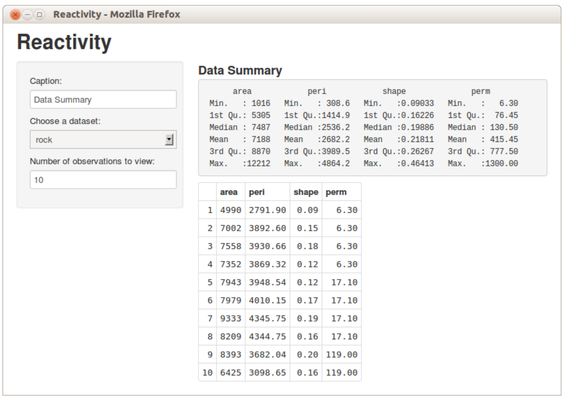

The Reactivity application is very similar to Hello Text, but goes into much more detail about reactive programming concepts. To run the example, type:

```
> 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 If those "dependencies" become out of date, then they know that their own return value has also become out of date.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(pageWithSidebar(

  # Application title
  headerPanel("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
  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")),

    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 expression 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 section focuses on the mechanics of building a Shiny application from the ground up and also covers tips on how to run and debug Shiny applications.