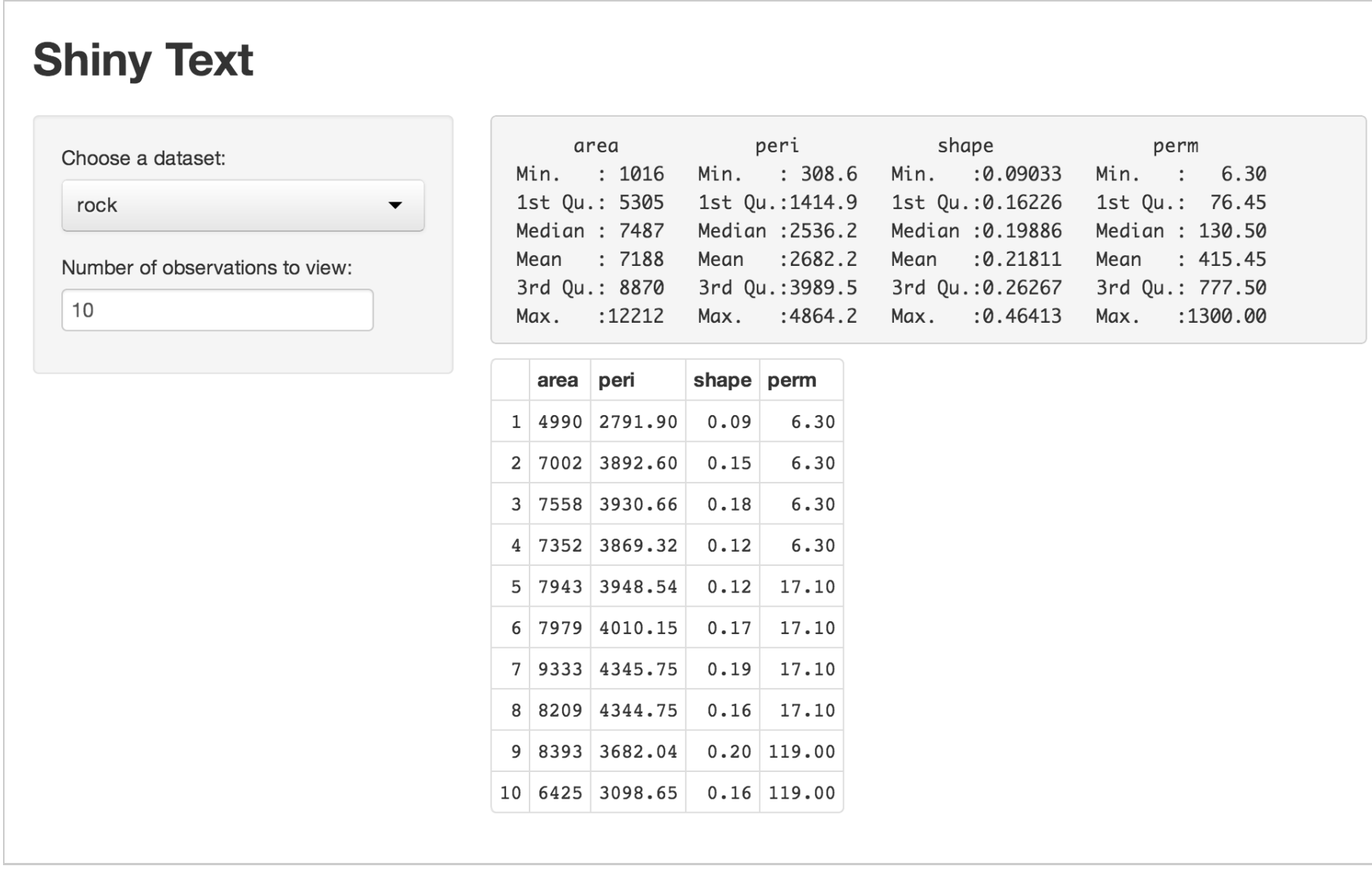
The server-side of the application is shown below. At one level, it's very simple–a random distribution is plotted as a histogram with the requested number of bins. However, you'll also notice that the code that generates the plot is wrapped in a call to `renderPlot`. The comment above the function explains a bit about this, but if you find it confusing, don't worry–we'll cover this concept in much more detail soon.

server.R

```
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 be automatically
  #     re-executed 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')
  })

})
```

The next example will show the use of more input controls, as well as the use of reactive functions to generate textual output.

Example 2: Shiny Text



The Shiny Text application demonstrates printing R objects directly, as well as displaying data frames using HTML tables. To run the example, type:

```
> library(shiny)
> runExample("02_text")
```

The first example had a single numeric input specified using a slider and a single plot output. This example has a bit more going on: two inputs and two types of textual output.

If you try changing the number of observations to another value, you'll see a demonstration of one of the most important attributes of Shiny applications: inputs and outputs are connected together "live" and changes are propagated immediately (like a spreadsheet). In this case, rather than the entire page being reloaded, just the table view is updated when the number of observations change.

Here is the user interface definition for the application. Notice in particular that the `sidebarPanel` and `mainPanel` functions are now called with two arguments (corresponding to the two inputs and two outputs displayed):

ui.R

```
library(shiny)

# Define UI for dataset viewer application
shinyUI(fluidPage(

  # Application title
  titlePanel("Shiny Text"),

  # Sidebar with controls to select a dataset and specify the
  # number of observations to view
  sidebarLayout(
    sidebarPanel(
      selectInput("dataset", "Choose a dataset:",
                  choices = c("rock", "pressure", "cars")),

      numericInput("obs", "Number of observations to view:", 10)
    ),

    # Show a summary of the dataset and an HTML table with the
   # requested number of observations
    mainPanel(
      verbatimTextOutput("summary"),

      tableOutput("view")
    )
  )
))
```

The server side of the application has also gotten a bit more complicated. Now we create:

- A reactive expression to return the dataset corresponding to the user choice
- Two other rendering expressions (`renderPrint` and `renderTable`) that return the `output$summary` and `output$view` values

These expressions work similarly to the `renderPlot` expression used in the first example: by declaring a rendering expression you tell Shiny that it should only be executed when its dependencies change. In this case that's either one of the user input values (`input$dataset` or `input$obs`).

server.R

```
library(shiny)
library(datasets)

# Define server logic required to summarize and view the selected
# dataset
shinyServer(function(input, output) {

  # Return the requested dataset
  datasetInput <- reactive({
    switch(input$dataset,
           "rock" = rock,
           "pressure" = pressure,
           "cars" = cars)
  })

  # Generate a summary of the dataset
  output$summary <- renderPrint({
    dataset <- datasetInput()
    summary(dataset)
  })

  # Show the first "n" observations
  output$view <- renderTable({
    head(datasetInput(), n = input$obs)
  })
})
```

We've introduced more use of reactive expressions but haven't really explained how they work yet. The next example will start with this one as a baseline and expand significantly on how reactive expressions work in Shiny.

Example 3: Reactivity

Reactivity

Caption:

Data Summary

Choose a dataset:

rock

Number of observations to view:

10

Data Summary

```
      area            peri           shape            perm
 Min.   : 1016   Min.   : 308.6   Min.   :0.09033   Min.   :   6.30
 1st Qu.: 5305   1st Qu.:1414.9   1st Qu.:0.16226   1st Qu.:  76.45
 Median : 7487   Median :2536.2   Median :0.19886   Median : 130.50
 Mean   : 7188   Mean   :2682.2   Mean   :0.21811   Mean   : 415.45
 3rd Qu.: 8870   3rd Qu.:3989.5   3rd Qu.:0.26267   3rd Qu.: 777.50
 Max.   :12212   Max.   :4864.2   Max.   :0.46413   Max.   :1300.00
```

	area	peri	shape	perm
1	4990	2791.90	0.09	6.30
2	7002	3892.60	0.15	6.30
3	7558	3930.66	0.18	6.30
4	7352	3869.32	0.12	6.30
5	7943	3948.54	0.12	17.10
6	7979	4010.15	0.17	17.10
7	9333	4345.75	0.19	17.10
8	8209	4344.75	0.16	17.10
9	8393	3682.04	0.20	119.00
10	6425	3098.65	0.16	119.00

The Reactivity application is very similar to Hello Text, but goes into much more detail about reactive programming concepts. To run the example, type:

```
> library(shiny)
> runExample("03_reactivity")
```

The previous examples have given you a good idea of what the code for Shiny applications looks like. We've explained a bit about reactivity, but mostly glossed over the details. In this section, we'll explore these concepts more deeply. If you want to dive in and learn about the details, see the Understanding Reactivity section, starting with Reactivity Overview.

What is Reactivity?

The Shiny web framework is fundamentally about making it easy to wire up *input values* from a web page, making them easily available to you in R, and have the results of your R code be written as *output values* back out to the web page.

```
input values => R code => output values
```

Since Shiny web apps are interactive, the input values can change at any time, and the output values need to be updated immediately to reflect those changes.

Shiny comes with a **reactive programming** library that you will use to structure your application logic. By using this library, changing input values will naturally cause the right parts of your R code to be reexecuted, which will in turn cause any changed outputs to be updated.

Reactive Programming Basics

Reactive programming is a coding style that starts with **reactive values**–values that change over time, or in response to the user–and builds on top of them with **reactive expressions**–expressions that access reactive values and execute other reactive expressions.

What's interesting about reactive expressions is that whenever they execute, they automatically keep track of what reactive values they read and what reactive expressions they invoked. If those "dependencies" become out of date, then they know that their own return value has also become out of date. Because of this dependency tracking, changing a reactive value will automatically instruct all reactive expressions that directly or indirectly depended on that value to re-execute.

The most common way you'll encounter reactive values in Shiny is using the `input` object. The `input` object, which is passed to your `shinyServer` function, lets you access the web page's user input fields using a list-like syntax. Code-wise, it looks like you're grabbing a value from a list or data frame, but you're actually reading a reactive value. No need to write code to monitor when inputs change–just write reactive expression that read the inputs they need, and let Shiny take care of knowing when to call them.

It's simple to create reactive expression: just pass a normal expression into `reactive`. In this application, an example of that is the expression that returns an R data frame based on the selection the user made in the input form:

```
datasetInput <- reactive({
   switch(input$dataset,
          "rock" = rock,
          "pressure" = pressure,
          "cars" = cars)
})
```

To turn reactive values into outputs that can viewed on the web page, we assigned them to the `output` object (also passed to the `shinyServer` function). Here is an example of an assignment to an output that depends on both the `datasetInput` reactive expression we just defined, as well as `input$obs`:

```
output$view <- renderTable({
   head(datasetInput(), n = input$obs)
})
```

This expression will be re-executed (and its output re-rendered in the browser) whenever either the `datasetInput` or `input$obs` value changes.

Back to the Code

Now that we've taken a deeper look at some of the core concepts, let's revisit the source code and try to understand what's going on in more depth. The user interface definition has been updated to include a text-input field that defines a caption. Other than that it's very similar to the previous example:

ui.R

```
library(shiny)

# Define UI for dataset viewer application
shinyUI(fluidPage(

  # Application title
  titlePanel("Reactivity"),

  # Sidebar with controls to provide a caption, select a dataset,
  # and specify the number of observations to view. Note that
  # changes made to the caption in the textInput control are
  # updated in the output area immediately as you type
  sidebarLayout(
    sidebarPanel(
      textInput("caption", "Caption:", "Data Summary"),

      selectInput("dataset", "Choose a dataset:",
                  choices = c("rock", "pressure", "cars")),

      numericInput("obs", "Number of observations to view:", 10)
    ),

    # Show the caption, a summary of the dataset and an HTML
    # table with the requested number of observations
    mainPanel(
      h3(textOutput("caption", container = span)),

      verbatimTextOutput("summary"),

      tableOutput("view")
    )
  )
))
```

Server Script

The server script declares the `datasetInput` reactive expression as well as three reactive output values. There are detailed comments for each definition that describe how it works within the reactive system:

server.R

```
library(shiny)
library(datasets)

# Define server logic required to summarize and view the selected
# dataset
shinyServer(function(input, output) {
```

```
# By declaring datasetInput as a reactive expression we ensure
# that:
#
#  1) It is only called when the inputs it depends on changes
#  2) The computation and result are shared by all the callers
#   (it only executes a single time)
#
datasetInput <- reactive({
  switch(input$dataset,
         "rock" = rock,
         "pressure" = pressure,
         "cars" = cars)
})

# The output$caption is computed based on a reactive expression
# that returns input$caption. When the user changes the
# "caption" field:
#
#  1) This function is automatically called to recompute the
#     output
#  2) The new caption is pushed back to the browser for
#     re-display
#
# Note that because the data-oriented reactive expressions
  # below don't depend on input$caption, those expressions are
  # NOT called when input$caption changes.
  output$caption <- renderText({
    input$caption
  })

  # The output$summary depends on the datasetInput reactive
  # expression, so will be re-executed whenever datasetInput is
  # invalidated
  # (i.e. whenever the input$dataset changes)
  output$summary <- renderPrint({
    dataset <- datasetInput()
    summary(dataset)
  })

  # The output$view depends on both the databaseInput reactive
  # expression and input$obs, so will be re-executed whenever
  # input$dataset or input$obs is changed.
  output$view <- renderTable({
    head(datasetInput(), n = input$obs)
  })
})
```

We've reviewed a lot code and covered a lot of conceptual ground in the first three examples. The next article (/articles/build.html) focuses on the mechanics of building a Shiny application from the ground up.

We love it when R users help each other, but RStudio does not monitor or answer the comments in this thread. If you'd like to get specific help, we recommend the Shiny Discussion Forum (https://groups.google.com/forum/#!forum/shiny-discuss) for in depth discussion of Shiny related questions and How to get help (http://shiny.rstudio.com/articles/help.html) for a list of the best ways to get help with R code.