University of Glasgow

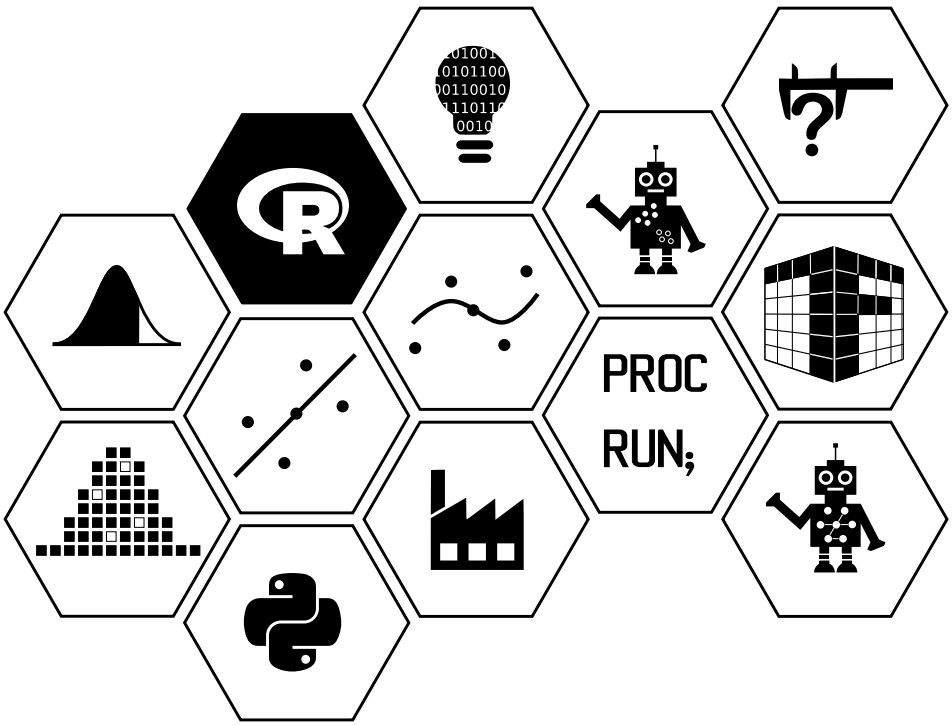
R Programming/ Statistical Computing

Craig Alexander

Academic Year 2023-24

Week 10:

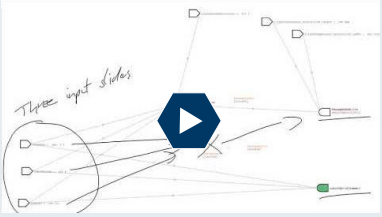
Building interactive webapps using Shiny



DATA ANALYTICS GLASGOW

Overview

RStudio Shiny provides an easy and intuitive way of developing webapps directly from inside R. Shiny will take care of all the technical details, so all we need to do is to specify the layout of the app and implement the data processing that generates the output. Shiny is especially well-suited to efficiently creating small apps.



Shiny Apps

https://youtu.be/aoNocuZYv7M

Duration: 16m10s

Shiny Demo Apps

The package shiny comes with a large number of examples. The 11 built-in examples can be listed using

```
library(shiny)
runExample()

## Valid examples are "01_hello", "02_text", "03_reactivity", "04_mpg", "05_sliders",
## "06_tabsets", "07_widgets", "08_html", "09_upload", "10_download", "11_timer"
```

The first example can then be run using

```
runExample("01_hello")
```

More than 100 examples are available on Github. These can be then run using, for example

```
# Requires package ggvis
runGitHub("shiny-examples", "rstudio", subdir = "051-movie-explorer")
```

Shiny automatically generates an HTML page ("user interface" or "ui" for short), which will open automatically when we run `runApp(...)`, `runExample(...)` or `runGitHub(...)` locally. Interacting with the controls will send messages to R, which in will turn trigger an update of the calculations and outputs. The entire process is automated, so we only need to describe how the user interface should look like and how to produce plots and other output.

Basic architecture

A shiny app is defined by two files which have to be in the same directory:

- The file `ui.R` specifies the layout ("user interface") of the app.
- The file `server.R` contains the "backend" performing the calculations and producing the plots and other outputs.

You can also put all the code into a single file `app.R` or provide a suitable R object or script to `runApp`. In this unit we will always use `ui.R` and `server.R` to make illustration clearer.

Example 1 (A simple Shiny app).

Our first Shiny app will generate a sample from the $N(\mu, \sigma^2)$ distribution, and plot the estimated probability density function. The user will be able to change sample size n, mean μ and standard deviation σ.

The focus in this example is on the big picture. We will take a more detailed look at individual functions later on.

The file `ui.R` specifies the user interface:

```
# This file controls how the shiny app looks like
```

```
fluidPage(

  # Title of your app (at top, spread across entire page)
  titlePanel("Our first Shiny app"),

  # Typically the controls are arranged in a sidebar on the left
  sidebarLayout(
      sidebarPanel(
          sliderInput("n",                        # Name of the variable
#                     ^^^ This name will be referred to in server.R
                      "Number of observations"  , # Label shown
                      min=1, max=1000, step=1,    # Range and (optional) step size
                      value = 100                 # Initial value
                      ),
          sliderInput("mu",                       # Name of the variable
#                     ^^^^ This name will be referred to in server.R
                      "Mean",                     # Label shown
                      min=-10, max=10, step=0.1,  # Range and (optional) step size
                      value = 0                   # Initial value
                      ),
          sliderInput("sigma",                    # Name of the variable
#                     ^^^^^^^ This name will be referred to in server.R
                      "Standard deviation",       # Label shown
                      min=0, max=5, step=0.1,     # Range and (optional) step size
                      value = 1                   # Initial value
                      )
      ),

  # The main panel is typically used for displaying R output
      mainPanel(h2("Estimated Density" ),
                plotOutput("densityPlot")         # We'll put a plot here
#                          ^^^^^^^^^^^^ This name will be referred to in server.R
                )
  )
)
```

The file `server.R` contains the code performing the calculations and producing the output:

```
library(ggplot2)

# This file contains the code for performing calculations
# and controls how the shiny app looks like

shinyServer(function(input, output) {

    output$densityPlot <-  renderPlot( {
#          ^^^^^^^^^^^ Must be the name used in ui.R
        n <- input$n                # Get n from slider
#                   ^ Must be the name used in ui.R
        mu <- input$mu              # Get mu from slider
#                    ^^ Must be the name used in ui.R
        sigma <- input$sigma        # Get sigma from slider
#                    ^^^^^ Must be the name used in ui.R
        x <- rnorm(n, mu, sigma)    # Generate sample
        ggplot(data.frame(x = x), aes(x, col="estimated")) +
            geom_density() +
            geom_rug() +
            xlim(-10,10) +
            ylim(0,0.75) +        # Plot the estimated density of the data ...
            stat_function(fun=function(x) dnorm(x, mu, sigma), aes(colour="exact"))
                                  # ... and add the true density
```

```
    })

})
```

The code for the app is available in the folder `Example1` in the GitHub repository here.

We can now run

```
# Change working directory to the directory Example1 containing server.R and ui.R
runApp()
```

which will show the app. You can also run the app in "showcase mode". This will also show the code and highlight the code being currently run in yellow.

```
runApp(display.mode="showcase")
```

Deploying Shiny Apps

If you run a Shiny app, it will by default only be accessible from your local computer. You can make your Shiny app available to others in various ways:

- Others can run the app locally on their computer, so you can send the code for your app to other R users (who must have the package `shiny` installed) or make the code for your app available online, for example on Gist or in a Github repository.
- You can upload your app to shinyapps.io: there are different pricing schemes, the most basic one being free. Apps can be uploaded to shinyapps.io using `deployApp()` or the "Publish" button in RStudio. Detailed guidance is available at https://shiny.rstudio.com/articles/shinyapps.html.
- You can install your own shiny server (either the open-source version or the commercial "pro" version). This requires access to a Linux server and basic server and web administration skills. See https://www.rstudio.com/products/shiny/download-server/ for details.

Shiny Gallery

https://shiny.rstudio.com/gallery/

The Shiny Gallery has a large number of showcase apps illustrating the wide range of apps that can be developed using Shiny.

RStudio Shiny Tutorial

https://shiny.rstudio.com/tutorial/

RStudio have produced a detailed Shiny tutorial, which provides additional details. The tutorial is available as video and in written form.

Shiny Cheat Sheet

https://github.com/rstudio/cheatsheets/blob/main/shiny.pdf

RStudio have put together a very handy and compact cheat sheet for Shiny.

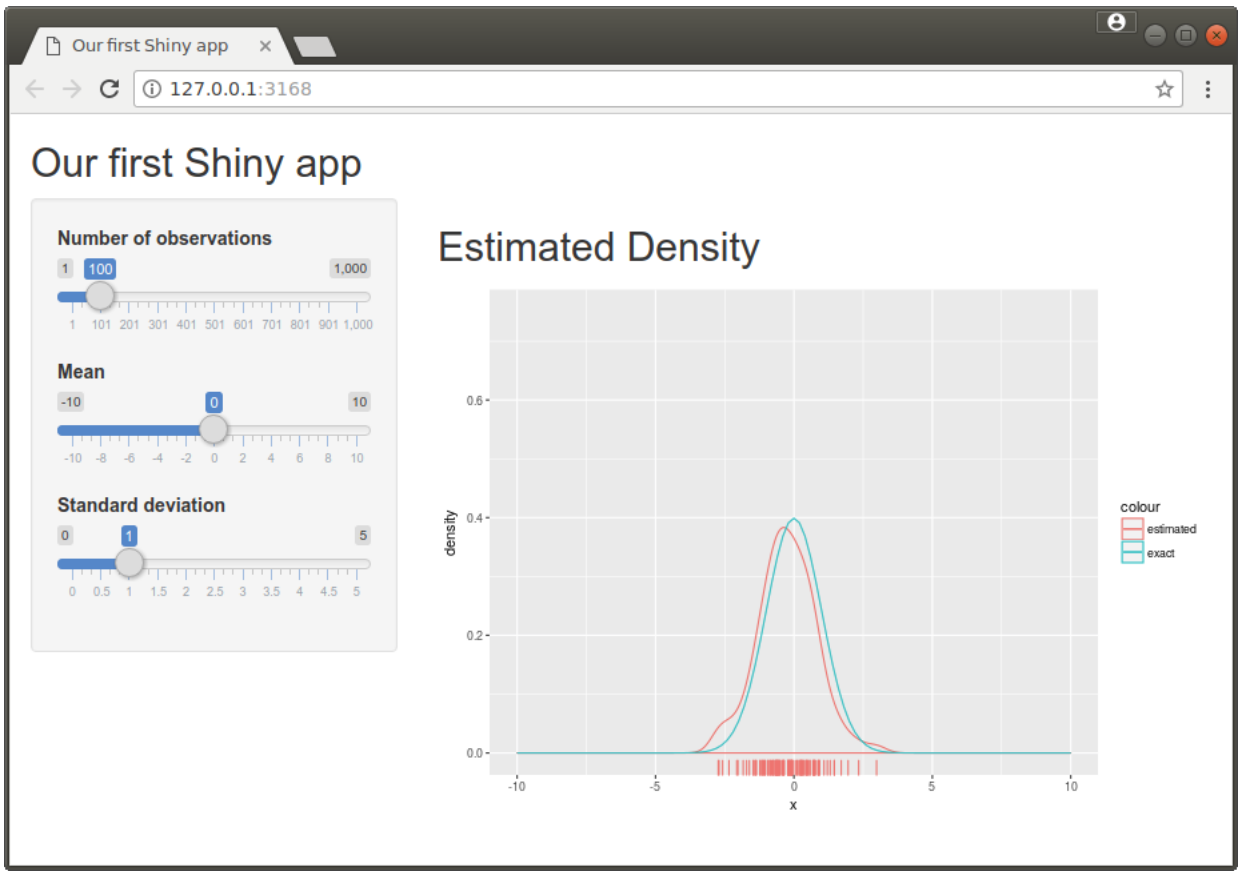


Figure 1: Screenshot of the app from example 1

Building a user interface in Shiny

In this section we will focus on how to construct a user interface in Shiny. We will only work with the file `ui.R`. Our file `server.R` will be an empty skeleton.

In the next section we will then look at how to implement functionality in `server.R`, but we will first look at constructing a user interface.

The file `ui.R` should contain a single call to the function creating the user interface. The function `fluidPage` creates a normal "fluid" web page containing the controls. There are also other functions for creating pages, notably `navbarPage`, which creates a page with a navigation bar at the top.

Twitter Bootstrap

Shiny uses the front-end framework Bootstrap to construct the HTML interface. Bootstrap was developed by Twitter and is by far the most widely used front-end framework for responsive web pages.

Bootstrap is based on a grid system, which splits the screen into up to twelve columns. Bootstrap's grid system is "responsive", i.e. the columns will be re-arranged depending on the window/screen size.

As a Shiny user you do not need to know much about Bootstrap, but knowing a little about the grid system might help with developing Shiny apps with a more complex layout of controls.

Organising controls

The arguments to `fluidPage` are the (groups of) controls from top to bottom of the Shiny app.

The function `titlePanel("Title goes here")` adds a main title to the app.

Example 2.

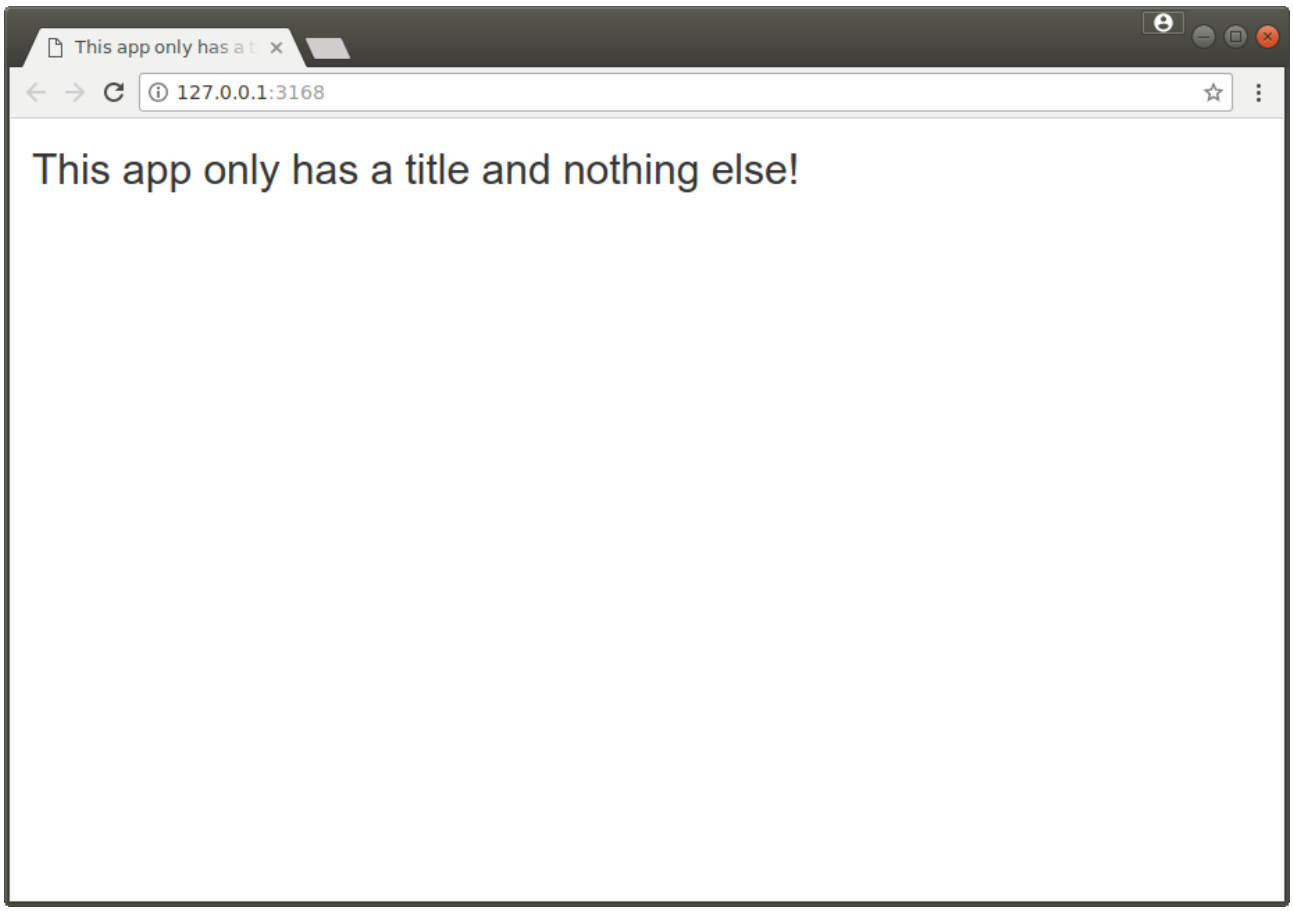


Figure 2: Screenshot of the app from example 2

The following `ui.R` creates a Shiny app which only has a title bar.

```
fluidPage(
    titlePanel("This app only has a title and nothing else!")
)
```

Layouts

If we specified the controls by adding them directly as arguments to `fluidPage` they would be simply arranged from top to bottom. Typically we want to arrange controls in a more sophisticated layout.

Sidebar layout The sidebar layout is the most common layout used in Shiny apps. It splits the screen into a narrow column containing the controls and a wide column for the outputs. We have used this layout in example 1.

The function `sidebarLayout` takes the content for the sidebar and the content for the main panel as its first two arguments. The sidebar argument is typically created using a call to the function `sidebarPanel` and the main panel using a call to the function `mainPanel`. A typical sidebar layout is thus set up using

```
sidebarLayout(
    sidebarPanel(
        # Controls and content for sidebar go here
    ),
    mainPanel(
        # Controls and content for main panel go here
    )
)
```

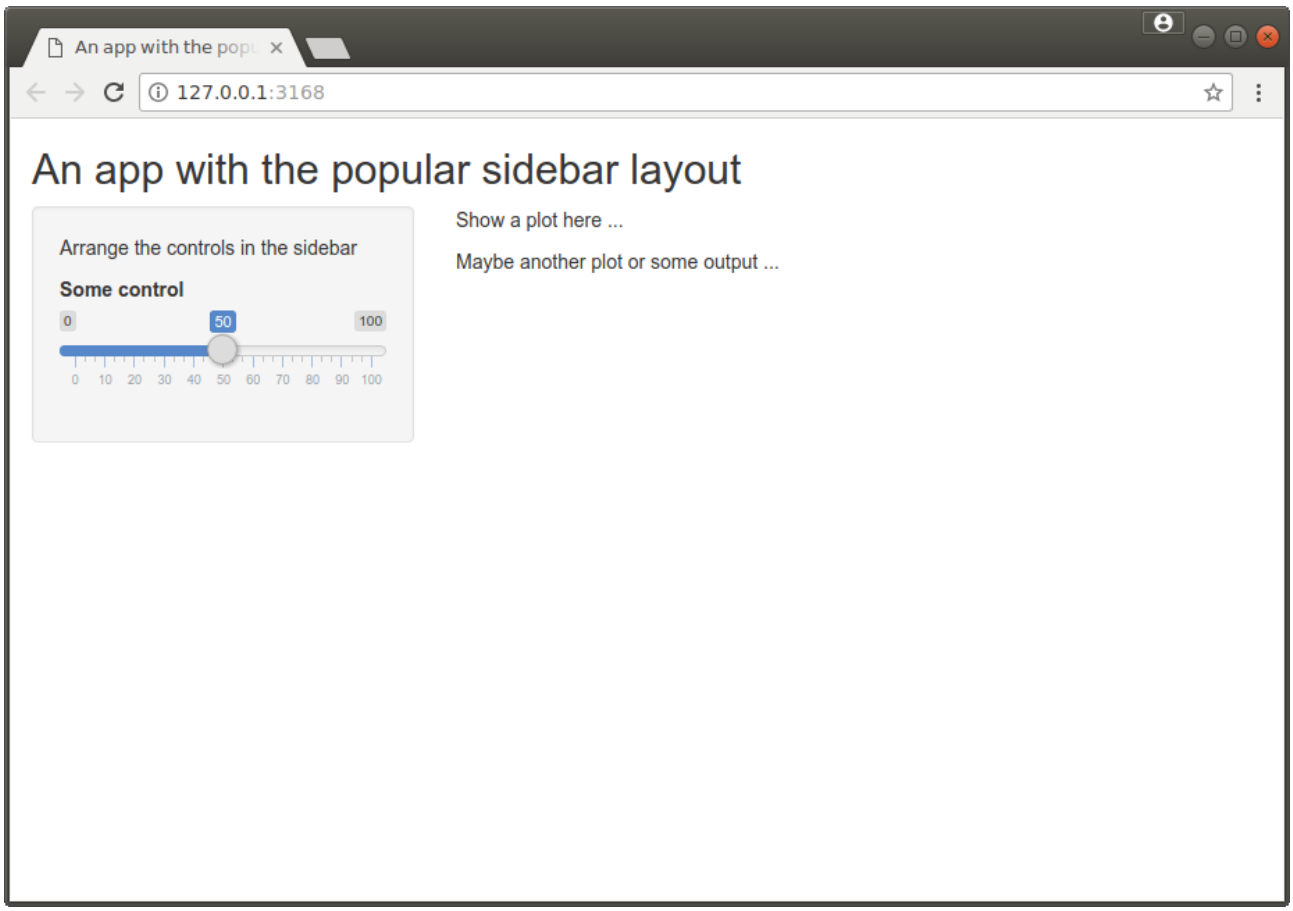


Figure 3: Screenshot of the app from example 3

Example 3.

The following `ui.R` creates a Shiny app which uses the sidebar layout. We will look at the functions adding content later on, so don't worry about them just now.

```
fluidPage(
    titlePanel("An app with the popular sidebar layout"),
    sidebarLayout(
        sidebarPanel(
            p("Arrange the controls in the sidebar"),
            sliderInput("ignored", "Some control", min=0, max=100, value=50)
        ),
        mainPanel(
            p("Show a plot here ..."),
            p("Maybe another plot or some output ...")
        )
    )
)
```

Manual grid layout A more flexible layout can be obtained by working with the Bootstrap 12 column layout. The function `fluidRow` create a row of columns. Each column is set up using the function `column`.

```
fluidRow(
    column(width,
        # Content of first column goes here
    ),
    column(width,
        # Content of second column goes here
    ),
    # More columns go here
```

```
)
```

For each row the widths of the columns have to be integers and should add up to at most 12.

You can use a `wellPanel` element to get an inset border and grey background.

Example 4.

The following `ui.R` creates a Shiny app using the Bootstrap grid system. The first row on columns splits the width a 6:3:3 ratio. The second row of columns splits the width in a 3:9 ratio. When the browser window is very narrow, all columns will be automatically stacked on top of each other (Bootstrap will do this for us automatically).

```
fluidPage(
    titlePanel("An app using the Bootstrap grid system"),
    fluidRow(
        column(6,
           h4("A wide column"),
           p("Try making the browser window a lot narrower ...")
        ),
        column(3,
            h4("A narrow column")
        ),
        column(3,
            wellPanel(
                h4("A narrow column in a well")
            )
        )
    ),
    fluidRow(
        column(3,
            wellPanel(
                h4("Another narrow column in a well")
          )
        ),
        column(9,
           h4("This is now a really wide column")
        )
    )
)
```

Tabs When an app shows a lot of content it can help to arrange the content in tabs. Tabs can be added using the function `tabsetPanel`, which takes as arguments the panels, created using the function `tabPanel`.

```
tabsetPanel(
    tabPanel("Title of first panel",
        # Content of first panel goes here
    ),
    tabPanel("Title of second panel",
        # Content of second panel goes here
    ),
    # More tabs go here
    )
```

Tabs can be used at any point in the Shiny app, you can even nest them.

Example 5.

The following `ui.R` creates a Shiny app using two tabs inside the main panel of a sidebar layout.

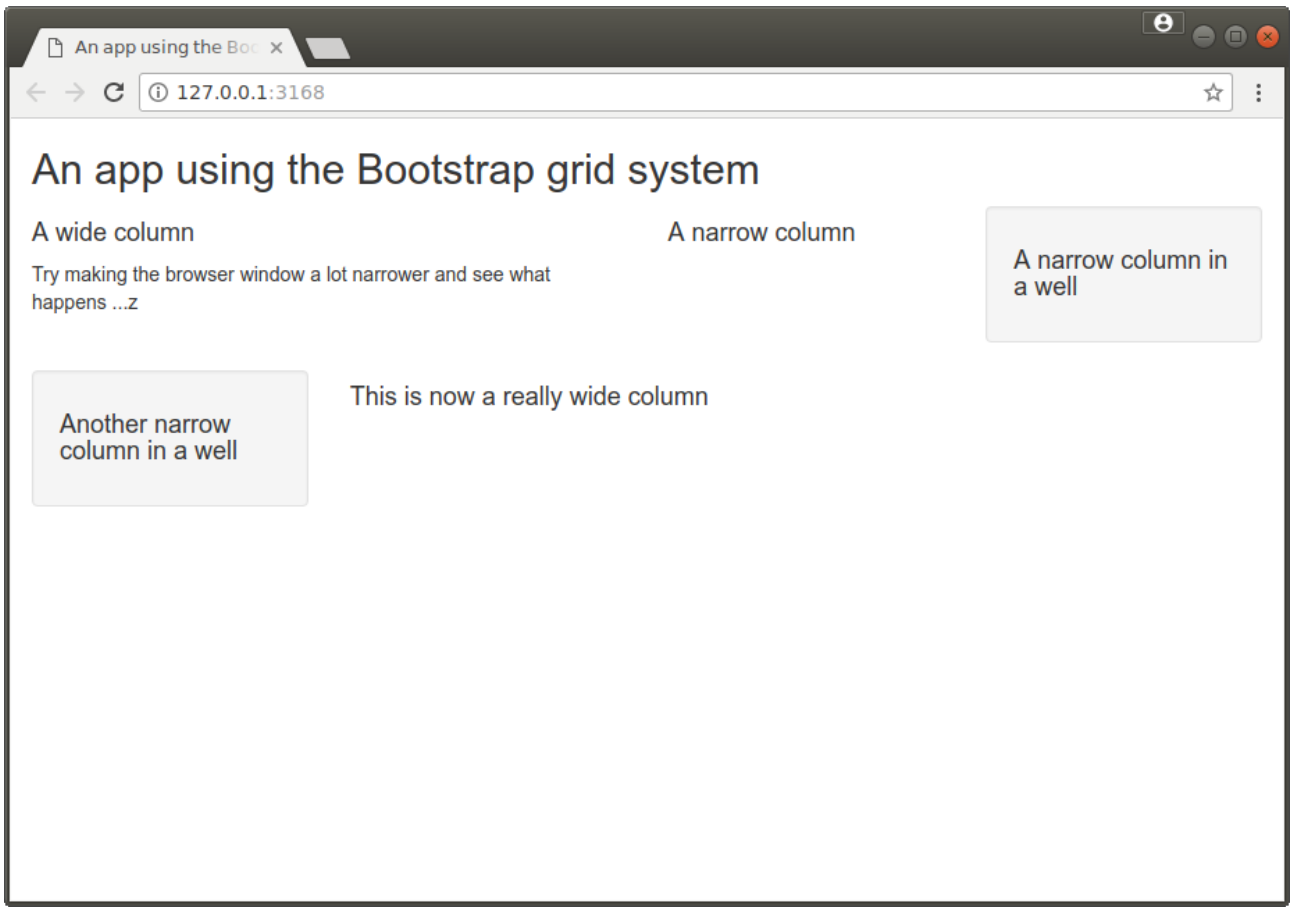


Figure 4: Screenshot of the app from example 4

```
fluidPage(
    titlePanel("An app with two tabs in the main panel")
    sidebarLayout(
        sidebarPanel(
            p("Arrange the controls in the sidebar"),
            sliderInput("ignored", "Some control", min=0, max=100, value=50)
        ),
        mainPanel(
            tabsetPanel(
                tabPanel("Plot",
                         p("Show a plot here ..."),
                         p("Click on the tab titles to change tabs ...")
                         ),
                tabPanel("Other output",
                          p("Another plot or some output ...")
                         )
            )
        )
    )
)
```

Shiny Layout Guide

https://shiny.rstudio.com/articles/layout-guide.html

The shiny layout guide contains more detailed information about creating layouts for shiny apps

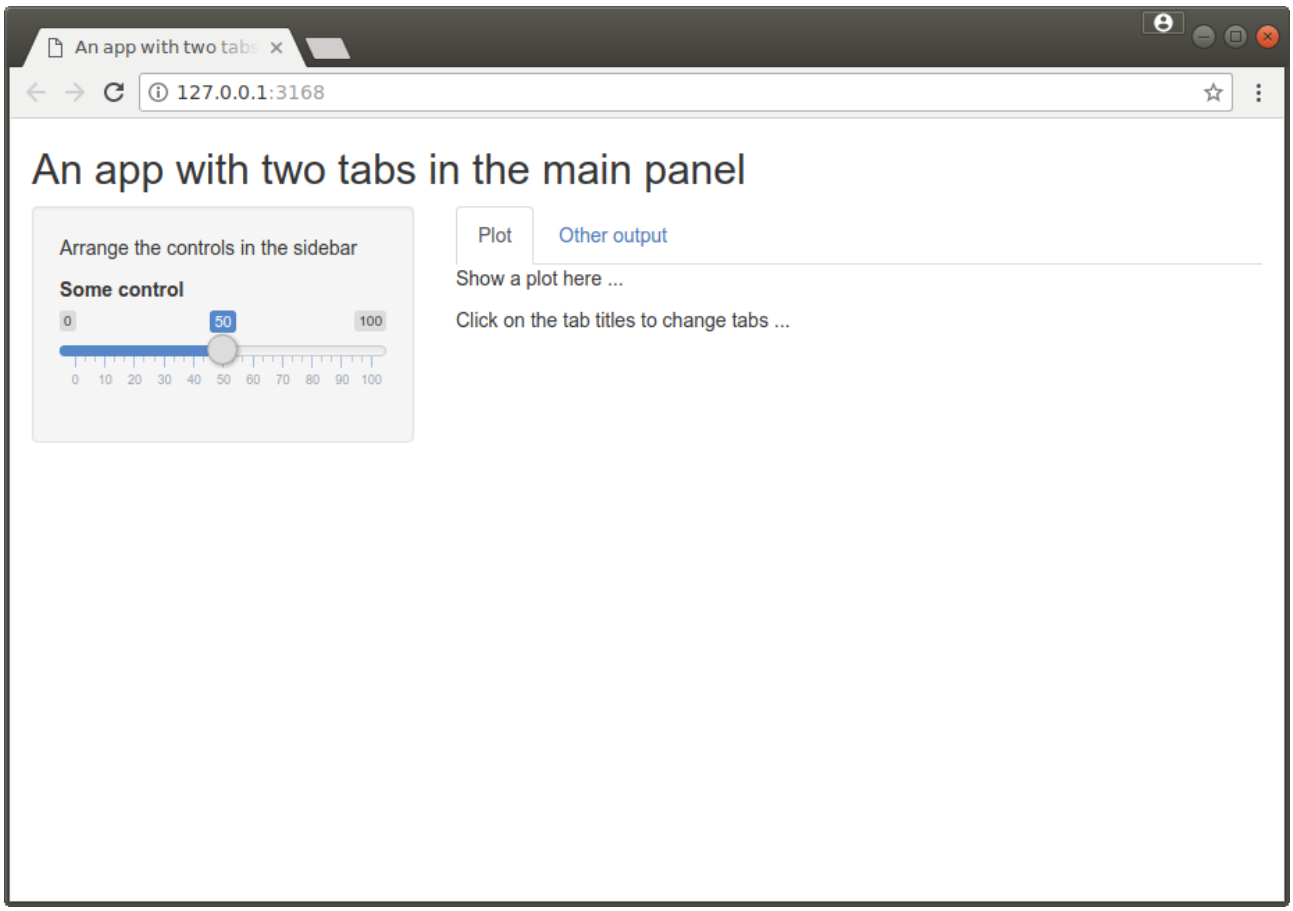


Figure 5: Screenshot of the app from example 5

Controls

Shiny has a wide range of input controls. Input controls allow the user to change parameters using the user interface.

The most common controls are given in the table below.

Function	Control widget
`actionButton`	Action Button
`checkboxInput`	Single check box
`checkboxGroupInput`	Group of check boxes
`dateInput`	Calendar to aid date selection
`dateRangeInput`	Pair of calendars for selecting a date range
`fileInput`	File upload control wizard
`helpText`	Help text that can be added to an input form
`numericInput`	Field to enter numbers
`radioButtons`	set of radio buttons
`selectInput`	Box with choices to select from
`selectizeInput`	Fancier box with choices to select from
`sliderInput`	Slider bar
`textInput`	Field to enter text

All these functions (with the exception of `helpText`) take as first argument the input id of the control. We can set this to any unique name. The name will be not visible to the user, but used by Shiny to refer to the object. The input id is required when we want to read out the value of an input control in `server.R`.

In addition to the input id, the above functions take additional arguments. The key arguments are showcased in the example below.

Because we have (not yet) coded up any application logic in `server.R` interacting with the input controls has no effect.

Example 6 (Showcase of popular input controls).

In this example we will use all the inputs listed above.

```
library(shiny)

# We will use this for categorical input. RHS is what is used inside app.
# LHS are the labels shown to the user
choices <- c("Option 1"="option1", "Option 2"="option2",
             "Option 3"="option3", "Option 4"="option4")

fluidPage(
    titlePanel("Popular input controls"),
    fluidRow(
        column(3,
            wellPanel(
                h4("Numerical inputs"),
                numericInput("numeric1", "numericInput", value=35,
                    min=0, max=100, step=0.1   # optional
                ),
                sliderInput("slider1","sliderInput (regular slider)",
                    value=35, min=0, max=100,
                    step=1                      # optional
                ),
                sliderInput("slider2","sliderInput (range slider)",
                    value=c(35, 48), min=0, max=100,
                    step=1                       # optional
                )
            )
        ),
        column(3,
            wellPanel(
                h4("Categorical inputs"),
                checkboxInput("checkbox1", "checkboxInput", value=TRUE),
                checkboxGroupInput("checkboxGroup1", "checkboxGroupInput",
                    selected=c("option1", "option3"),
                    choices=choices
                ),
                selectizeInput("selectize1", "selectizeInput", selected="option4",
                    choices=choices,
                    multiple=FALSE            # optional
                ),
                radioButtons("radioButtons1", "radioButton", selected="option3",
                    choices=choices
                )
            )
        ),
        column(3,
            wellPanel(
                h4("Date inputs"),
                dateInput("date1", "dateInput", value="2017-10-03",
                    min="2017-01-01", max="2017-12-31", # optional
                    format="dd/mm/yyyy"       # optional
                ),
                dateRangeInput("dateRange1", "dateRangeInput",
                    start="2017-10-03", end="2017-12-24",
                    min="2017-01-01", max="2017-12-31", # optional
                    format="dd/mm/yyyy"       # optional
                ),
```

```
                    sliderInput("siderDate1", "sliderInput for dates",
                        value=as.Date("2017-10-03"),
                        min=as.Date("2017-01-01"), max=as.Date("2017-12-31"), #optional
                        timeFormat="%d/%m/%Y"     # optional
                    ),
                    sliderInput("siderDate2", "sliderInput for dates (range slider)",
                        value=as.Date(c("2017-10-03","2017-12-24")),
                        min=as.Date("2017-01-01"), max=as.Date("2017-12-31"), #optional
                        timeFormat="%d/%m/%Y"     # optional
                    )
                )
            ),
            column(3,
                wellPanel(
                   h4("Text inputs"),
                   textInput("text1", "textInput", value="default content")
                ),
                wellPanel(
                    h4("Buttons"),
                    actionButton("button1", "actionButton",
                        icon=icon("paw")                # optional
                    )
                ),
                wellPanel(
                    h4("File uploads"),
                    fileInput("file1", "fileInput")
                )
            )
        ),
        # Ignore the output for the moment, we'll look at it in the next section
        h4("Reading out the values of the input controls in R (next section)"),
        verbatimTextOutput("inputvals")
    )
```

HTML elements

If you are familiar with using HTML you might want to add some custom HTML content to an app, but keep in mind that the design philosophy of Shiny is that the user would rarely need to do this (other than adding titles).

We can add regular HTML5 tags to Shiny apps using wrapper functions `tags$<HTMLtagname>(...)`. For example we can add a paragraph of text using

```
tags$p("This is a paragraph containing text.")
```

Common tags (`p`, `h1` to `h6`, `pre`, ...) are directly available, so we could have used

```
p("This is a paragraph containing text.")
```

Any named arguments are included in the HTML as attributes of the tag. For example, CSS style attributes can be set using the (optional) argument `style`. Tags can also be nested within each other.

Raw HTML can be included using the function `HTML`.

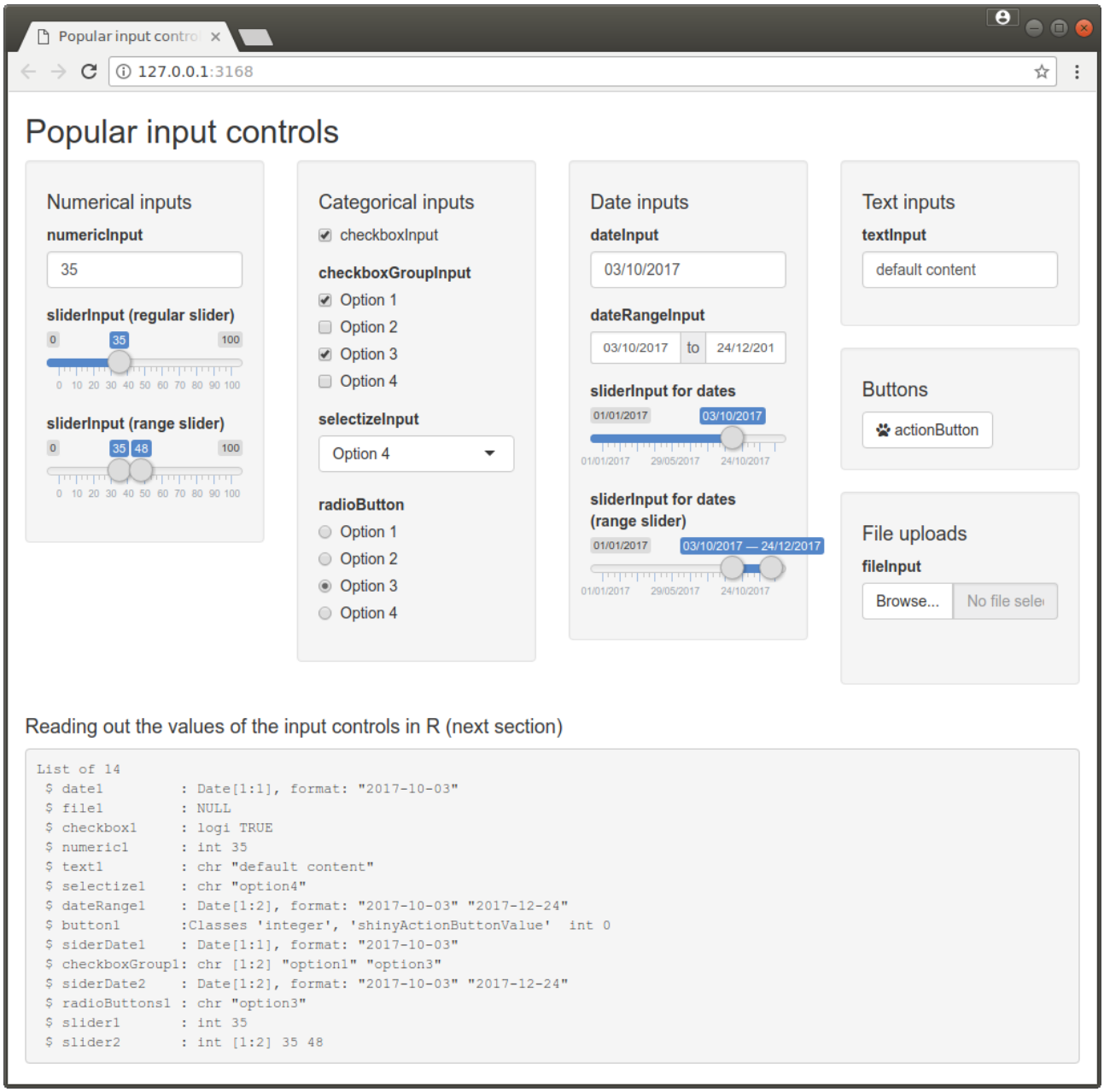


Figure 6: Screenshot of the app from example 6

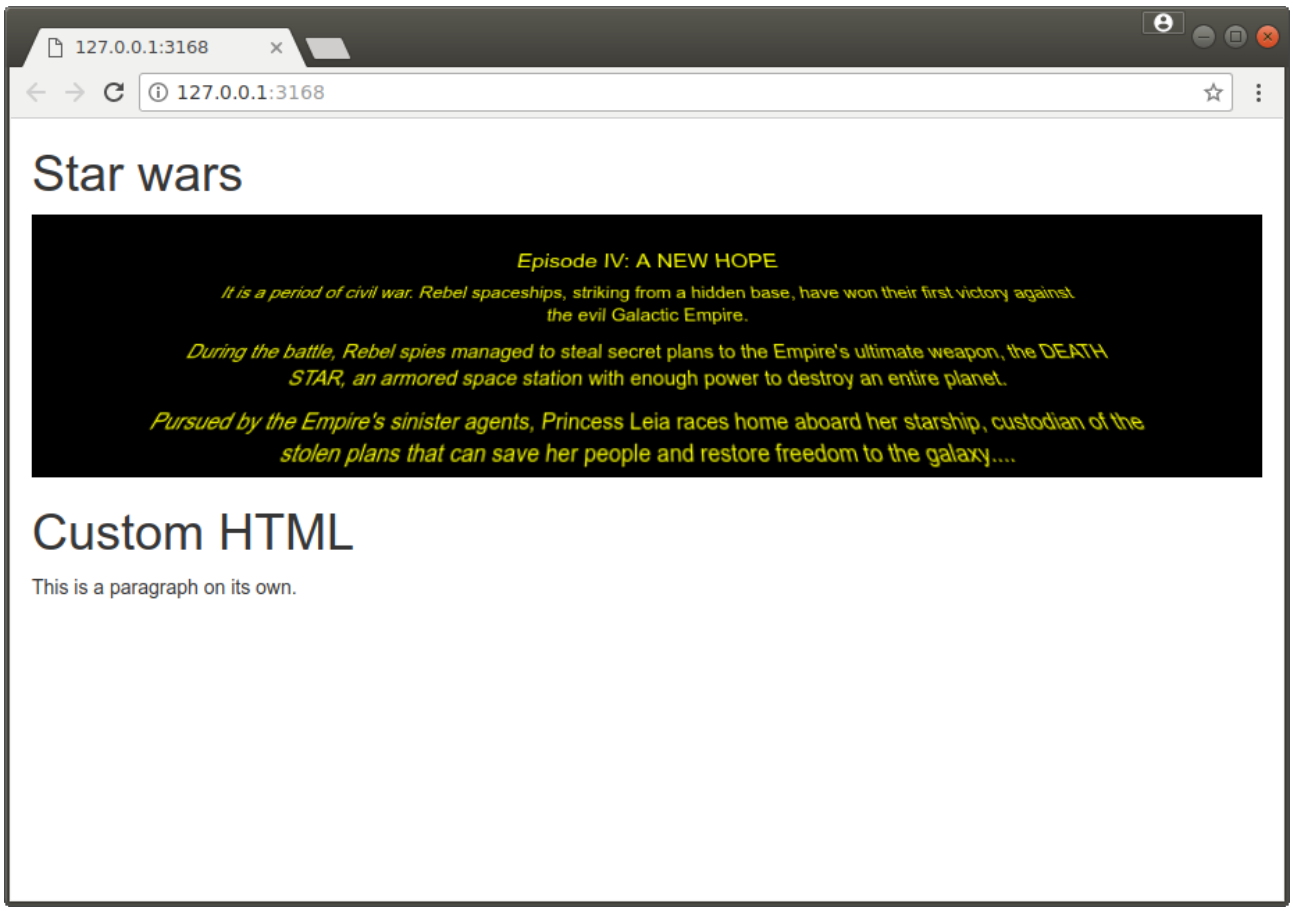


Figure 7: Screenshot of the app from example 7

Example 7 (HTML content).

This example shows how HTML content can be included into a Shiny app.

```
fluidPage(
    h1("Star wars"),
    div(
        div(
            h4("Episode IV: A NEW HOPE"),
            p(paste("It is a period of civil war. Rebel spaceships, striking from a",
                    "hidden base, have won their first victory against the evil",
                    "Galactic Empire.")),
            p(paste("During the battle, Rebel spies managed to steal secret plans",
                    "to the Empire's ultimate weapon, the DEATH STAR, an armored",
                    "space station with enough power to destroy an entire planet.")),
            p(paste("Pursued by the Empire's sinister agents, Princess Leia races",
                      "home aboard her starship, custodian of the stolen plans that",
                      "can save her people and restore freedom to the galaxy....")),
            style=paste("transform:rotateX(25deg); text-align: center; color:yellow;",
                        "font-size: 150%; width:75%; display: inline-block;")
        ),
        style=paste("perspective: 250px; perspective-origin: 50% 50%;",
                    "background-color: black; text-align:center;")
    ),
    h1("Custom HTML"),
    HTML("<p>This is a paragraph on its own.</p>")
)
```

I have only used `paste` to avoid having too long lines of code.

Data-driven components

So far we have only put together a user interface containing controls, but not data-driven components showing plots or other output. In order to have a component that shows output from R we need to set up this component in both `ui.R` and `server.R`. In `ui.R` we need to specify that a reactive component will occupy a certain place in the app. In `server.R` we need to specify a function which generates the content.

There are different types of content we might want to show in the app. The table below gives an overview of the most common content types.

Content type	Output function	Render function
Plots	`plotOutput`	`renderPlot`
Raw text / R output	`verbatimTextOutput`	`renderPrint`
Data (basic table)	`tableOutput`	`renderTable`
Data (fancy table)	`dataTableOutput`	`renderDataTable`
Download	`downloadButton`	`downloadHandler`
Shiny controls (adaptive)	`uiOutput`	`renderUI`

For every piece of content there needs to be a call to the output function in `ui.R` and a call to the corresponding render function in `server.R`.

The basic structure of `server.R` is

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

    output$outputid <-  renderTYPE( {
        # Generate content for this piece of content
    } )

    # Set more outputs
})
```

It is important that the input ids also match up. In `ui.R` the input id is the first argument to the output function. This input id label must then be used when updating the content in `server.R`: the value returned by the render function needs to stored in `output$outputid`.

For example, in example 1 we have created the plot output in `ui.R` using the command

```
plotOutput("densityPlot")
#          ^^^^^^^^^^^^^ output id
```

In `server.R` we have provided the code to create and update the plot using

```
output$densityPlot <-  renderPlot( { ... } )
#      ^^^^^^^^^^^ must be the same output id as used in ui.R
```

Note that we have to make sure the output id (‘ “densityPlot” ’) in ‘ui.R’ matches the one used in ‘server.R’.

All `renderXYZ` functions take an expression as first argument which will be evaluated to generate the corresponding output. Inside this expression we can access the values from the input components using `input$inputid`, where `inputid` is the label set for this component in `ui.R` (first argument to the functions creating the UI elements).

Example 8.

In this example we create three outputs: a plot (with a regression line, which is only meaningful if the data is suitably subset), a printed `summary` of a data set and table of the data. We also allow users to download the subset.

The example only has a single double-ended slider as input, which selects a subset of the motorcycle data from MASS using the specified range.

The interface is specified in `ui.R`.

```
fluidPage(
    titlePanel("Subsetting the motorcycle data"),
    sidebarLayout(
        sidebarPanel(
            sliderInput("range", "Subset range", min=0, max=60, value=c(0,60)),
#                       ^^^^^ Label used in server.R when querying slider
            downloadButton("download1", "Download subset", icon="download")
#                          ^^^^^^^^^^^  Label used in server.R for download
        ),
        mainPanel(
            plotOutput("plot1"),
#                      ^^^^^ Label used in server.R when updating plot
            verbatimTextOutput("text1"),
#                              ^^^^^ Label used in server.R when updating output
            dataTableOutput("table1")
#                           ^^^^^^ Label used in server.R when updating table
        )
    )
)
```

The server-side logic to produce the outputs is defined in `server.R`. For every call to `plotOutput`, `verbatimTextOutput`, `dataTableOutput` and `downloadButton` in `ui.R`, there is a matching call to `renderPlot`, `renderPrint`, `renderDataTable` and `downloadHandler` in `server.R`.

```
library(ggplot2)
library(MASS)

shinyServer(function(input, output) {

    output$plot1 <- renderPlot( {
#         ^^^^^ Label from ui.R
        mcycle2 <- subset(mcycle, times>=input$range[1] & times<=input$range[2])
#                                 Label from ui.R ^^^^^                   ^^^^^
        qplot(times, accel, data=mcycle2, xlim=range(mcycle$times),
              ylim=range(mcycle$accel)) + geom_smooth(method="lm")
    } )

    output$text1 <- renderPrint( {
#         ^^^^^ Label from ui.R
        mcycle2 <- subset(mcycle, times>=input$range[1] & times<=input$range[2])
#                                 Label from ui.R ^^^^^                   ^^^^^
        summary(mcycle2)
    } )

    output$table1 <- renderDataTable( {
#         ^^^^^ Label from ui.R
        subset(mcycle, times>=input$range[1] & times<=input$range[2])
#                     Label from ui.R ^^^^^                   ^^^^^
    } )

    output$download1 <- downloadHandler(
#         ^^^^^^^^^ Label from ui.R
        filename = "mcycle-subset.csv", # Filename shown to user in browser
        content = function(file) { # Function to generate the download file
            mcycle2 <- subset(mcycle, times>=input$range[1] & times<=input$range[2])
#                                    Label from ui.R ^^^^^                   ^^^^^
            write.csv(mcycle2, file, row.names=FALSE)
        }
    )
```

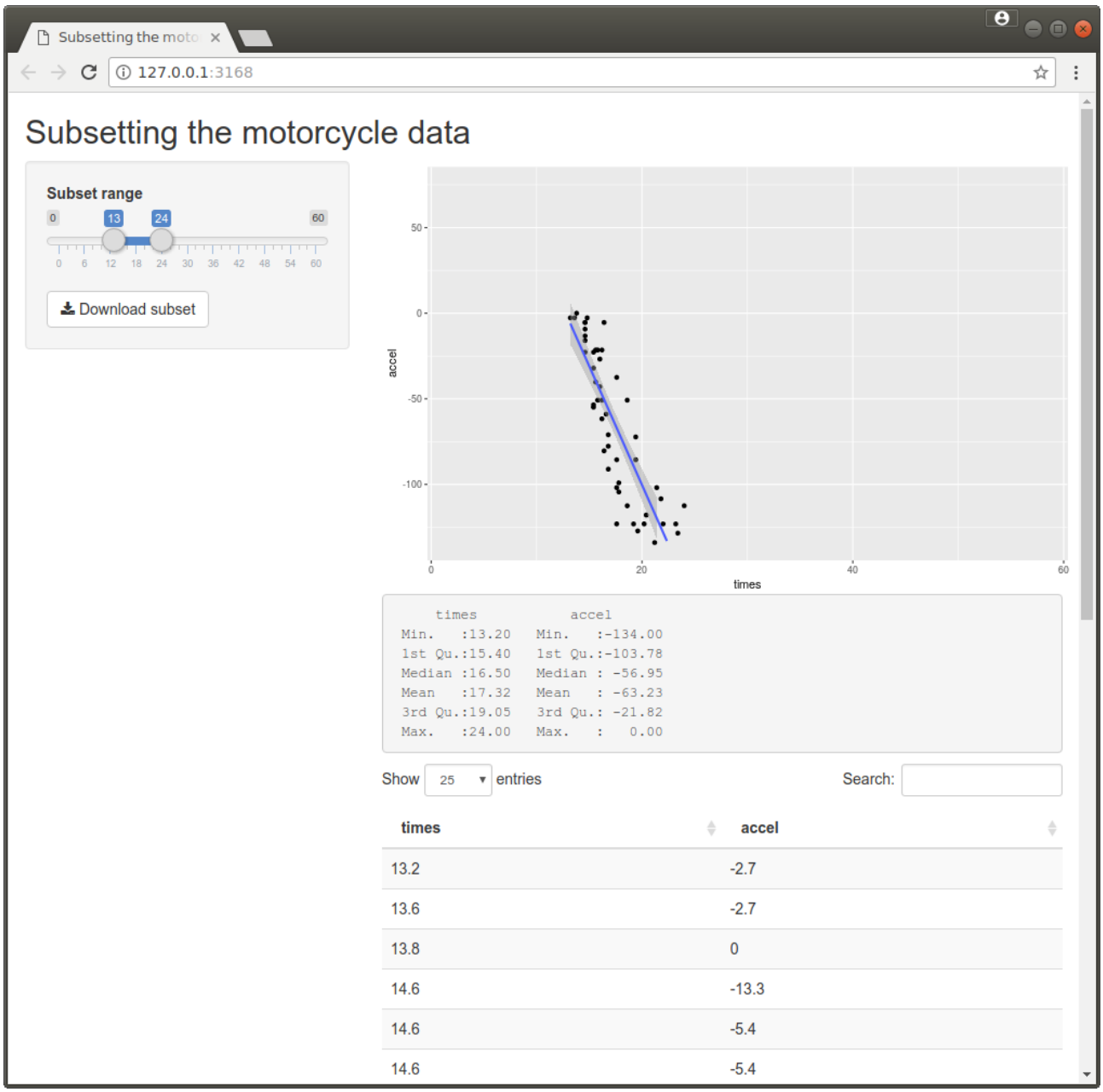


Figure 8: Screenshot of the app from example 8

```
})
```

For every output, we independently produce the subset of the motorcycle data. This doesn't seem efficient. In example 10 we will see how we can subset the data only once with a custom reactive expression.

Note that we have not told Shiny when to update a component. Shiny will figure out when to update what output all by itself. We will look at this in more detail in the next section.

Data-driven user interface

Sometimes we want to adapt the user interface based on the change to an input by the user. In other words, we might have to adapt parts of the user interface "on the fly". We can do this by using a `uiOutput` in `ui.R` and generating the corresponding content in `server.R` using `renderUI`. example 9 illustrates this.

Example 9.

In this example, we modify the app from example 8, so that the user can choose what output they want to have.

We add ratio buttons to the user interface and replace all outputs by a call `uiOutput`. This generates a placeholder we can then fill in `server.R`.

We thus use the following `ui.R`.

```
fluidPage(
    titlePanel("Subsetting the motorcycle data"),
    sidebarLayout(
        sidebarPanel(
            sliderInput("range", "Subset range", min=0, max=60, value=c(0,60)),
            radioButtons("outputType", "Select output type",
                         choices=c("Plot", "Summary", "Table"), selected="Plot")
        ),
        mainPanel(
            uiOutput("ui")   # Put adaptive UI component here
        )
    )
)
```

In `server.R` we generate the component to display based on the selection of the user. The content of the components is generated as before, so we leave these calls as they are.

```
library(ggplot2)
library(MASS)

shinyServer(function(input, output) {

    output$ui <- renderUI( { # Add the output component  based on the user's choice
        if (input$outputType=="Plot")
            output <- plotOutput("plot1")
        if (input$outputType=="Summary")
            output <- verbatimTextOutput("text1")
        if (input$outputType=="Table")
            output <- dataTableOutput("table1")
        output
    } )

    output$plot1 <- renderPlot( {
        mcycle2 <- subset(mcycle, times>=input$range[1] & times<=input$range[2])
        qplot(times, accel, data=mcycle2, xlim=range(mcycle$times),
              ylim=range(mcycle$accel)) + geom_smooth(method="lm")
    } )

    output$text1 <- renderPrint( {
        mcycle2 <- subset(mcycle, times>=input$range[1] & times<=input$range[2])
        summary(mcycle2)
    } )

    output$table1 <- renderDataTable( {
        subset(mcycle, times>=input$range[1] & times<=input$range[2])
    } )

})
```

In this example all we do is effectively show or hide output based on the user's choice. This can also be done using a `conditionalPanel`. The latter however requires a little knowledge of Javascript and JQuery, as the conditions have to specified in Javascript, rather than as an R expression.

From a user-interface perspective, a tab panel would probably have been best.

If all you want to change in the user interface is what value a user has selected it is typically easier to use the `update`

functions for the components, for example `updateRadioButtons` or `updateNumericInput`.

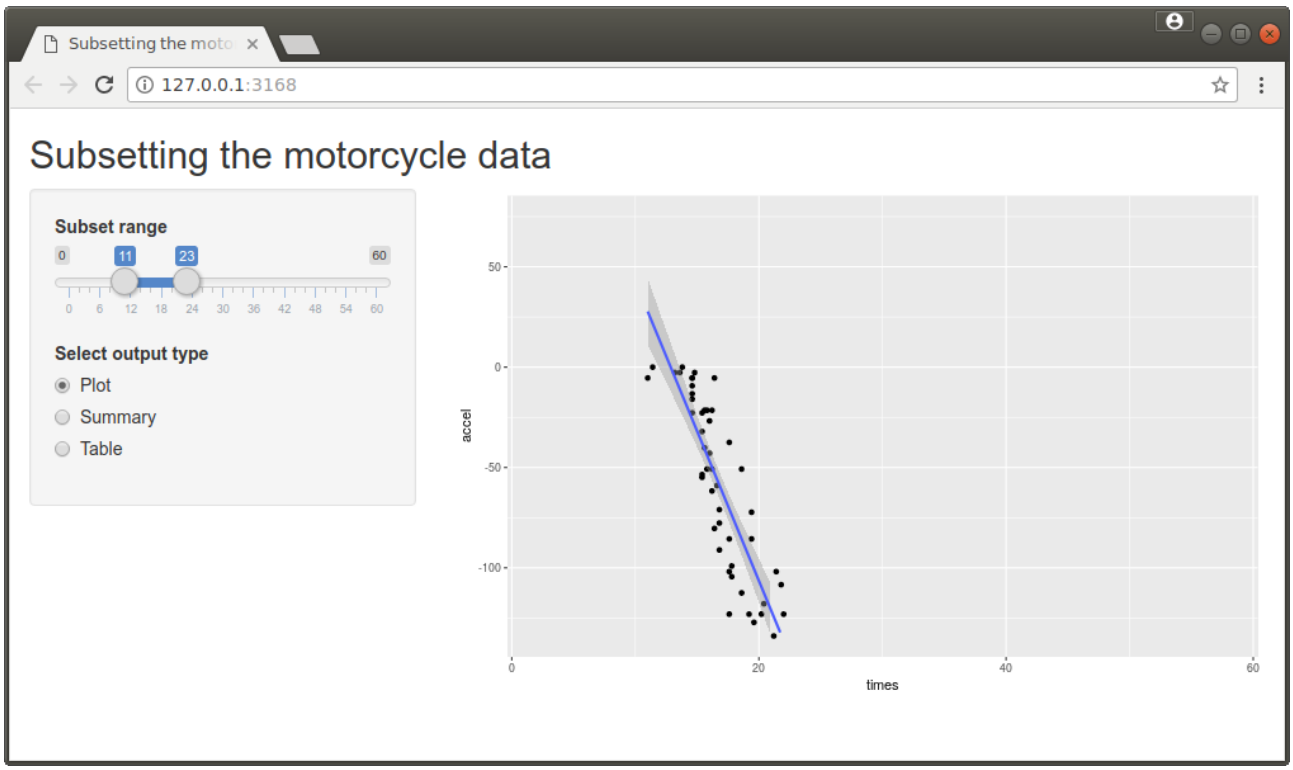


Figure 9: Screenshot of the app from example 9

Under the bonnet

Reactivity

Shiny automatically figures out whether and when to update output objects. It does so by monitoring which `input\$inputids` you use.

This is handy, but also means that you need to play by the rules. Otherwise the algorithms used by Shiny to figure out what depends on what will fail.

- Shiny's logic is based on the idea that a rendered object needs to be updated if and only if any of the `input\$inputids` used in `renderYXZ` change (or any other reactive expression used in `renderXYZ` changes – more on that later).
- This also means that you are only meant to access `input\$inputid` inside a call to `renderXYZ` (and also inside custom reactive expressions). The video has an example of what goes wrong if you use an input outside.

(`downloadHandler` counts as a `renderXYZ` function in this context).

You can visualise how inputs and output objects relate to each other by activating the reactivity log. Before you start your app run `options(shiny.reactlog=TRUE)`. Then press `CTRL/Command-F3` after you have used the app. Shiny then opens a new window showing how the components depend. (There is a slider at the top which lets you move in time, you typically want to drag this to the right).

Custom reactive expressions

Often output components are related to each other and there is some commonality between how the outputs are computed: if two outputs perform at least in parts the same calculation, it would make sense to "share" these calculations so that they are not re-run unnecessarily.

If we want to perform a "shared" calculation if an input changes we cannot simply put the code outside `renderXYZ`. Shiny would not know what this code needs to re-run every time an input is updated.

We can however tell Shiny to do so by using a custom reactive expression. For this you wrap your calculation in `reactive(...)` and place it inside `shinyServer`.

```
reactive.expression <- reactive( {
   # Perform calculation here
} )
```

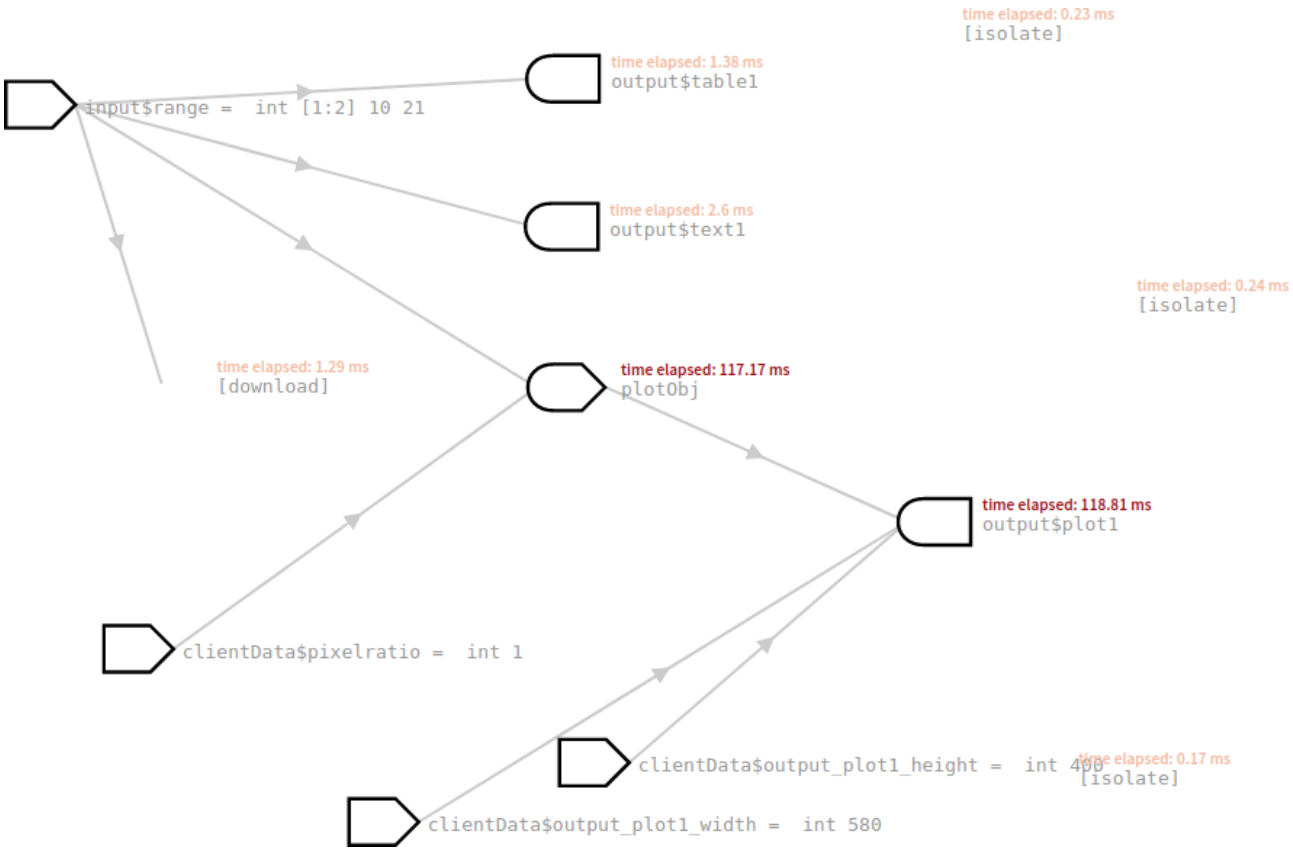


Figure 10: Shiny reactivity log of the app from example 8

Custom reactive expressions will be monitored and updated by Shiny in the same way as output object created using `renderXYZ`. We can now use the result from that calculation in the `renderXYZ` functions and Shiny will make sure that this calculation will not be repeated unnecessarily.

Note that `reactive` returns a function, so if you want to use the value of `reactive.expression` you have to use `reactive.expression()`.

Example 10.

In example 8 we subset the data set `mcycle` independently three times: once in `renderPlot`, once in `renderPrint` and once in `renderDataTable` (as well as in `downloadHandler`, but download data is only generated when the user clicks the button). So every time a user has moved the slider, we perform the subsetting three times. This isn't a good idea for two reasons: first of all it leads to redundant code, which is a bad thing in itself. It also impacts on the performance and responsiveness of your app (though not that much in our toy case as the data set is tiny, so subsetting is quick).

If we want to generate `mcycle2` outside the `renderXYZ` functions we need to use a custom reactive expression.

We can use the same `ui.R` as in example 8, however `server.R` changes.

```
library(ggplot2)
library(MASS)

shinyServer(function(input, output) {

    mcycle2 <- reactive( {   # Define reactive object mcycle2
       subset(mcycle, times>=input$range[1] & times<=input$range[2])
#                  Label from ui.R ^^^^^                    ^^^^^
    } )
```

```
    output$plot1 <- renderPlot( {
        qplot(times, accel, data=mcycle2(), xlim=range(mcycle$times),
#                                ^^^^^^^^^ Get the reactive object
              ylim=range(mcycle$accel)) + geom_smooth(method="lm")
    } )

    output$text1 <- renderPrint( {
        summary(mcycle2())
#               ^^^^^^^^^ Get the reactive object
    } )

    output$table1 <- renderDataTable( {
        mcycle2()
#       ^^^^^^^^^ Get the reactive object
    } )

    output$download1 <- downloadHandler(
        filename = "mcycle-subset.csv",
        content = function(file) {
            write.csv(mcycle2(), file, row.names=FALSE)
#                     ^^^^^^^^^ Get the reactive object
        }
    )

})
```

The Shiny reactivity log will now change. Changes to the inputs now feeds into `mcycle2` which then in turn feeds into the output objects.

Isolating reactive expressions

Sometimes you want to use an input value, but you do not want to update that component every time a user changes an input. This is typically the case if you need to perform expensive computations: rather than updating the output every time a slider is moved you are probably better off providing the user with a "Run" or "Update" button, which triggers the expensive computations.

This means we need to be able to do two things.

- We need to get the button to trigger the computations. We can do this by simply including `input\$buttonname` in `renderXYZ` code that generates the outputs that need to be calculated. (Shiny only checks whether an input is accessed, it doesn't mind if you do nothing more with it). Alternatively, we could use the function

```
observeEvent(input$buttonname, {
   # Actions to perform when input$buttonname fires
} )
```

to explicitly trigger an action. - We need to stop the outputs from reacting to changes in the other input values. We can achieve this using the function `isolate`. If we use `isolate(input\$inputid)` to access the input `inputid` instead of just `input\$inputid`, Shiny will not treat this output as depending on `inputid`: it will thus not update the output automatically whenever `input\$inputid` changes.

Example 11.

In example 1 we simulated from a sample from the normal distribution with the user being able to choose sample size, mean and standard deviation. Suppose we want to add a button to trigger the simulation. Moving the sliders on its own is not meant to trigger a new simulation.

The file `ui.R` is pretty much the same as in example 1, except for the button we have added.

```
fluidPage(
  titlePanel("Our first Shiny app (modified)"),
```

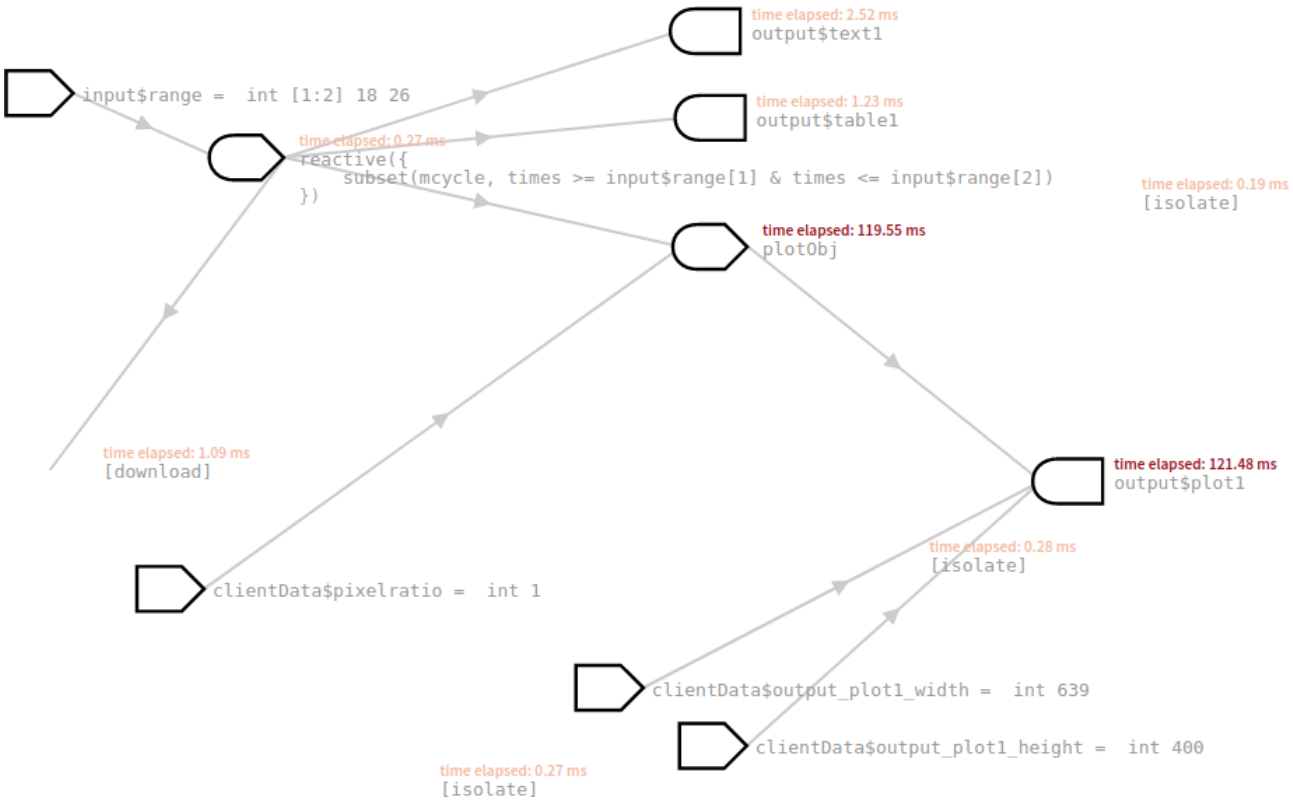


Figure 11: Shiny reactivity log of the app from example 10. The output objects now depend on the inputs through the reactive expression `mcycle2`.

```
    sidebarLayout(
        sidebarPanel(
            sliderInput("n",
                        "Number of observations"  ,
                        min=1, max=1000, step=1,
                        value = 100
                        ),
            sliderInput("mu",
                        "Mean",
                        min=-10, max=10, step=0.1,
                        value = 0
                        ),
            sliderInput("sigma",
                        "Standard deviation",
                        min=0, max=5, step=0.1,
                        value = 1
                        ),
            actionButton("simulate",                # Name of new action variable
                         "Simulate")                # Label shown on new button
        ),

      # The main panel is typically used for displaying R output
        mainPanel(h2("Estimated Density" ),
                  plotOutput("densityPlot")
                  )
    )
)
```

In `server.R` we make the `plotOutput` depend on the button by simply putting in `input\$simulate`. We wrap `input\$n`, `input\$mu` and `input\$sigma` in `isolate`, so that the `plotOutput` does not get triggered by these controls any more.

```
library(ggplot2)

shinyServer(function(input, output) {

    output$densityPlot <-  renderPlot( {
        input$simulate               # Force dependency on the button
        n <- isolate(input$n)        # Prevent dependency on n
        mu <- isolate(input$mu)      # Prevent dependency on mu
        sigma <- isolate(input$sigma) # Prevent dependency on sigma
        x <- rnorm(n, mu, sigma)
        qplot(x, geom="density", col="estimated") +
            geom_rug() + xlim(-10,10) + ylim(0,0.75) +
            stat_function(fun=function(x) dnorm(x, mu, sigma), aes(colour="exact"))
     })

})
```

Alternatively, we could have used

```
library(ggplot2)

shinyServer(function(input, output) {

    observeEvent(input$simulate, {    # Run this code if simulate button is pressed
        output$densityPlot <-  renderPlot( {
            n <- isolate(input$n)         # Prevent dependency on n
            mu <- isolate(input$mu)       # Prevent dependency on mu
            sigma <- isolate(input$sigma) # Prevent dependency on sigma
            x <- rnorm(n, mu, sigma)
            qplot(x, geom="density", col="estimated") +
                geom_rug() + xlim(-10,10) + ylim(0,0.75) +
                stat_function(fun=function(x) dnorm(x, mu, sigma), aes(colour="exact"))
        })

    }, ignoreNULL=FALSE)              # ignoreNULL=FALSE triggers the event
                                      # when the app starts (otherwise no
                                      # density would be initially visible)

})
```

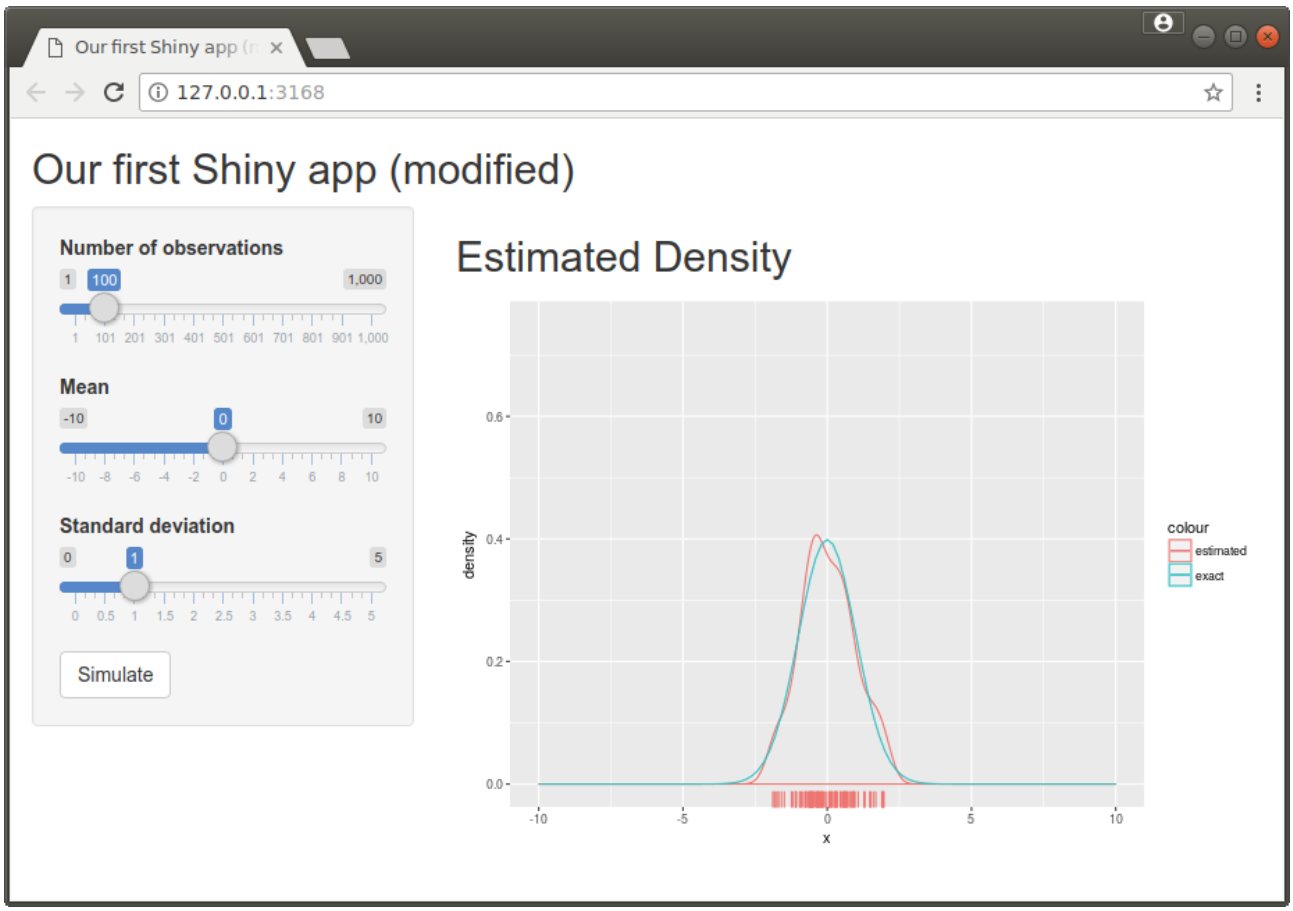


Figure 12: Screenshot of the app from example 11

When will which code get run?

Code inside a reactive statement (`reactive` or `renderXYZ`) will get run every time the user changes a inputs used inside that statement. The code snippet below shows in the comments when code placed in other parts of `server.R` gets run.

```
# Code put here gets run once when the server is being started
# Code put here creates one object shared between sessions

shinyServer(function(input, output) {

    # Code here is run once per client connection ("session")
    # Code here creates one object per client connection ("session")

    output$outputid <-  renderTYPE( {
        # Code here is run every time an input is being changed
    } )

    # ...

})
```

You can exploit this to speed up your app. If, for example, all users use the same big data set, it is probably best if you load it outside `shinyServer`. That way the data set is only loaded once and we also only keep one copy in memory.

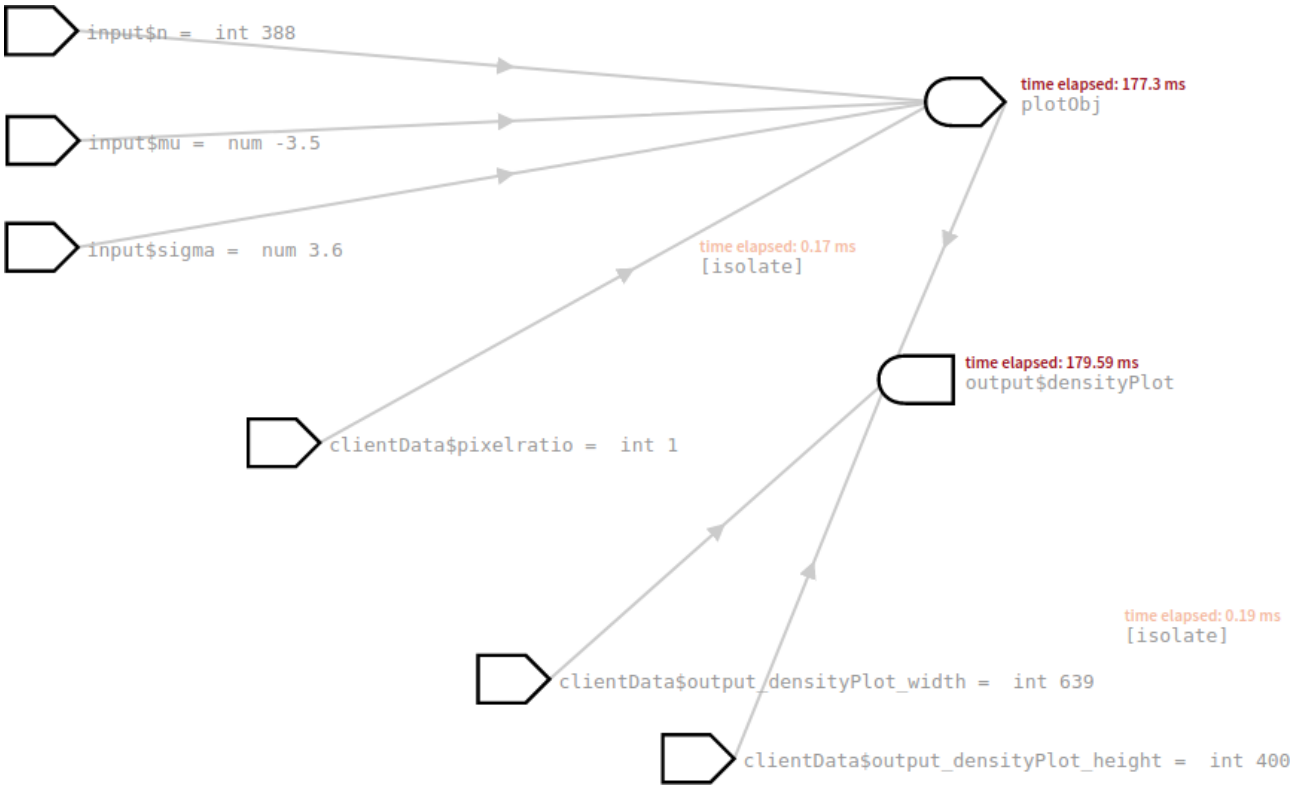


Figure 13: Shiny reactivity log of the app from example 1. Changes in the input sliders will trigger a new plot.

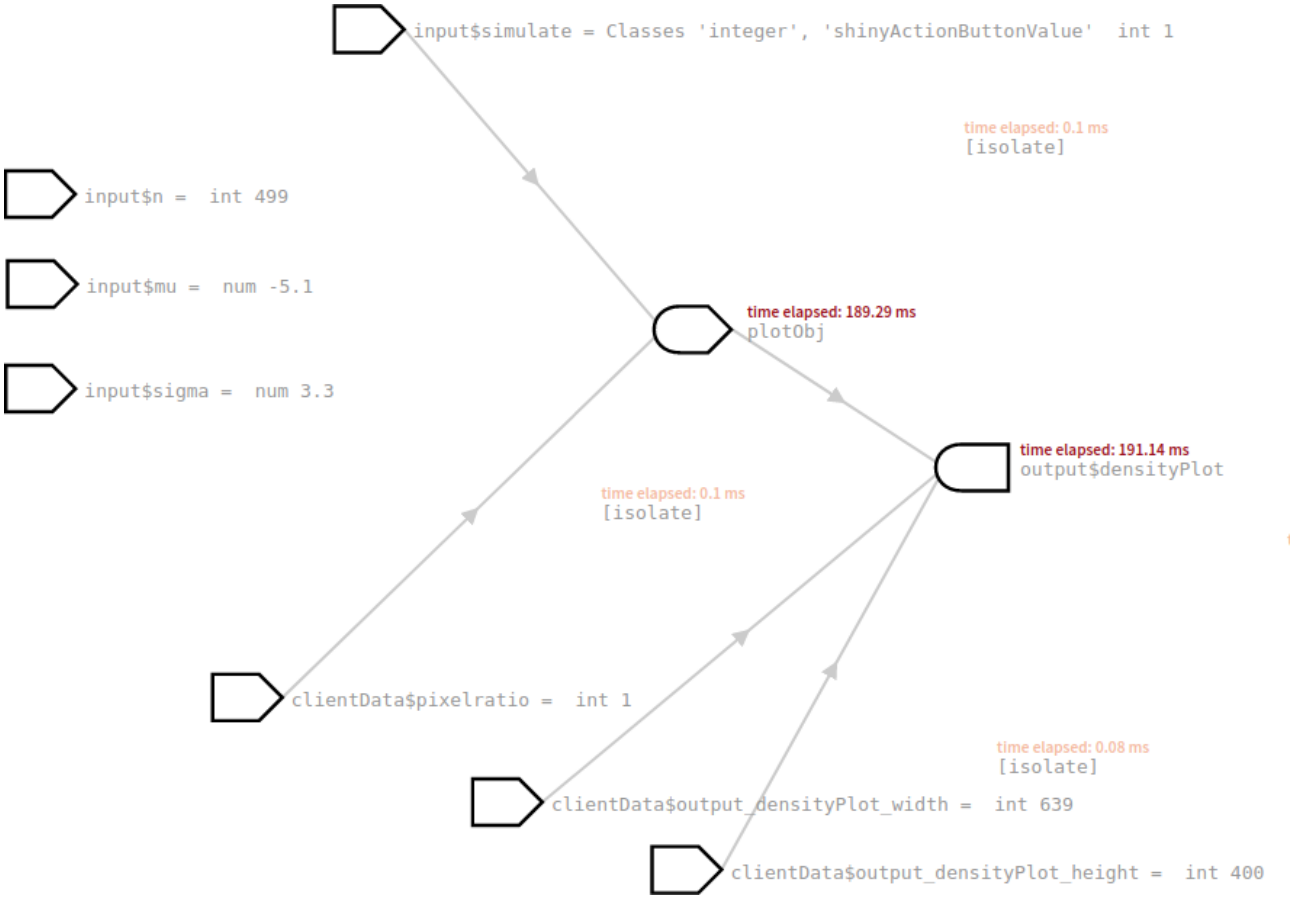


Figure 14: Shiny reactivity log of the app from example 11. Only pressing the "Simulate" button will trigger a new plot.