

# Contents

<b>Index</b>	<b>7</b>
<b>1 Tutorial</b>	<b>7</b>
1.1 Teach yourself Shiny . . . . .	7
1.2 Are you ready for Shiny? . . . . .	9
1.3 The Shiny Webinar . . . . .	14
1.4 LESSON 1:Welcome to Shiny . . . . .	16
1.5 LESSON 2:Build a user-interface . . . . .	23
1.6 LESSON 3:Add control widgets . . . . .	33
1.7 LESSON 4:Display reactive output . . . . .	40
1.8 LESSON 5:Use R scripts and data . . . . .	46
1.9 LESSON 6:Use reactive expressions . . . . .	55
1.10 LESSON 7:Share your apps . . . . .	61
<b>2 Articles</b>	<b>72</b>
2.1 Articles . . . . .	72
2.2 The basic parts of a Shiny app . . . . .	75
2.3 How to build a Shiny app . . . . .	83
2.4 How to launch a Shiny app . . . . .	89
2.5 How to get help . . . . .	90
2.6 Single-file Shiny apps . . . . .	92
2.7 App formats and launching apps . . . . .	94
2.8 Persistent data storage in Shiny apps . . . . .	98
2.9 Application layout guide . . . . .	109
2.10 Display modes . . . . .	122
2.11 Tabsets . . . . .	128
2.12 Customize your UI with HTML . . . . .	132
2.13 Build your entire UI with HTML . . . . .	140
2.14 Build a dynamic UI that reacts to user . . . . .	144
2.15 Shiny HTML Tags Glossary . . . . .	147
2.16 Progress indicators . . . . .	154
2.17 Getting started with shinyapps.io . . . . .	160
2.18 Setting up custom domains with . . . . .	170
2.19 Scaling and Performance Tuning with . . . . .	172
2.20 Share data across sessions with . . . . .	177
2.21 Migrating shinyapps.io authentication . . . . .	180
2.22 Introduction to Shiny Server . . . . .	181
2.23 Save your app as a function . . . . .	185
2.24 Sharing apps to run locally . . . . .	194
2.25 Introduction to R Markdown . . . . .	197
2.26 Introduction to interactive documents . . . . .	208
2.27 R Markdown integration in the . . . . .	214

2.28	The R Markdown Cheat sheet . . . . .	222
2.29	Using Action Buttons . . . . .	224
2.30	Using sliders . . . . .	231
2.31	Help users download data from your . . . . .	234
2.32	Using selectize input . . . . .	237
2.33	Render images in a Shiny app . . . . .	241
2.34	How to use DataTables in a Shiny . . . . .	246
2.35	Reactivity: An overview . . . . .	250
2.36	Stop reactions with isolate() . . . . .	256
2.37	Execution scheduling . . . . .	260
2.38	How to understand reactivity in R . . . . .	266
2.39	Write error messages for your UI with . . . . .	276
2.40	Scoping rules for Shiny apps . . . . .	287
2.41	Debugging techniques for Shiny apps . . . . .	291
2.42	Learn about your user with . . . . .	293
2.43	Unicode characters in Shiny apps . . . . .	298
2.44	Style your apps with CSS . . . . .	303
2.45	Build custom input objects . . . . .	312
2.46	Build custom output objects . . . . .	317
2.47	Add Google Analytics to a Shiny app . . . . .	320
2.48	How to create User Privileges . . . . .	329
2.49	Allow different libraries for different . . . . .	332
2.50	Interactive plots . . . . .	336
2.51	Selecting rows of data . . . . .	339
2.52	Interactive plots - advanced . . . . .	346
2.53	Upgrade notes for Shiny 0.11 . . . . .	349
2.54	Upgrade notes for Shiny 0.12 . . . . .	351
<b>3</b>	<b>Function Reference</b>	<b>352</b>
3.1	Function reference version 0.12.1 . . . . .	352
3.2	Panel with absolute positioning . . . . .	358
3.3	Create a Bootstrap page . . . . .	359
3.4	Create a column within a UI definition . . . . .	360
3.5	Conditional Panel . . . . .	362
3.6	Create a page with a fixed layout . . . . .	364
3.7	Create a page with fluid layout . . . . .	366
3.8	Create a header panel . . . . .	369
3.9	Create a help text element . . . . .	370
3.10	Create an icon . . . . .	371
3.11	Create a main panel . . . . .	373
3.12	Create a page with a top level . . . . .	374
3.13	Create a navigation list panel . . . . .	377
3.14	Create a page with a sidebar . . . . .	379
3.15	Layout a sidebar and main area . . . . .	381
3.16	Create a sidebar panel . . . . .	383
3.17	Create a tab panel . . . . .	385
3.18	Create a tabset panel . . . . .	387
3.19	Create a panel containing an . . . . .	389

3.20	Input panel . . . . .	390
3.21	Flow layout . . . . .	391
3.22	Split layout . . . . .	393
3.23	Lay out UI elements vertically . . . . .	395
3.24	Create a well panel . . . . .	396
3.25	Load the MathJax library and typeset . . . . .	397
3.26	Action button/link . . . . .	398
3.27	Checkbox Group Input Control . . . . .	399
3.28	Checkbox Input Control . . . . .	401
3.29	Create date input . . . . .	402
3.30	Create date range input . . . . .	405
3.31	File Upload Control . . . . .	409
3.32	Create a numeric input control . . . . .	410
3.33	Create radio buttons . . . . .	411
3.34	Create a select list input control . . . . .	413
3.35	Slider Input Widget . . . . .	415
3.36	Create a submit button . . . . .	417
3.37	Create a text input control . . . . .	418
3.38	Create a password input control . . . . .	419
3.39	Change the value of a checkbox . . . . .	420
3.40	Change the value of a checkbox . . . . .	422
3.41	Change the value of a date input on . . . . .	424
3.42	Change the start and end values of a . . . . .	426
3.43	Change the value of a number input . . . . .	428
3.44	Change the value of a radio input on . . . . .	430
3.45	Change the value of a select input on . . . . .	432
3.46	Change the value of a slider input on . . . . .	434
3.47	Change the selected tab on the client . . . . .	436
3.48	Change the value of a text input on . . . . .	437
3.49	Create an HTML output element . . . . .	439
3.50	Create an plot or image output . . . . .	440
3.51	Set options for an output object. . . . .	445
3.52	Create a table output element . . . . .	446
3.53	Create a text output element . . . . .	448
3.54	Create a verbatim text output element . . . . .	449
3.55	Create a download button or link . . . . .	450
3.56	Reporting progress (object-oriented . . . . .	451
3.57	Reporting progress (functional API) . . . . .	453
3.58	HTML Builder Functions . . . . .	455
3.59	Mark Characters as HTML . . . . .	457
3.60	Include Content From a File . . . . .	458
3.61	Include content only once . . . . .	459
3.62	HTML Tag Object . . . . .	460
3.63	Validate proper CSS formatting of a . . . . .	462
3.64	Evaluate an expression using tags . . . . .	463
3.65	Plot Output . . . . .	465
3.66	Text Output . . . . .	466
3.67	Printable Output . . . . .	468

3.68 Table output with the JavaScript . . . . .	470
3.69 Image file output . . . . .	472
3.70 Table Output . . . . .	474
3.71 UI Output . . . . .	475
3.72 File Downloads . . . . .	476
3.73 Plot output (deprecated) . . . . .	477
3.74 Print output (deprecated) . . . . .	478
3.75 Table output (deprecated) . . . . .	479
3.76 Text output (deprecated) . . . . .	480
3.77 UI output (deprecated) . . . . .	481
3.78 Scheduled Invalidation . . . . .	482
3.79 Checks whether an object is a . . . . .	484
3.80 Create a non-reactive scope for an . . . . .	485
3.81 Make a reactive variable . . . . .	487
3.82 Create a reactive observer . . . . .	488
3.83 Event handler . . . . .	490
3.84 Create a reactive expression . . . . .	493
3.85 Reactive file reader . . . . .	495
3.86 Reactive polling . . . . .	497
3.87 Timer . . . . .	499
3.88 Create an object for storing reactive . . . . .	501
3.89 Convert a reactivevalues object to a . . . . .	503
3.90 Reactive domains . . . . .	504
3.91 Reactive Log Visualizer . . . . .	505
3.92 Create a Shiny UI handler . . . . .	506
3.93 Define Server Functionality . . . . .	507
3.94 Run Shiny Application . . . . .	509
3.95 Run Shiny Example Applications . . . . .	511
3.96 Run a Shiny application from a URL . . . . .	512
3.97 Stop the currently running Shiny app . . . . .	514
3.98 Create a web dependency . . . . .	515
3.99 Resource Publishing . . . . .	516
3.100Register an Input Handler . . . . .	517
3.101Deregister an Input Handler . . . . .	519
3.102Mark a function as a render function . . . . .	520
3.103Validate input values and other . . . . .	521
3.104Session object . . . . .	525
3.105Convert an expression to a function . . . . .	527
3.106Install an expression as a function . . . . .	529
3.107Parse a GET query string from a URL . . . . .	530
3.108Run a plotting function and save the . . . . .	532
3.109Make a random number generator . . . . .	533
3.110Print message for deprecated . . . . .	534
3.111Collect information about the Shiny . . . . .	535
3.112Global options for Shiny . . . . .	536
3.113Find rows of data that are selected by . . . . .	538
3.114Create an object representing . . . . .	540
3.115Create an object representing click . . . . .	541

3.116	Create an object representing . . . . .	542
3.117	Create an object representing hover . . . . .	543
3.118	Find rows of data that are near a . . . . .	544
3.119	Create a Shiny app object . . . . .	546
3.120	Evaluate an expression without a . . . . .	548
3.121	OVERVIEW . . . . .	549
3.122	Create a web dependency . . . . .	550
3.123	Resource Publishing . . . . .	551
3.124	Register an Input Handler . . . . .	553
3.125	Deregister an Input Handler . . . . .	555
3.126	Mark a function as a render function . . . . .	556
3.127	Validate input values and other . . . . .	557
3.128	Session object . . . . .	562
3.129	Convert an expression to a function . . . . .	564
3.130	Install an expression as a function . . . . .	566
3.131	Parse a GET query string from a URL . . . . .	568
3.132	Run a plotting function and save the . . . . .	570
3.133	Make a random number generator . . . . .	572
3.134	Print message for deprecated . . . . .	574
3.135	Collect information about the Shiny . . . . .	575
3.136	Global options for Shiny . . . . .	576
3.137	Create a Shiny app object . . . . .	578
3.138	Evaluate an expression without a . . . . .	581
<b>A</b>	<b>Program listings to create this document</b>	<b>582</b>
A.1	recipe.txt . . . . .	582
A.2	stg02.mkPdfSet.py . . . . .	582
A.3	fulltex.bat . . . . .	585

# Index

- animate, 231, 232, 415
- clientData, 74, 95, 96, 243, 244, 293–295, 297, 330, 443, 472, 473, 525, 526, 530, 562, 563, 569
- conditionalPanel, 144, 145, 255, 352, 362, 363, 475
- fluidPage, 17, 23, 24, 26, 27, 29, 31, 34, 41, 51, 69, 76, 77, 80, 92, 94, 95, 99, 110–115, 117, 118, 120, 132, 133, 136, 163, 187–190, 266, 276, 285, 305, 306, 308, 309, 324, 347, 352, 359, 364, 366, 367, 377, 379, 381, 395, 434, 441, 446, 471, 491, 522, 546, 547, 558, 578, 579
- fluidrow, 25, 34, 35, 111, 112, 117–119, 121, 347, 352, 359–361, 366, 367, 442, 443, 446, 471
- HTML, 138, 139, 151, 152, 285, 306, 309, 457, 458
- includeText, 354, 458
- inputs
  - actionButton, 33, 34, 69, 73, 99, 155, 156, 158, 224–226, 229, 230, 233, 256–258, 281, 302, 353, 398, 400, 401, 404, 407, 409, 410, 412, 414, 416–419, 441–443, 491, 492, 522, 558
  - checkboxGroupInput, 34, 35, 145, 247, 353, 398–401, 404, 408–410, 412, 414, 416–421, 522, 558
  - checkboxInput, 34, 35, 56, 69, 84, 85, 87, 99, 113, 144, 145, 163, 353, 398, 400, 401, 404, 408–410, 412, 414, 416–419, 422, 423
  - dateInput, 34, 35, 353, 398, 400–404, 407–410, 412, 414, 416–419, 424, 425
  - dateRangeInput, 34, 35, 56, 69, 353, 398, 400, 401, 404–407, 409, 410, 412, 414, 416–419, 426, 427
  - numericInput, 34, 35, 78, 81, 145, 353, 383, 391, 398, 400, 401, 404, 408–410, 412, 414, 416–420, 422, 424, 426, 428–430, 432, 434, 437, 491, 510, 547, 579, 580
  - submitButton, 34, 230, 324, 353, 371, 398, 400, 401, 404, 408–410, 412, 414, 416–419
  - textInput, 34, 36, 69, 80, 81, 99, 311, 353, 398, 400, 401, 404, 408–410, 412, 414, 416–419, 437, 438, 526, 563
- invalidateLater, 355, 482, 499, 500
- isolate, 14, 73, 225, 226, 256–258, 355, 398, 466, 468, 482, 485, 486, 490, 491, 494, 499–503, 528, 536, 548, 565, 577, 581
- jQuery, 141, 146, 246, 312, 313, 315, 318, 320, 325–327
- logging, 355, 505
- MathJax, 352, 397
- mouse
  - clickId, 440, 441
  - hoverId, 440, 441
- navbarMenu, 116, 352, 374, 375
- navbarPage, 25, 109, 115–117, 120, 228–230, 311, 352, 371, 372, 374, 375, 385, 436
- outputOptions, 353, 445
- outputs
  - dataTableOutput, 99, 246, 247, 351, 354, 446, 447, 471
  - plotOutput, 17, 41, 52, 69, 76, 86, 87, 92, 94, 95, 110–112, 114, 121, 129, 132, 133, 136, 155, 156, 164, 187–190, 256, 277, 285, 295, 305–309, 324, 336, 337, 339–344, 347, 353, 360, 367, 373, 380, 381, 385, 387, 391, 393, 394, 440–442, 465, 510, 522, 538–545, 547, 558, 579, 580
  - verbatimTextOutput, 41, 78, 81, 114, 121, 129, 294, 295, 336, 337, 339–343, 347, 354, 385, 387, 442, 443, 449
- plotPNG, 356, 465, 473, 532, 536, 570, 576
- reactiveValues, 15, 226, 227, 229, 230, 254, 271, 355, 484, 486, 489, 493, 501–503, 525, 527, 562, 565
- renderDataTable, 99, 246–249, 319, 351, 354, 413, 432, 446, 447, 470, 471
- renderUI, 42, 144–146, 210, 331, 355, 397, 439, 475, 481
- RJSONIO, 317, 318, 351, 553, 555, 563
- selectInput, 34, 35, 41, 51, 77, 81, 84, 85, 87, 113, 144, 145, 163, 164, 209, 210, 235, 237, 245, 277, 285, 297, 324, 353, 362, 383, 391, 398, 400, 401, 404, 408–410, 412–414, 416–419, 432, 433
- session
  - argument, 96, 99, 238, 242, 243, 293, 294, 330, 420, 422, 424, 426, 428, 430, 433, 435–437, 442, 443, 452, 482, 495, 496, 498, 499, 504, 507, 526, 563
  - clientData, 74, 96, 243, 244, 293–295, 297, 330, 443, 526, 563
  - user, 169, 330
- tabPanel, 114–117, 128, 129, 229, 230, 247, 352, 371, 372, 374–377, 385, 387, 388
- tabsetPanel, 109, 113–115, 128, 129, 228–230, 247, 352, 353, 376, 385–388, 436
- updates
  - updateCheckboxGroupInput, 353, 400, 420, 421
  - updateCheckboxInput, 353, 401, 422
  - updateDateInput, 353, 404, 424
  - updateDateRangeInput, 353, 407, 426, 427
  - updateNumericInput, 353, 410, 420, 422, 424, 426, 428–430, 432, 434, 437
  - updateRadioButtons, 353, 412, 430, 431
  - updateSelectInput, 353, 414, 432, 433
  - updateTabsetPanel, 353, 388, 436
  - updateTextInput, 353, 418, 419, 437, 526, 563
- validate, 74, 82, 226, 260–264, 273, 276–286, 354–356, 393, 413, 415, 462, 482, 488–491, 493, 495, 497, 499, 500, 521, 522, 524, 557, 558, 560, 561

# 1 Tutorial

## 1.1 Teach yourself Shiny

# Teach yourself Shiny

You can teach yourself to use Shiny in two ways. You can watch the “How to Start Shiny” webinar series, or you can work through the self-paced Shiny tutorial below.

## Who should take the tutorial?

You will get the most out of the webinar or tutorial if you already know how to program in R, but not Shiny.

If R is new to you, you may want to check out the learning resources at [www.rstudio.com/training](http://www.rstudio.com/training) before taking this tutorial. If you are not sure whether you are ready for Shiny, try our [quiz](#).

If you use Shiny on a regular basis, you may want to skip this tutorial and visit the articles section of the Development Center. In the articles section, we cover individual Shiny topics at an advanced level.

## How to Start Shiny

The How to Start Shiny webinar series is recorded here in a single video. The video last two hours and 25 minutes, and will teach you how to build, deploy, and customize your own Shiny web apps.

[View individual chapters](#)

## The Shiny tutorial

This seven lesson tutorial will take you from R programmer to Shiny developer. Each lesson takes about 20 minutes and teaches one new Shiny skill. By the end of the lessons, you will know how to build and deploy a Shiny app.

Each lesson includes an exercise. Don't skip the exercises, even if you are tempted to get to the next lesson. The learning occurs in the exercises. How do we know? Because we designed the tutorial to be this way.

*Teach yourself Shiny: You can teach yourself to use Shiny in two ways. You can watch the "How to Start Shiny" webinar series, or you can*

Click the Lesson 1 button to get started and say hello to Shiny!

- [Lesson 1](#) - Welcome to Shiny
- [Lesson 2](#) - Layout the user interface
- [Lesson 3](#) - Add control widgets
- [Lesson 4](#) - Display reactive output
- [Lesson 5](#) - Use R scripts and data
- [Lesson 6](#) - Use reactive expressions
- [Lesson 7](#) - Share your apps

[Continue to lesson 1](#)

Shiny is an RStudio project. © 2014 RStudio, Inc.



Are you ready for Shiny?: You should know a little about R before you learn Shiny. Shiny is not a substitute for the R language, but a way to

## 1.2 Are you ready for Shiny?

# Are you ready for Shiny?

You should know a little about R before you learn Shiny. Shiny is not a substitute for the R language, but a way to extend R to a new domain, interactive web apps.

But how much R do you need to know?

This quiz will help you decide whether you know enough about R to feel confident with Shiny. You are ready to make the most of Shiny if you can answer each of the questions below.

## 1. Common Errors

`qplot()` is a function that comes in the `ggplot2` package in R. You can use `qplot()` to create quick scatterplots if you pass `qplot()` two variable names and the name of the data set that contains the variables.

The code below is a correctly written `qplot()` call; but if you copy and paste the code into R, you will get an error message when you run the code. Why?

```
qplot(Sepal.Width, Sepal.Length, data = iris)
```

Reveal answer

## 2. More Common Errors

You can assign values to R objects. For example, you could assign the age of your cat to an object named `cat` like this, `cat <- 4`.

Suppose you ran the code below and received the error message that follows. What does it mean?

```
cat + 1
## Error in cat + 1 : non-numeric argument to binary operator
```

Reveal answer

## 3. One More Common Error

The code below creates a function that returns a list. Assume that you run the code.

```
make_list <- function() {
  list(date = Sys.Date(),
        time = Sys.time(),
        timezone = Sys.timezone())
}

make_list()
```

Are you ready for Shiny?: You should know a little about R before you learn Shiny. Shiny is not a substitute for the R language, but a wa

```
## $date
## [1] "2015-03-12"
##
## $time
## [1] "2015-03-12 16:58:13 EDT"
##
## $timezone
## [1] "America/New_York"
```

You can call the function and immediately subset its result with R's dollar sign syntax. However, the code below will fail to do this. Why does the code fail, and how can you fix it?

```
make_list$time
## Error in make_list$time : object of type 'closure' is not subsettable
```

Reveal answer

## 4. Lists

The code below creates a list object.

```
lst <- list(numbers = 1:10, letters = letters, boolean = c(TRUE, FALSE))
lst
## $numbers
## [1] 1 2 3 4 5 6 7 8 9 10
## $letters
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
## [14] "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
## $boolean
## [1] TRUE FALSE
```

What will each of these return? What type of object will each be?

```
lst$numbers
lst[1]
lst[[1]]
```

Reveal answer

## 5. Data frames

Here is a data frame that comes with R. How can you calculate the sum of its `temperature` column?

```
pressure
##   temperature pressure
## 1           0  0.0002
## 2          20  0.0012
## 3          40  0.0060
## 4          60  0.0300
## 5          80  0.0900
## 6         100  0.2700
```

Are you ready for Shiny?: You should know a little about R before you learn Shiny. Shiny is not a substitute for the R language, but a wa

```
## /      120    0.7500
## 8      140    1.8500
## 9      160    4.2000
## 10     180    8.8000
## 11     200   17.3000
## 12     220   32.1000
## 13     240   57.0000
## 14     260   96.0000
## 15     280  157.0000
## 16     300  247.0000
## 17     320  376.0000
## 18     340  558.0000
## 19     360  806.0000
```

Reveal answer

## 6. Plots

How can you make a scatterplot of the `pressure` data? The plot should show temperature on the x axis and pressure on the y axis.

Reveal answer

## 7. Missing values

Suppose I change the first temperature value to `NA`, which stands for a missing value.

```
pressure$temperature[1] <- NA
```

What will `sum(pressure$temperature)` return? How can I ask `sum` to ignore the `NA`?

```
pressure
  temperature pressure
1           NA  0.0002
2           20  0.0012
3           40  0.0060
4           60  0.0300
5           80  0.0900
6          100  0.2700
7          120  0.7500
8          140  1.8500
9          160  4.2000
10         180  8.8000
11         200 17.3000
12         220 32.1000
13         240 57.0000
14         260 96.0000
15         280 157.0000
16         300 247.0000
17         320 376.0000
18         340 558.0000
19         360 806.0000
```

Reveal answer

Are you ready for Shiny?: You should know a little about R before you learn Shiny. Shiny is not a substitute for the R language, but a wa

## 8. Functions

Write a function that can take a vector of numbers as input, and return the mean of the numbers as output. Recall that the mean of a vector is the sum of the vector divided by the length of the vector.

Reveal answer

## 9. Scoping

What will this code return?

```
x <- 1
f <- function() {
  y <- 2
  c(x, y)
}
f()
```

Reveal answer

## 10. Assignment

What will the code below return?

```
obj <- 1
change_obj <- function(obj) {
  obj <- 2
}
change_obj(obj)
obj
```

Reveal answer

## 11. Packages

How would you install and load the `shiny` package so that you can use it in your R session? How often will you need to install the package? How often will you need to load it?

Reveal answer

## 12. Working directory

What is your working directory and how can you change it?

Reveal answer

## 13. Scripts

What is an R script? How can you “source” one, and what will that do?

Reveal answer

*Are you ready for Shiny?: You should know a little about R before you learn Shiny. Shiny is not a substitute for the R language, but a wa*

## Results

If you stumbled on these questions, you may find learning Shiny to be frustrating or confusing. But don't feel glum, R is easy to learn!

You can learn more about R by attending a [live training](#), reading a [book](#), or studying the [free online resources](#) at the RStudio [training](#) website.

If you answered all of the questions above, you're ready to go! A good way to learn Shiny is with our online tutorial.

Take the tutorial

Shiny is an [RStudio](#) project. © 2014 RStudio, Inc.

*The Shiny Webinar: Watch the complete webinar above, or jump to a specific chapter by clicking a link below. The entire webinar is two*

## 1.3 The Shiny Webinar

# The Shiny Webinar

Watch the complete webinar above, or jump to a specific chapter by clicking a link below. The entire webinar is two hours and 25 minutes long.

## Part 1 - How to build a Shiny app

1. Introduction
2. R
3. App architecture
4. App template
5. Inputs and outputs
6. The server function
7. Sharing apps
8. Shinyapps.io
9. Shiny servers
10. Recap - Part 1

## Part 2 - How to customize reactions

1. Introduction
2. Review of Part 1
3. Reactivity
4. Reactive values
5. Reactive functions
6. `render*()`
7. `reactive()`
8. `isolate()`
9. `observeEvent()`

*The Shiny Webinar: Watch the complete webinar above, or jump to a specific chapter by clicking a link below. The entire webinar is two*

10. [eventReactive\(\)](#)
11. [reactiveValues\(\)](#)
12. [Recap - Part 2](#)
13. [Parting tips](#)

## Part 3 - How to customize appearance

1. [Introduction](#)
2. [Review of Parts 1 and 2](#)
3. [HTML UI](#)
4. [Adding static content](#)
5. [Building layouts](#)
6. [Panels and tabsets](#)
7. [Prepackaged layouts](#)
8. [CSS](#)
9. [Recap - Part 3](#)

Shiny is an [RStudio](#) project. © 2014 RStudio, Inc.

## 1.4 LESSON 1: Welcome to Shiny

□ Lesson 1 2 3 4 5 6 7 □

### LESSON 1

# Welcome to Shiny

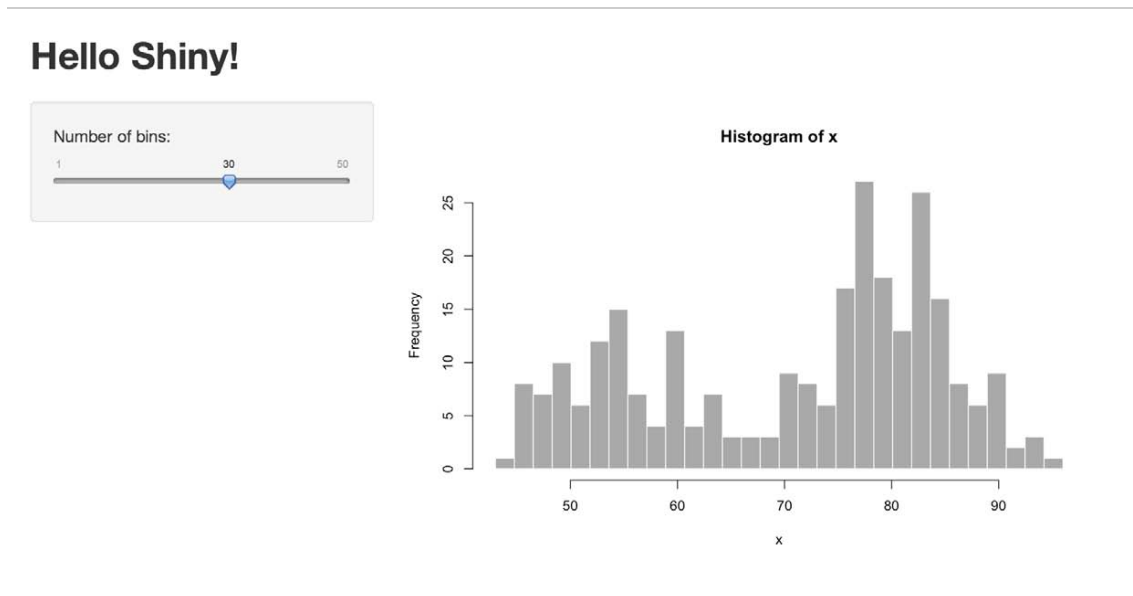
Shiny is an R package that makes it easy to build interactive web applications (apps) straight from R. This lesson will get you started building Shiny apps right away.

If you still haven't installed the Shiny package, open an R session, connect to the internet, and run

```
> install.packages("shiny")
```

This tutorial is based on the preview release of RStudio IDE. The preview release contains new features designed for Shiny. Download it [here](#).

## Examples



The Shiny package has [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:



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

## 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`. Here is the `ui.R` script for the Hello Shiny example.

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

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 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).

## 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")

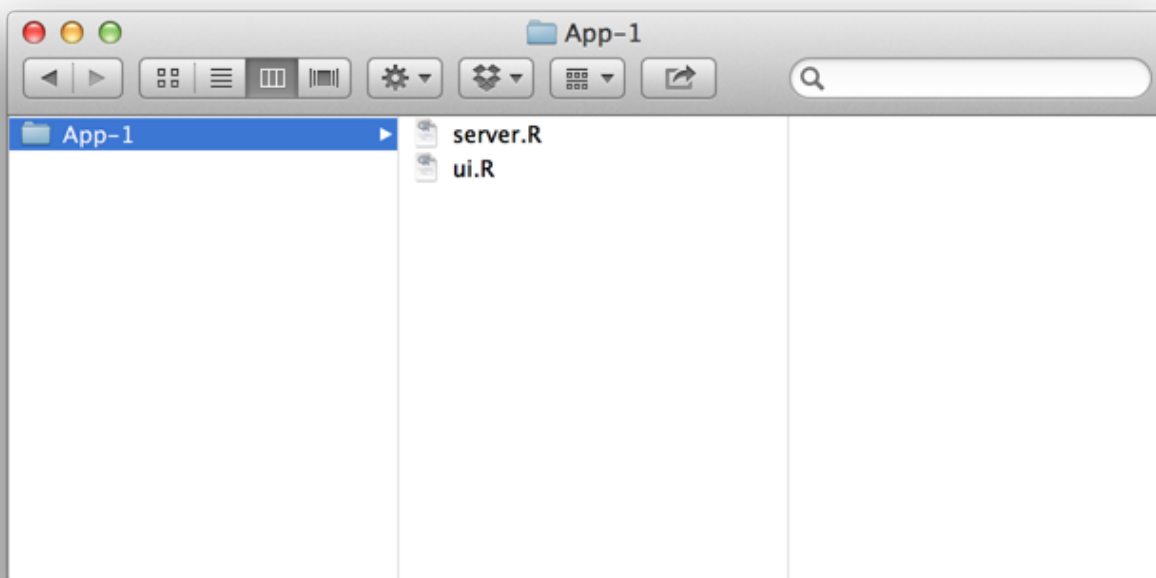
```

Note: `runApp` is similar to `read.csv`, `read.table`, and many other functions in R. 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 called `"01_hello"`. This directory is designed to work with the `runExample("01_hello")` call.)

## Your Turn

Create a new directory named `App-1` in your [working directory](#). Then copy and paste the `ui.R` and `server.R` scripts above into your directory (the scripts from Hello Shiny). When you are finished the directory should look like this:

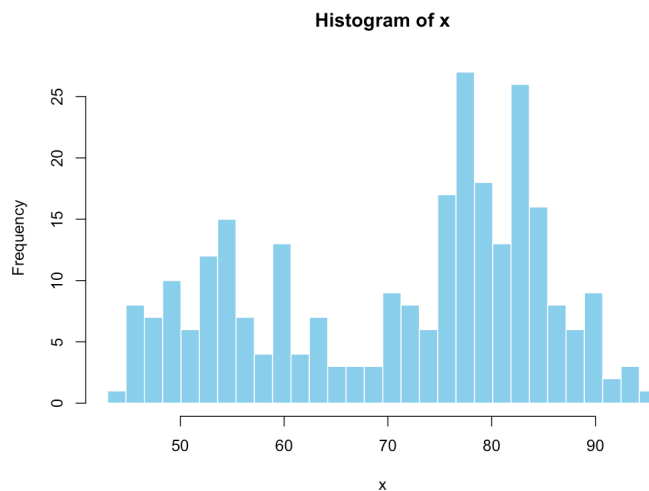
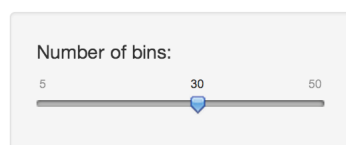


Launch your app by running `runApp("App-1")`. Then click escape and make some changes to your app:

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. If it doesn't, or if you want to check your code, press the model answers button to reveal how we did these tasks.

## Hello World!



By default, Shiny apps display in "normal" mode, like the app pictured above. Hello Shiny and the other built in examples display in "showcase mode", a different mode that displays the `server.R` and `ui.R` scripts alongside the app.

If you would like your app to display in showcase mode, you can run

```
runApp("App-1", display.mode = "showcase") .
```

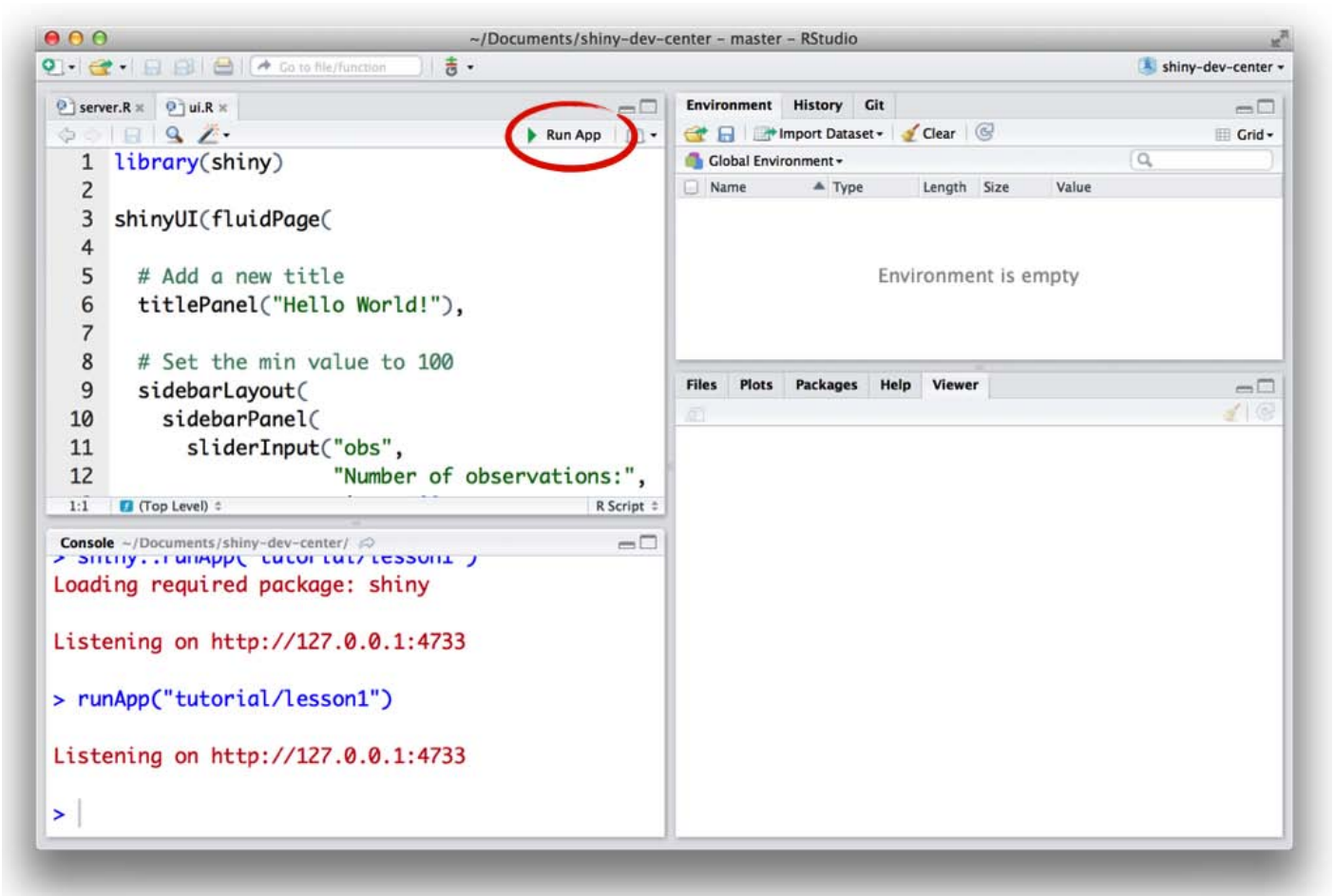
## Model Answers

Reveal answer

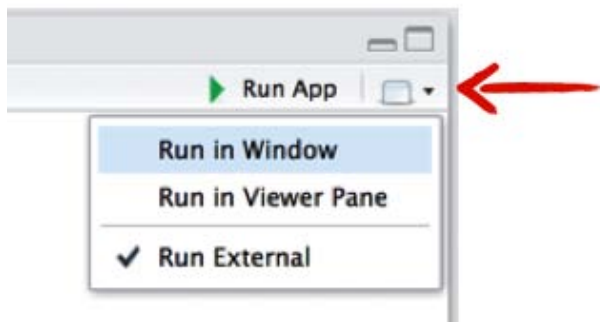
## Relaunching Apps

To relaunch your Shiny app:

- Run `runApp("App-1")` , or
- Open the `ui.R` or `server.R` scripts in your RStudio editor. RStudio will recognize the Shiny script and provide a Run App button (at the top of the editor). Either click this button to launch your app or use the keyboard shortcut: Command+Shift+Enter (Control+Shift+Enter on Windows).



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.



## Recap

To create your own Shiny app:

- Make a directory named for your app.
- Save your app's `server.R` and `ui.R` script inside that directory.
- Launch the app with `runApp` or RStudio's keyboard shortcuts.
- Exit the Shiny app by clicking escape.

## Go Further

You can create Shiny apps by copying and modifying existing Shiny apps. The [Shiny gallery](#) provides some good examples, or use the eleven pre-built Shiny examples listed below.

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

runExample("01_hello") # a histogram
runExample("02_text") # tables and data frames
runExample("03_reactivity") # a reactive expression
runExample("04_mpg") # global variables
runExample("05_sliders") # slider bars
runExample("06_tabsets") # tabbed panels
runExample("07_widgets") # help text and submit buttons
runExample("08_html") # Shiny app built from HTML
runExample("09_upload") # file upload wizard
runExample("10_download") # file download wizard
runExample("11_timer") # an automated timer
```

Each demonstrates a feature of Shiny apps. All Shiny example apps open in “showcase” mode (with the `ui.R` and `server.R` scripts in the display).

But why limit yourself to copying other apps? The next few lessons will show you how to build your own Shiny apps from scratch. You'll learn about each part of a Shiny app, and finish by deploying your own Shiny app online.

When you are ready, click to [Lesson 2](#), where you will learn how to build the layout and appearance of your Shiny apps.

[Continue to lesson 2](#)

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Eric Zivot

---

There is no run App button in the latest Rstudio for windows.

Garrett

---

The runApp button is a new feature in the next version of RStudio. You can download a preview of the version to use [here](#). The tutorial uses the preview version of RStudio. I apologize if I didn't make that clear enough at the start of Lesson 1.

Mona Jalal

---

how can I get rid of the unused area of the sidebarPanel in the left handside rather than using some nasty solution like this?

<http://stackoverflow.com/quest...>

Marcus

---

This initially stumped me as well, the version of the beta interface is 0.98.885 which provides the runApp button for server.R/ui.R scripts.

comments powered by Disqus

## 1.5 LESSON 2: Build a user-interface

□ 1 Lesson 2 3 4 5 6 7 □

### LESSON 2

# Build a user-interface

Now that you understand the structure of a Shiny app, it's time to build your first app from scratch.

This lesson will show you how to build a user-interface for your app. You will learn how to lay out the user-interface and then add text, images, and other HTML elements to your Shiny app.

We'll use the `App-1` app you made in [Lesson 1](#). To get started, open its `server.R` and `ui.R` files. 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, an appropriate starting point for this lesson.

## Layout

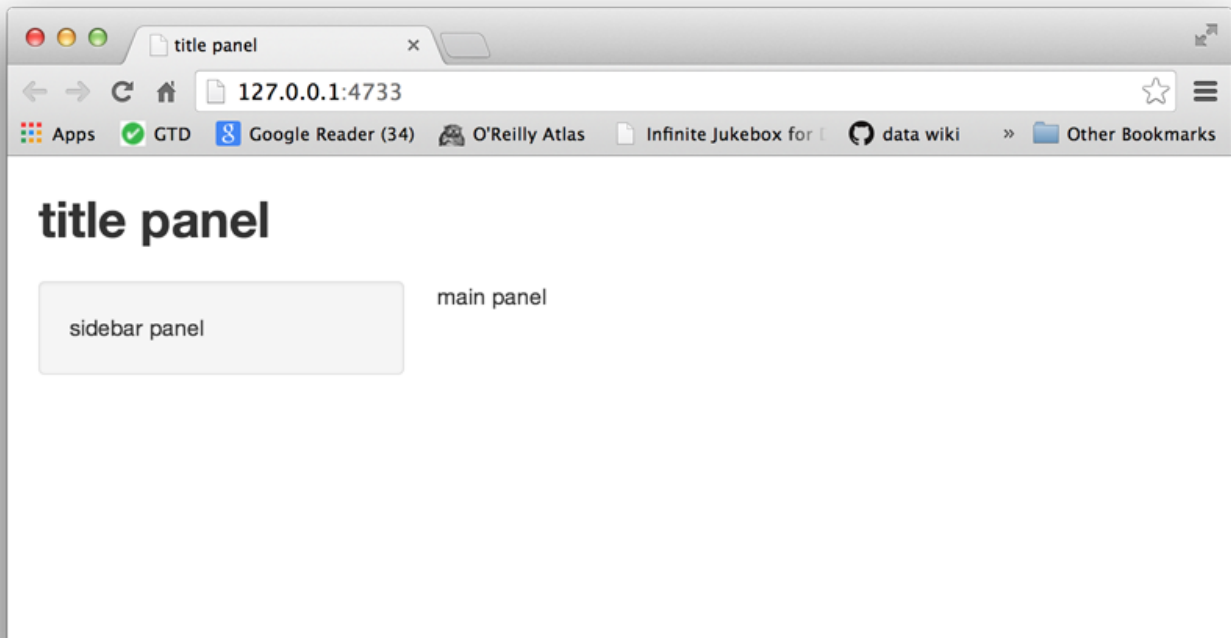
Shiny `ui.R` scripts use the function `fluidPage` to create a display that automatically adjusts to the dimensions of your user's browser window. You lay out your app by placing elements in the `fluidPage` function.

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")  
  )  
))
```

```
)
))
```



`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

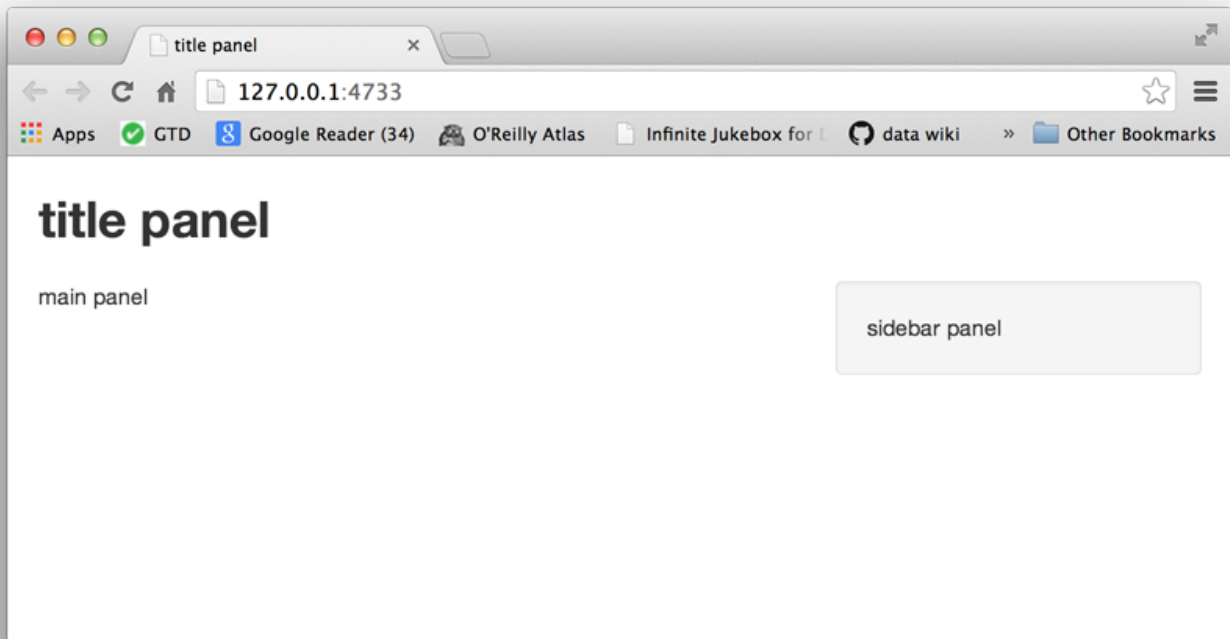
These functions place content in either the sidebar or the main panels. The sidebar panel will appear on the left side of your app by default. You can move it to the right side by giving `sidebarLayout` the optional argument `position = "right"`.

```
# ui.R
```

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

  sidebarLayout(position = "right",
    sidebarPanel("sidebar panel"),
    mainPanel("main panel")
  )
))
```





`titlePanel` and `sidebarLayout` create a basic layout for your Shiny app, but you can also create more advanced layouts. You can use `navbarPage` to give your app a multi-page user-interface that includes a navigation bar. Or you can use `fluidRow` and `column` to build your layout up from a grid system. If you'd like to learn more about these advanced options, read the [Shiny Application Layout Guide](#). We will stick with `sidebarLayout` in this tutorial.

## 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, e.g. `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. Let's try out a few of them.

shiny function HTML5 equivalent creates

<code>p</code>	<code>&lt;p&gt;</code>	A paragraph of text
<code>h1</code>	<code>&lt;h1&gt;</code>	A first level header
<code>h2</code>	<code>&lt;h2&gt;</code>	A second level header
<code>h3</code>	<code>&lt;h3&gt;</code>	A third level header
<code>h4</code>	<code>&lt;h4&gt;</code>	A fourth level header
<code>h5</code>	<code>&lt;h5&gt;</code>	A fifth level header
<code>h6</code>	<code>&lt;h6&gt;</code>	A sixth level header
<code>a</code>	<code>&lt;a&gt;</code>	A hyper link
<code>br</code>	<code>&lt;br&gt;</code>	A line break (e.g. a blank line)
<code>div</code>	<code>&lt;div&gt;</code>	A division of text with a uniform style
<code>span</code>	<code>&lt;span&gt;</code>	An in-line division of text with a uniform style
<code>pre</code>	<code>&lt;pre&gt;</code>	Text 'as is' in a fixed width font
<code>code</code>	<code>&lt;code&gt;</code>	A formatted block of code

<code>img</code>	<code>&lt;img&gt;</code>	An image
<code>strong</code>	<code>&lt;strong&gt;</code>	Bold text
<code>em</code>	<code>&lt;em&gt;</code>	Italicized text
HTML		Directly passes a character string as HTML code

## Headers

To create a header element:

- select a header function (e.g., `h1` or `h5`)
- 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.

```
> library(shiny)
> 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 script and then relaunch your app. Remember to relaunch a Shiny app you may run `runApp("App-1")`, click the Run App button, or use your keyboard shortcuts.

```
# 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.

You can create this effect with `align = "center"`, as in `h6("Episode IV", align = "center")`. In general, any HTML tag attribute can be set as an argument in any Shiny tag function.

If you are unfamiliar with HTML tag attributes, you can look them up in one of the many free online HTML resources such as [w3schools](http://w3schools.com).

Here's the code that made the Star Wars-inspired user-interface:

```
# ui.R

shinyUI(fluidPage(
  titlePanel("My Shiny App"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel(
      h6("Episode IV", align = "center"),
      h6("A NEW HOPE", align = "center"),
      h5("It is a period of civil war.", align = "center"),
      h4("Rebel spaceships, striking", align = "center"),
      h3("from a hidden base, have won", align = "center"),
      h2("their first victory against the", align = "center"),
      h1("evil Galactic Empire.")
    )
  )
))
```

## Formatted text

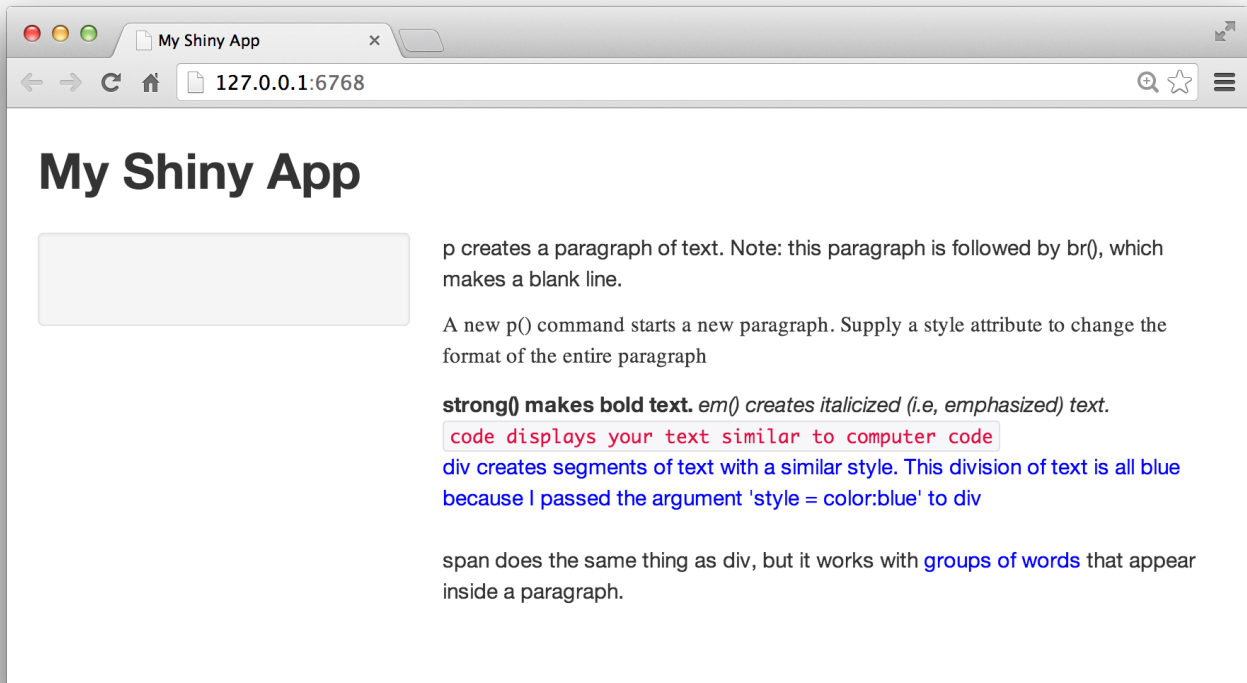
Shiny offers many tag functions for formatting text. The easiest way to describe them is by running through an example.

Paste the `ui.R` script below into your `ui.R` file and save it. If your Shiny app is still running, you can refresh your web page or preview window, and it will display the changes. If your app is closed, just relaunch it.

Compare the displayed app to your updated `ui.R` script to discover how to format text in a Shiny app.

```
# ui.R

shinyUI(fluidPage(
  titlePanel("My Shiny App"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel(
      p("p creates a paragraph of text."),
      p("A new p() command starts a new paragraph. Supply a style attribute to change the
format of the entire paragraph.", style = "font-family: 'times'; font-size: 16pt"),
      strong("strong() makes bold text."),
      em("em() creates italicized (i.e, emphasized) text."),
      br(),
      code("code displays your text similar to computer code"),
      div("div creates segments of text with a similar style. This division of text is all
blue because I passed the argument 'style = color:blue' to div", style = "color:blue"),
      br(),
      p("span does the same thing as div, but it works with",
      span("groups of words", style = "color:blue"),
      "that appear inside a paragraph.")
    )
  )
))
```



## Images

Images can enhance the appearance of your app and help your users understand the content. Shiny looks for the `img` function to place image files in your app.

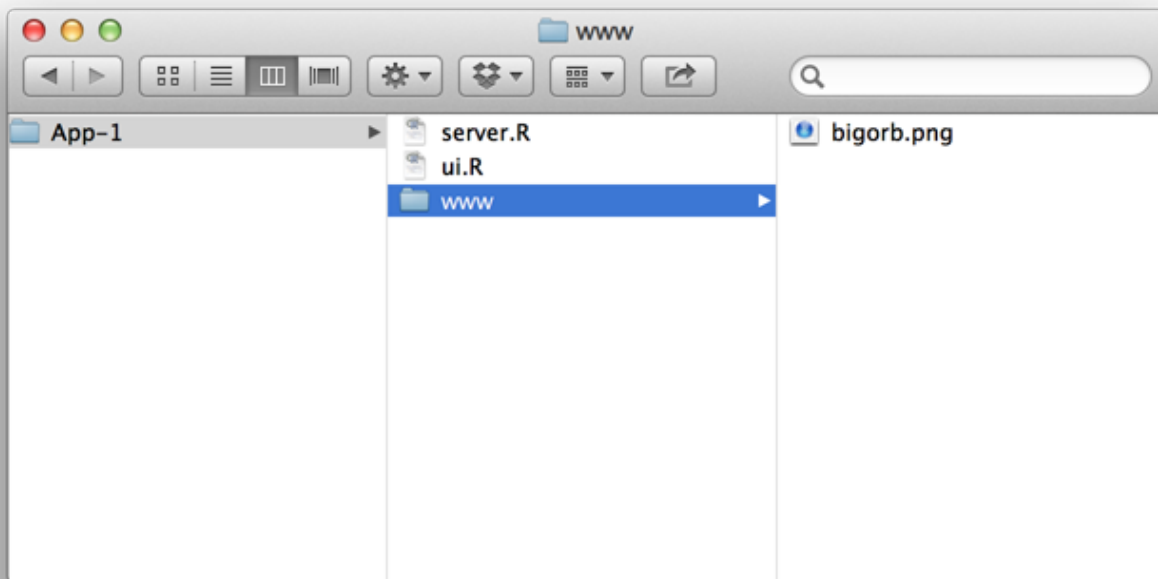
To insert an image, give the `img` function the name of your image file as the `src` argument (e.g., `img(src = "my_image.png")`). You must spell out this argument since `img` passes your input to an HTML tag, and `src` is what the tag expects.

You can also include other HTML friendly parameters such as `height` and `width`. Note that `height` and `width` numbers will refer to pixels.

```
img(src = "my_image.png", height = 72, width = 72)
```

The `img` function looks for your image file in a specific place. Your file *must* be in a folder named `www` in the same directory as the `ui.R` script. Shiny treats this directory in a special way. Shiny will share any file placed here with your user's web browser, which makes `www` a great place to put images, style sheets, and other things the browser will need to build the web components of your Shiny app.

So if you want to use an image named `bigorb.png`, your `App-1` directory should look like this one:



With this file arrangement, the `ui.R` script below can create this app. Download `bigorb.png` [here](#) and try it out.

```
# ui.R

shinyUI(fluidPage(
  titlePanel("My Shiny App"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel(
      img(src="bigorb.png", height = 400, width = 400)
    )
  )
))
```



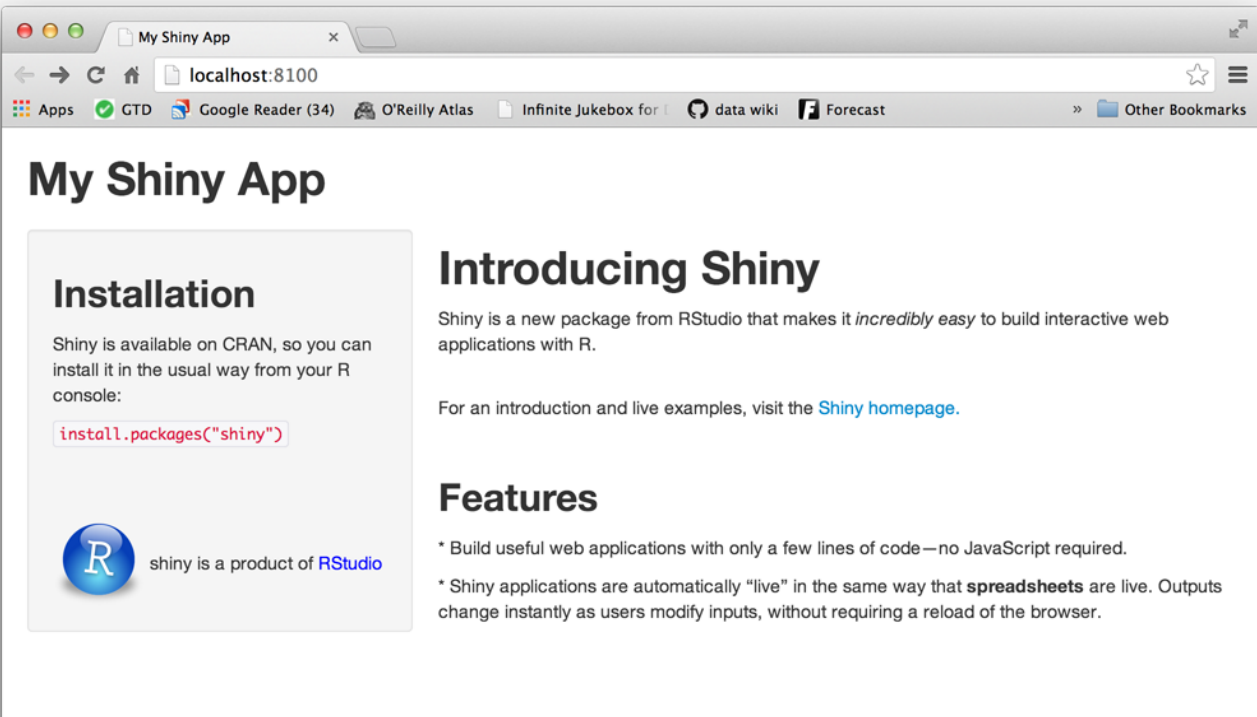
## Other tags

This lesson covers the most popular Shiny tag functions, but there are many more tag functions for you to use. You can learn about additional tag functions in [Customize your UI with HTML](#) and the [Shiny HTML Tags Glossary](#).

## Your turn

You can use Shiny's layout, HTML, and `img` functions to create very attractive and useful user-interfaces. See how well you understand these functions by recreating the Shiny app pictured below. Use the examples in this tutorial to work on it and then test it out.

Our `ui.R` script is found under the Model Answer button, but don't copy and paste it. Make sure you understand how the code works before moving on.



## Model Answer

Reveal answer

## Recap

With your new skills, you can:

- create a user-interface with `fluidPage`, `titlePanel` and `sidebarLayout`
- create an HTML element with one of Shiny's tag functions
- set HTML tag attributes in the arguments of each tag function
- add an element to your web page by passing it to `titlePanel`, `sidebarPanel` or `mainPanel`
- add multiple elements to each panel by separating them with a comma
- add images by placing your image in a folder labeled `www` within your Shiny app directory and then calling the `img` function

Now that you can place simple content in your user-interface, let's look at how you would place more complicated content, like widgets. Widgets are interactive web elements that your user can use to control the app. They are also the subject of [Lesson 3](#).

Continue to lesson 3

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Shobbo

---

I want to get an image to the right of the title.

```
shinyUI(pageWithSidebar(  
  headerPanel("EME Demo for Graphs (Standard!!)",  
    tags$head(  
      tags$img(src="PicLight1.png", height=100, width=100)  
    )  
  ),  
  )
```

It put the image above the title. Any way to put next to it.

THANKS

isomorphisms

---

Logically written. Thanks.

isomorphisms

---

comments powered by Disqus



## 1.6 LESSON 3: Add control widgets

□ 1 2 Lesson 3 4 5 6 7 □

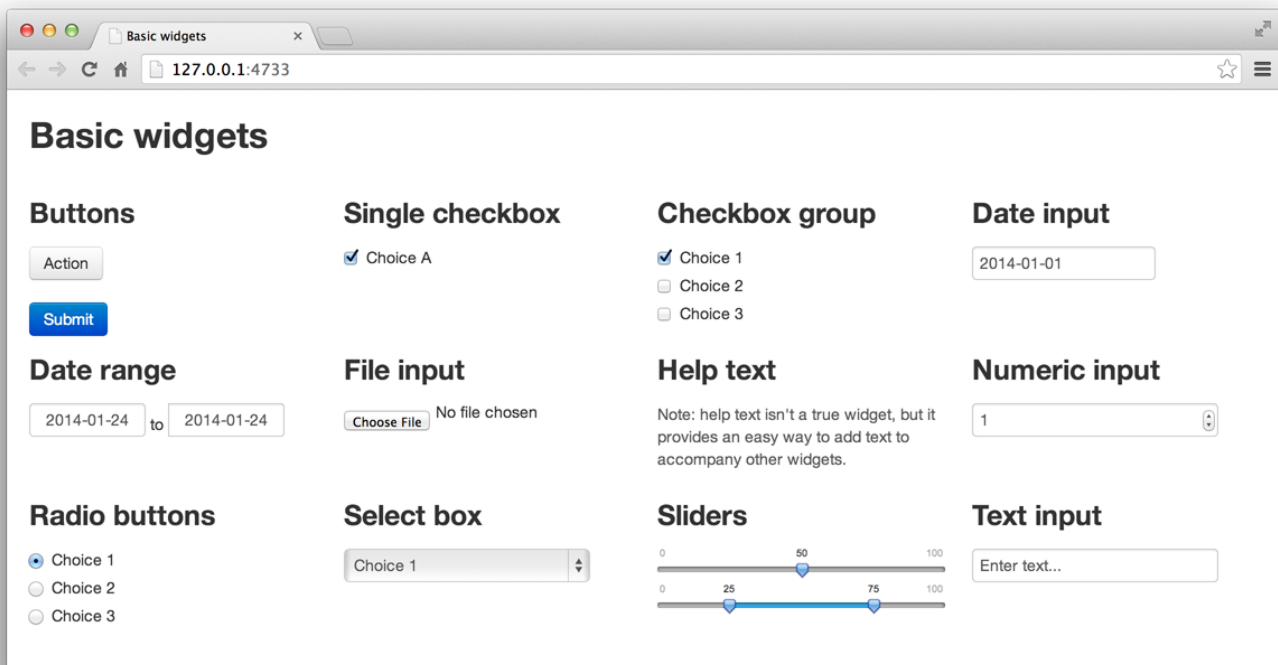
## LESSON 3

# Add control widgets

This lesson will show you how to add control widgets to your Shiny apps. What's a widget? A web element that your users can interact with. Widgets provide a way for your users to send messages to the Shiny app.

Shiny widgets collect a value from your user. When a user changes the widget, the value will change as well. This sets up opportunities that we'll explore in [Lesson 4](#).

## Control widgets



Shiny comes with a family of pre-built widgets, each created with a transparently named R function. For example, Shiny provides a function named `actionButton` that creates an Action Button and a function named `sliderInput` that creates a slider bar.

The standard Shiny widgets are:

function                      widget

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

Some of these widgets are built using the [Twitter Bootstrap](#) project, a popular open source framework for building user-interfaces.

## Adding widgets

You can add widgets to your web page in the same way that you added other types of HTML content in [Lesson 2](#). To add a widget to your app, place 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. They include things 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 `ui.R` script below makes the app pictured above. Change your own `App-1 ui.R` script to match it, and then launch the app (`runApp("App-1")`), select Run App, or use shortcuts).

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. If you are interested in the layout scheme for this Shiny app, read the description in the [application layout guide](#). This lesson will not cover this slightly more complicated layout scheme, but it is interesting to note what it does.

```
# 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))
  ),

```

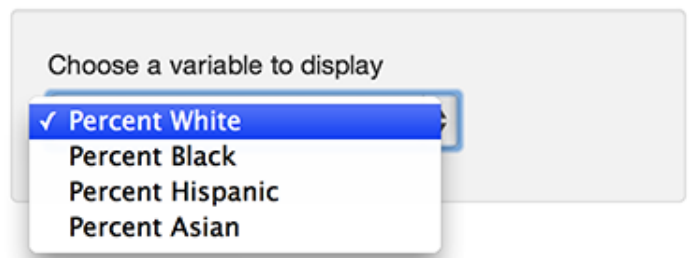
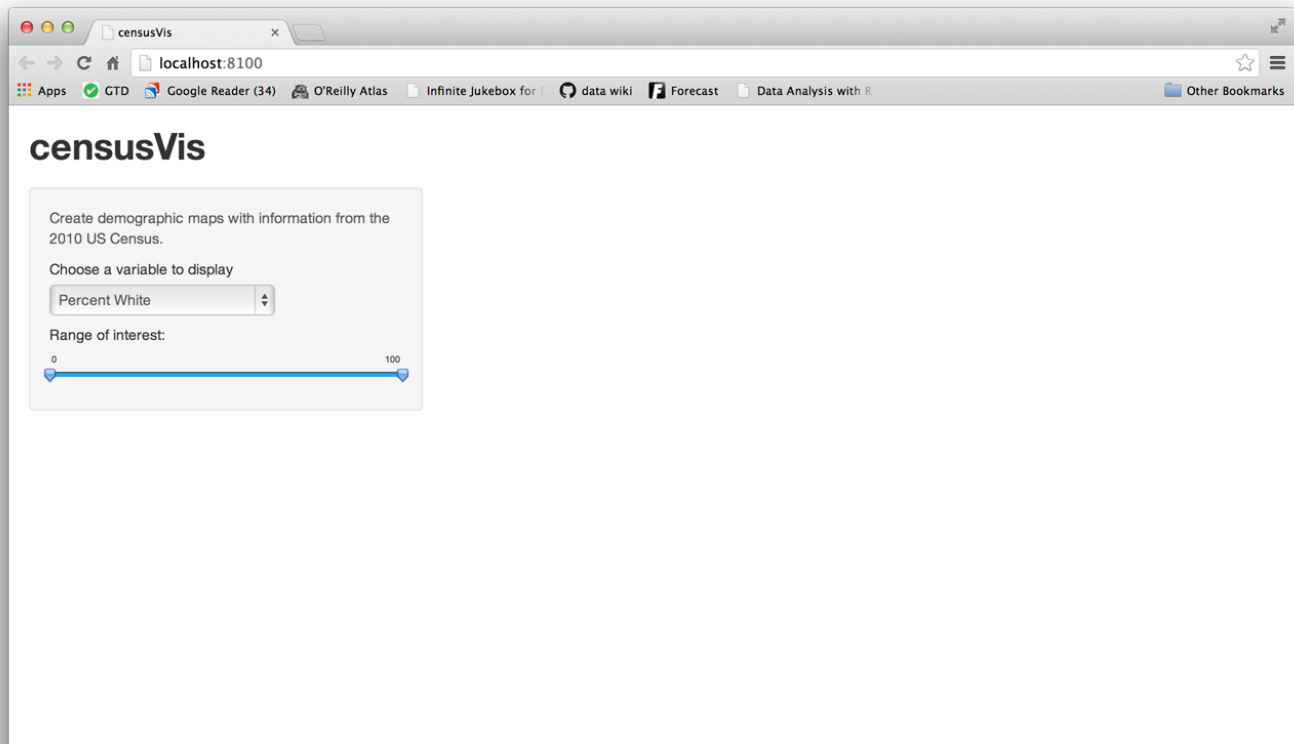
```

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

```

## Your turn

Rewrite your `ui.R` script to create the user-interface displayed below. Notice that this Shiny app uses a basic Shiny layout (no columns) and contains three of the widgets pictured above. The other values of the select box are shown below the image of the app.



## Model Answer

Reveal answer

## Recap

It is easy to add fully functional widgets to your Shiny app.

- Shiny provides a family of functions to create these widgets.
- Each function requires a name and a label.
- Some widgets need specific instructions to do their jobs.
- You add widgets to your Shiny app just like you added other types of HTML content (see [Lesson 2](#))

## Go Further

The [Shiny Widgets Gallery](#) provides templates that you can use to quickly add widgets to your Shiny apps.

To use a template, visit the [gallery](#). The gallery displays each of Shiny’s widgets, and demonstrates how the widgets’ values change in response to your input.

# Shiny Widgets Gallery

For each widget below, the Current Value(s) window displays the value that the widget provides to shinyServer. Notice that the values change as you interact with the widgets.

### Action button

Action

---

Current Value:

[1] 0

See Code

### Single checkbox

Choice A

---

Current Value:

[1] TRUE

See Code

### Checkbox group

Choice 1

Choice 2

Choice 3

---

Current Values:

[1] "1"

See Code

### Date input

2014-01-01

---

Current Value:

[1] "2014-01-01"

### Date range

2014-05-13

to

2014-05-13

---

Current Values:

[1] "2014-05-13" "2014-05-13"

### File input

Choose File no file selected

---

Current Value:

NULL

Select the widget that you want and click the “See Code” button below the widget. The gallery will take you to an example app that describes the widget. To use the widget, copy and paste the code in the example’s `ui.R` file to your `ui.R` file.

## Checkbox group

- Choice 1  
 Choice 2  
 Choice 3

[1] "1"

### Checkbox Group

checkboxGroupInput(inputId, label, choices, selected = NULL)

Creates a group of checkboxes. The widget sends the server a character vector that contains the values of the selected boxes.

**Arguments**

**inputId** The name to use to look up the value of the widget (as a character string)

**label** A label to display above the check boxes

**choices** A list of values. The widget will build a checkbox for each value of the list. If the list has names, these will be displayed next to the checkboxes. Otherwise the values themselves will be displayed.

**selected** The values that should initially be selected, if any.

*Make this widget by copying the code in ui.R.*

```
shinyUI(fluidPage(
  # Copy the chunk below to make a group of checkboxes
  checkboxGroupInput("checkboxGroup", label = h3("Checkbox group"),
    choices = list("Choice 1" = 1, "Choice 2" = 2, "Choice 3" = 3),
    selected = 1),
  hr(),
  fluidRow(column(3, verbatimTextOutput("value")))
))
```

[Show with app](#)

Code license: MIT

In [Lesson 4](#), you will learn how to connect widgets to reactive output, objects that update themselves whenever your user changes a widget.

Continue to lesson 4

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)  
[Chrome](#)  
[Internet Explorer 10+](#)  
[Safari](#)

-----  
 singco

i just like this tutorial!☺thanks

-----  
 Garrett

Thanks singco, I'm glad it is helpful!

-----  
 dazhi

catch you ^\_^

xiny Shen

---

I am a little confused about the checkbox widget, for example, if I have for check boxes for user, and each of them trigger their own SQL query in server. how can I do this? do I need any if-else statement? or do the checkbox widget provide any name for each of the box?

[comments powered by Disqus](#)

---

## 1.7 LESSON 4: Display reactive output

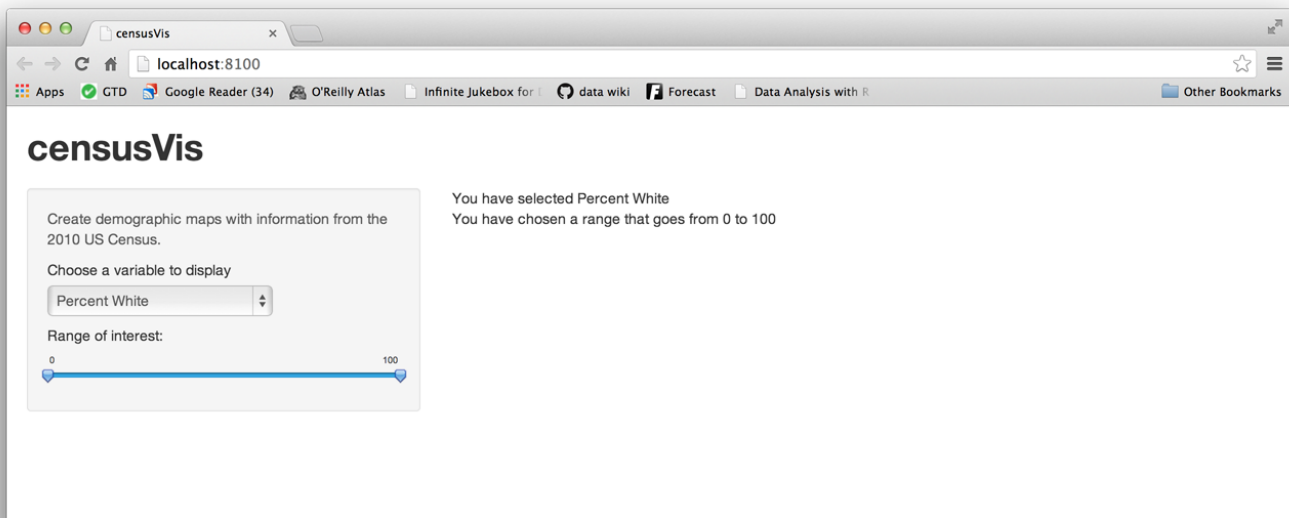
□ 1 2 3 Lesson 4 5 6 7 □

### LESSON 4

# Display reactive output

Time to give your Shiny app a “live” quality! This lesson will teach you how to build reactive output to display in your Shiny app. Reactive output automatically responds when your user toggles a widget.

By the end of this lesson, you’ll know how to make a simple Shiny app with two reactive lines of text. Each line will display the values of a widget based on your user’s input.



This new Shiny app will need its own, new directory. Create a folder in your working directory named `census-app`. This is where we’ll save the `ui.R` and `server.R` files that you make in this lesson.

## Two steps

You can create reactive output with a two step process.

1. Add an R object to your user-interface with `ui.R`.
2. Tell Shiny how to build the object in `server.R`. The object will be reactive if the code that builds it calls a widget value.

### Step 1: Add an R object to the UI



Shiny provides a family of functions that turn R objects into output for your user-interface. Each function creates a specific type of output.

Output function creates	
<code>htmlOutput</code>	raw HTML
<code>imageOutput</code>	image
<code>plotOutput</code>	plot
<code>tableOutput</code>	table
<code>textOutput</code>	text
<code>uiOutput</code>	raw HTML
<code>verbatimTextOutput</code>	text

You can add output to the user-interface in the same way that you added HTML elements and widgets. Place the output function inside `sidebarPanel` or `mainPanel` in the `ui.R` script.

For example, the `ui.R` file below uses `textOutput` to add a reactive line of text to the main panel of the Shiny app pictured above.

```
# ui.R

shinyUI(fluidPage(
  titlePanel("censusVis"),

  sidebarLayout(
    sidebarPanel(
      helpText("Create demographic maps with
        information from the 2010 US Census."),

      selectInput("var",
        label = "Choose a variable to display",
        choices = c("Percent White", "Percent Black",
          "Percent Hispanic", "Percent Asian"),
        selected = "Percent White"),

      sliderInput("range",
        label = "Range of interest:",
        min = 0, max = 100, value = c(0, 100))
    ),

    mainPanel(
      textOutput("text1")
    )
  )
))
```

Notice that `textOutput` takes an argument, the character string "text1". Each of the `*Output` functions require a single argument: a character string that Shiny will use as the name of your reactive element. Your users will not see this name, but you will use it later.

## Step 2: Provide R code to build the object.

Placing a function in `ui.R` tells Shiny where to display your object. Next, you need to tell Shiny how to build the object.

Do this by providing R code that builds the object in `server.R`. The code should go in the unnamed function that appears inside `shinyServer` in your `server.R` script.

The unnamed function plays a special role in the Shiny process; it builds a list-like object named `output` that contains all of the code needed to update the R objects in your app. Each R object needs to have its own entry in the list.

You can create an entry by defining a new element for `output` within the unnamed function, like below. The element name should match the name of the reactive element that you created in `ui.R`.

In the script below, `output$text1` matches `textOutput("text1")` in your `ui.R` script.

```
# server.R

shinyServer(function(input, output) {

  output$text1 <- renderText({
    "You have selected this"
  })

})
```

You do not need to arrange for the unnamed function to return `output` in its last line of code. R will automatically update `output` through reference class semantics.

Each entry to `output` should contain the output of one of Shiny's `render*` functions. These functions capture an R expression and do some light pre-processing on the expression. Use the `render*` function that corresponds to the type of reactive object you are making.

render function creates

<code>renderImage</code>	images (saved as a link to a source file)
<code>renderPlot</code>	plots
<code>renderPrint</code>	any printed output
<code>renderTable</code>	data frame, matrix, other table like structures
<code>renderText</code>	character strings
<code>renderUI</code>	a Shiny tag object or HTML

Each `render*` function takes a single argument: an R expression surrounded by braces, `{}`. The expression can be one simple line of text, or it can involve many lines of code, as if it were a complicated function call.

Think of this R expression as a set of instructions that you give Shiny to store for later. Shiny will run the instructions when you first launch your app, and then Shiny will re-run the instructions every time it needs to update your object.

For this to work, your expression should return the object you have in mind (a piece of text, a plot, a data frame, etc). You will get an error if the expression does not return an object, or if it returns the wrong type of object.

## Use widget values

If you run the `server.R` script above, the Shiny app will display "You have selected this" in the main panel. However, the text will not be reactive. It will not change even if you manipulate the widgets of your app.

You can make the text reactive by asking Shiny to call a widget value when it builds the text. Let's look at how to do this.

Take a look at the first line of code in `server.R`. Do you notice that the unnamed function mentions *two* arguments, `input` and `output`? You already saw that `output` is a list-like object that stores instructions for building the R objects in your app.

`input` is a second list-like object. It stores the current values of all of the widgets in your app. These values will be saved under the names that you gave the widgets in `ui.R`.

So for example, our app has two widgets, one named "var" and one named "range" (you gave the widgets these names in [Lesson 3](#)). The values of "var" and "range" will be saved in `input` as `input$var` and `input$range`. Since the slider widget has two values (a min and a max), `input$range` will contain a vector of length two.

Shiny will automatically make an object reactive if the object uses an `input` value. For example, the `server.R` file below creates a reactive line of text by calling the value of the select box widget to build the text.

```
# server.R

shinyServer(
  function(input, output) {

    output$text1 <- renderText({
      paste("You have selected", input$var)
    })

  }
)
```

Shiny tracks which outputs depend on which widgets. When a user changes a widget, Shiny will rebuild all of the outputs that depend on the widget, using the new value of the widget as it goes. As a result, the rebuilt objects will be completely up-to-date.

This is how you create reactivity with Shiny, by connecting the values of `input` to the objects in `output`. Shiny takes care of all of the other details.

## Launch your app and see the reactive output

When you are ready, update your `server.R` and `ui.R` files to match those above. Then launch your Shiny app by running `runApp("censusVis", display.mode = "showcase")` at the command line. Your app should look like the app below, and your statement should update instantly as you change the select box widget.

Watch the `server.R` script. When Shiny rebuilds an output, it highlights the code it is running. This temporary highlighting can help you see how Shiny generates reactive output.

```
# server.R

shinyServer(
  function(input, output) {

    output$text1 <- renderText({
      paste("You have selected", input$var)
    })

    output$text2 <- renderText({
      paste("You have chosen a range that goes from",
            input$range[1], "to", input$range[2])
    })

  }
)
```

## Your turn

Add a second line of reactive text to the main panel of your Shiny app. This line should display “You have chosen a range that goes from *something* to *something*”, and each *something* should show the current minimum (min) or maximum (max) value of the slider widget.

Don't forget to update both your `ui.R` and `server.R` files.

## Model answer

Reveal answer

## Recap

In this lesson, you created your first reactive Shiny app. Along the way, you learned to

- use an `*Output` function in the `ui.R` script to place reactive objects in your Shiny app
- use a `render*` function in the `server.R` script to tell Shiny how to build your objects
- surround R expressions by braces, `{}`, in each `render*` function
- save your `render*` expressions in the `output` list, with one entry for each reactive object in your app.
- create reactivity by including an `input` value in a `render*` expression

If you follow these rules, Shiny will automatically make your objects reactive.

In [Lesson 5](#) you will create a more sophisticated reactive app that relies on R scripts and external data.

Continue to lesson 5

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Jean

---

Thanks for these tutorials. Very helpful.

There's an error in this lesson. The folder name "census-app" and the argument to `runApp()`, "censusVis" should be the same. One or the other should be changed to match the other.

Howard

---

Hi. Thanks for a great read.

*LESSON 4: Display reactive output*

How can I pass a reactive string variable from server.R to ui.R that would be evaluated "as-is" in ui.R?

For example, I would like to evaluate the "horizontalTableWidth" variable in the ui.R:  
`div(tableOutput("table"), style = horizontalTableWidth)`

I cant get no.. loading..table

---

comments powered by [Disqus](#)

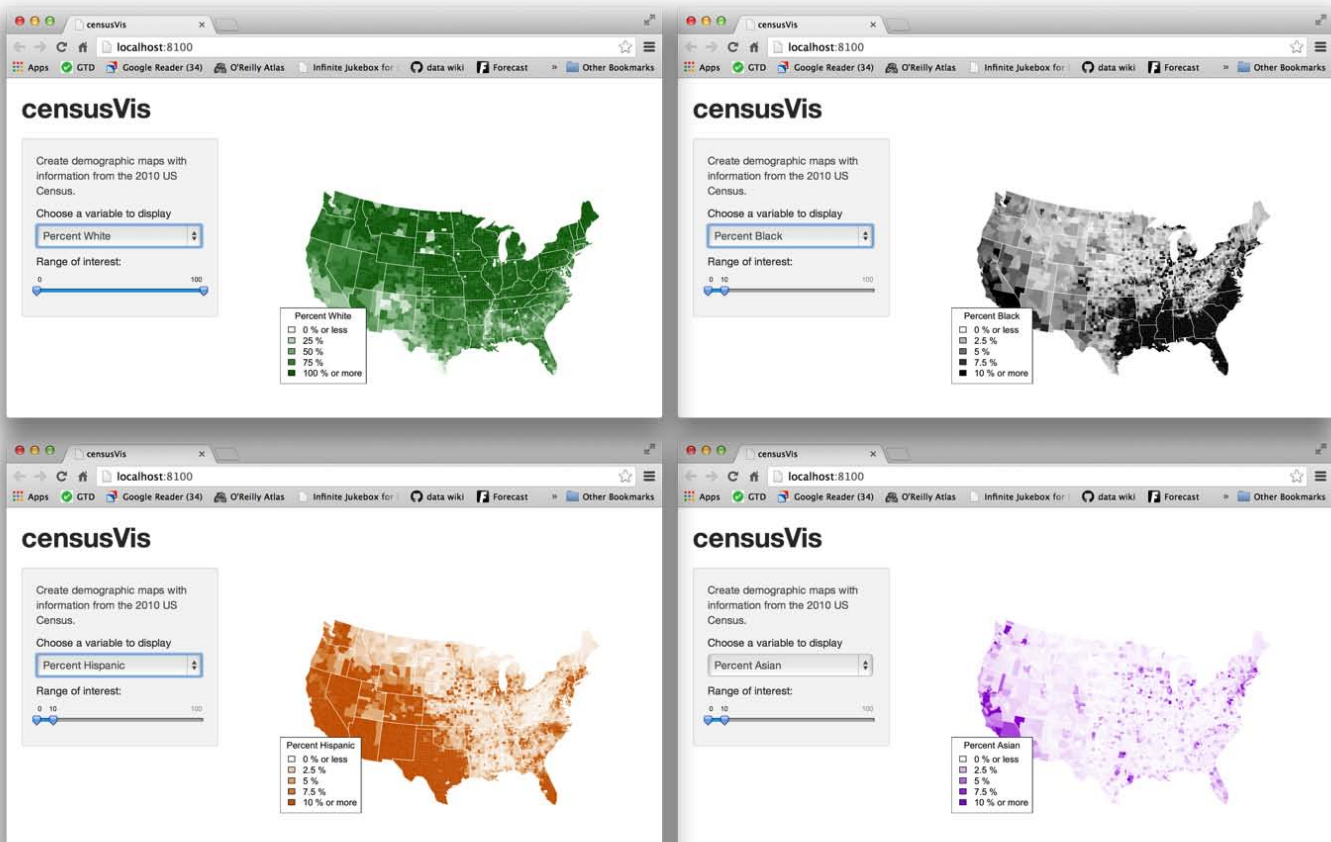
## 1.8 LESSON 5: Use R scripts and data

□ 1 2 3 4 Lesson 5 6 7 □

## LESSON 5

# Use R scripts and data

This lesson will show you how to load data, R Scripts, and packages to use in your Shiny apps. Along the way, you will build a sophisticated app that visualizes US Census data.



## counties.rds

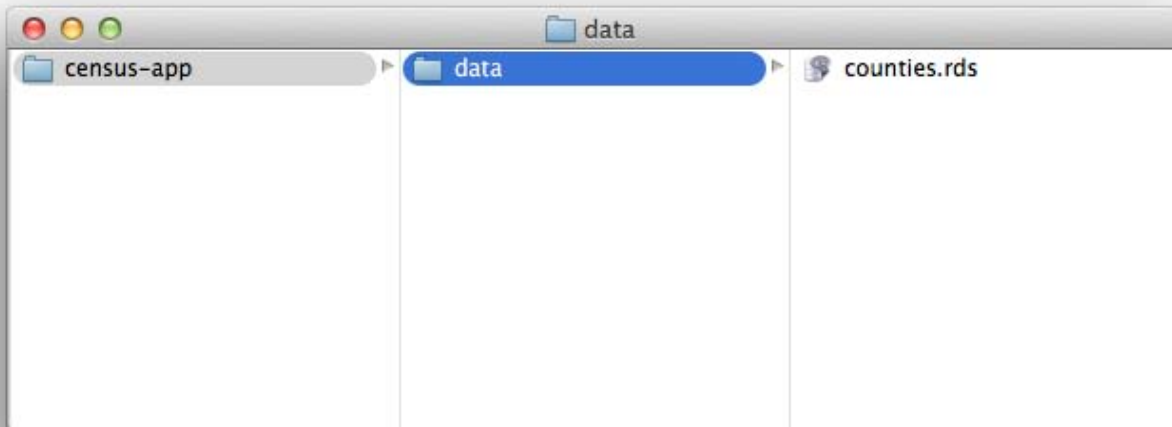
`counties.rds` is a dataset of demographic data for each county in the United States, collected with the `UScensus2010` R package. You can download it [here](#).

Once you have the file,

- Create a new folder named `data` in your `census-app` directory.

- Move `counties.rds` into the `data` folder.

When you're done, your `census-app` folder should look like this.



The dataset in `counties.rds` contains

- the name of each county in the United States
- the total population of the county
- the percent of residents in the county who are white, black, hispanic, or asian

```
counties <- readRDS("census-app/data/counties.rds")
head(counties)
```

	name	total.pop	white	black	hispanic	asian
1	alabama, autauga	54571	77.2	19.3	2.4	0.9
2	alabama, baldwin	182265	83.5	10.9	4.4	0.7
3	alabama, barbour	27457	46.8	47.8	5.1	0.4
4	alabama, bibb	22915	75.0	22.9	1.8	0.1
5	alabama, blount	57322	88.9	2.5	8.1	0.2
6	alabama, bullock	10914	21.9	71.0	7.1	0.2

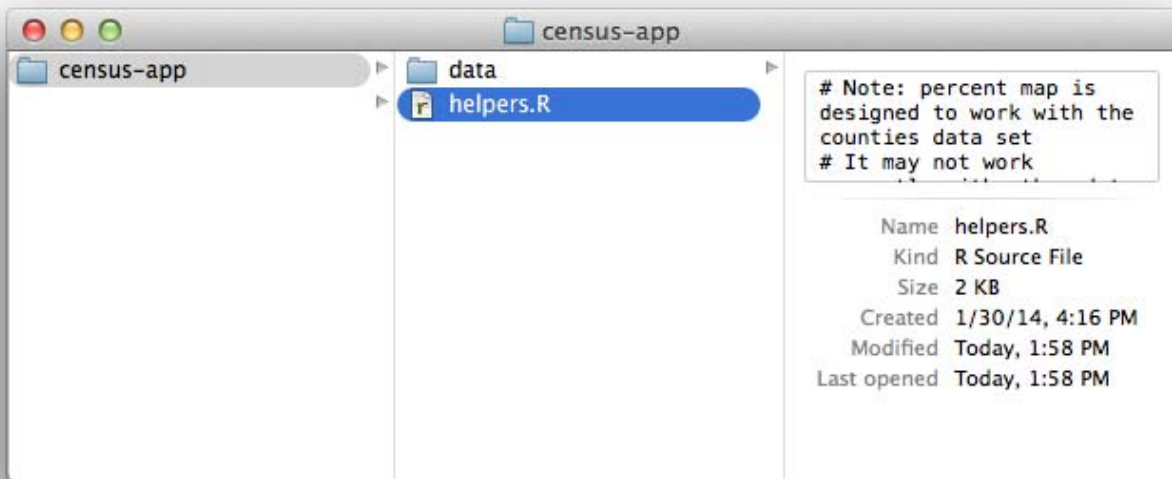
## helpers.R

`helpers.R` is an R script that can help you make [choropleth maps](#), like the ones pictured above. A choropleth map is a map that uses color to display the regional variation of a variable. In our case, `helpers.R` will create `percent_map`, a function designed to map the data in `counties.rds`. You can download `helpers.R` [here](#).

`helpers.R` uses the `maps` and `mapproj` packages in R. If you've never installed these packages before, you'll need to do so before you make this app. Run

```
> install.packages(c("maps", "mapproj"))
```

Save `helpers.R` inside your `census-app` directory, like below.



The `percent_map` function in `helpers.R` takes five arguments:

Argument	Input
<code>var</code>	a column vector from the <code>counties.rds</code> dataset
<code>color</code>	any character string you see in the output of <code>colors()</code>
<code>legend.title</code>	A character string to use as the title of the plot's legend
<code>max</code>	A parameter for controlling shade range (defaults to 100)
<code>min</code>	A parameter for controlling shade range (defaults to 0)

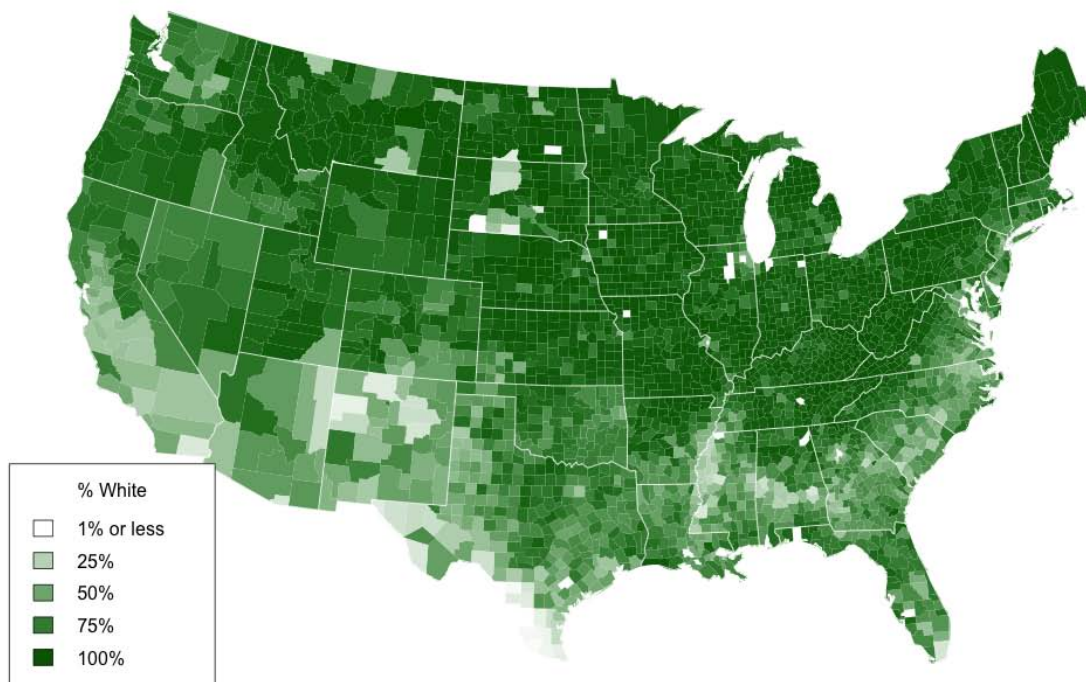
You can use `percent_map` at the command line to plot the counties data as a choropleth map, like this.

```
library(maps)
library(mapproj)
source("census-app/helpers.R")
counties <- readRDS("census-app/data/counties.rds")
percent_map(counties$white, "darkgreen", "% white")
```

*Note: The code above assumes that `census-app` is a sub-directory in your working directory. Make certain to set your working directory as the parent directory for `census-app`. To change your working directory location, click on `Session > Set Working Directory > Choose Directory...` in the RStudio menu bar.*

`percent_map` plots the counties data as a choropleth map. Here it will plot the percent of white residents in the counties in the color dark green.





## Loading files and file paths

Take a look at the above code. To use `percent_map`, we first ran `helpers.R` with the `source` function, and then loaded `counties.rds` with the `readRDS` function. We also ran `library(maps)` and `library(mapproj)`.

You will need to ask Shiny to call the same functions before it uses `percent_map` in your app, but how you write these functions will change. Both `source` and `readRDS` require a file path, and file paths do not behave the same way in a Shiny app as they do at the command line.

When Shiny runs the commands in `server.R`, it will treat all file paths as if they begin in the same directory as `server.R`. In other words, the directory that you save `server.R` in will become the working directory of your Shiny app.

Since you saved `helpers.R` in the same directory as `server.R`, you can ask Shiny to load it with

```
source("helpers.R")
```

Since you saved `counties.rds` in a sub-directory (named `data`) of the directory that `server.R` is in, you can load it with.

```
counties <- readRDS("data/counties.rds")
```

You can load the `maps` and `mapproj` packages in the normal way with

```
library(maps)  
library(mapproj)
```

which does not require a file path.

# Execution

Shiny will execute all of these commands if you place them in your `server.R` script. However, where you place them in `server.R` will determine how many times they are run (or re-run), which will in turn affect the performance of your app.

Shiny will run some sections of `server.R` more often than others.

Shiny will run the whole script the first time you call `runApp`. This causes Shiny to execute `shinyServer`. `shinyServer` then gives Shiny the unnamed function in its first argument.

```
# server.R

# A place to put code

shinyServer(
  function(input, output) {
    # Another place to put code

    output$map <- renderPlot({
      # A third place to put code
    })
  }
)
```

Run once when app is launched

Shiny saves the unnamed function until a new user arrives. Each time a new user visits your app, Shiny runs the unnamed function again, one time. The function helps Shiny build a distinct set of reactive objects for each user.

Here's what we've learned so far:

- The `server.R` script is run once, when you launch your app
- The unnamed function inside `shinyServer` is run once *each time* a user visits your app
- The R expressions inside `render*` functions are run many times. Shiny runs them once each time a user changes a widget.

How can you use this information?

Source scripts, load libraries, and read data sets at the beginning of `server.R` *outside* of the `shinyServer` function. Shiny will only run this code once, which is all you need to set your server up to run the R expressions contained in `shinyServer`.

Define user specific objects inside `shinyServer`'s unnamed function, but outside of any `render*` calls. These would be objects that you think each user will need their own personal copy of. For example, an object that records the user's *session information*. This code will be run once per user.

Only place code that Shiny *must* rerun to build an object inside of a `render*` function. Shiny will rerun *all* of the code in a `render*` chunk each time a user changes a widget mentioned in the chunk. This can be quite often.

You should generally avoid placing code inside a `render` function that does not need to be there. The code will slow down the entire app.

## Your Turn 1

Copy and paste the following `ui.R` and `server.R` files to your `census-app` directory. Then add

```
source("helpers.R")
counties <- readRDS("data/counties.rds")
library(maps)
library(mapproj)
```

to `server.R`. Be sure to place the commands in an efficient location.

*Note: This is the first of two steps that will complete your app. Choose the best place to insert the code above, but do not try to run the app. Your app will return an error until you replace # some arguments with real code in Your Turn 2.*

`ui.R`

```
# ui.R

shinyUI(fluidPage(
  titlePanel("censusVis"),

  sidebarLayout(
    sidebarPanel(
      helpText("Create demographic maps with
        information from the 2010 US Census."),

      selectInput("var",
        label = "Choose a variable to display",
        choices = c("Percent White", "Percent Black",
          "Percent Hispanic", "Percent Asian"),
        selected = "Percent White"),

      sliderInput("range",
```

```

    label = "range of interest:",
    min = 0, max = 100, value = c(0, 100))
  ),

  mainPanel (plotOutput("map"))
)
))

```

server.R

```

# server.R

shinyServer(
  function(input, output) {

    output$map <- renderPlot({

      percent_map( # some arguments )
    })

  }
)

```

## Model Answer 1

Reveal answer

## Finishing the app

The censusVis app has one reactive object, a plot named “map”. The plot is built with the `percent_map` function, which takes five arguments.

- The first three arguments, `var`, `color`, and `legend.title`, depend on the value of the select box widget.
- The last two arguments, `max` and `min`, should be the max and min values of the slider bar widget.

The `server.R` script below shows one way to craft reactive arguments for `percent_map`. R’s `switch` function can transform the output of a select box widget to whatever you like. However, the script is incomplete. It does not provide values for `color`, `legend.title`, `max`, or `min`. Note: the script will not run as is. You will need to finish the script before you run it, which is the task of Your Turn 2.

```

# server.R

library(maps)
library(mapproj)
counties <- readRDS("data/counties.rds")
source("helpers.R")

shinyServer(
  function(input, output) {
    output$map <- renderPlot({
      data <- switch(input$var,
        "Percent White" = counties$white,
        "Percent Black" = counties$black,
        "Percent Hispanic" = counties$hispanic,
        "Percent Asian" = counties$asian)
    })
  }
)

```

```

percent_map(var = data, color = ?, legend.title = ?, max = ?, min = ?)
  })
}
)

```

## Your Turn 2

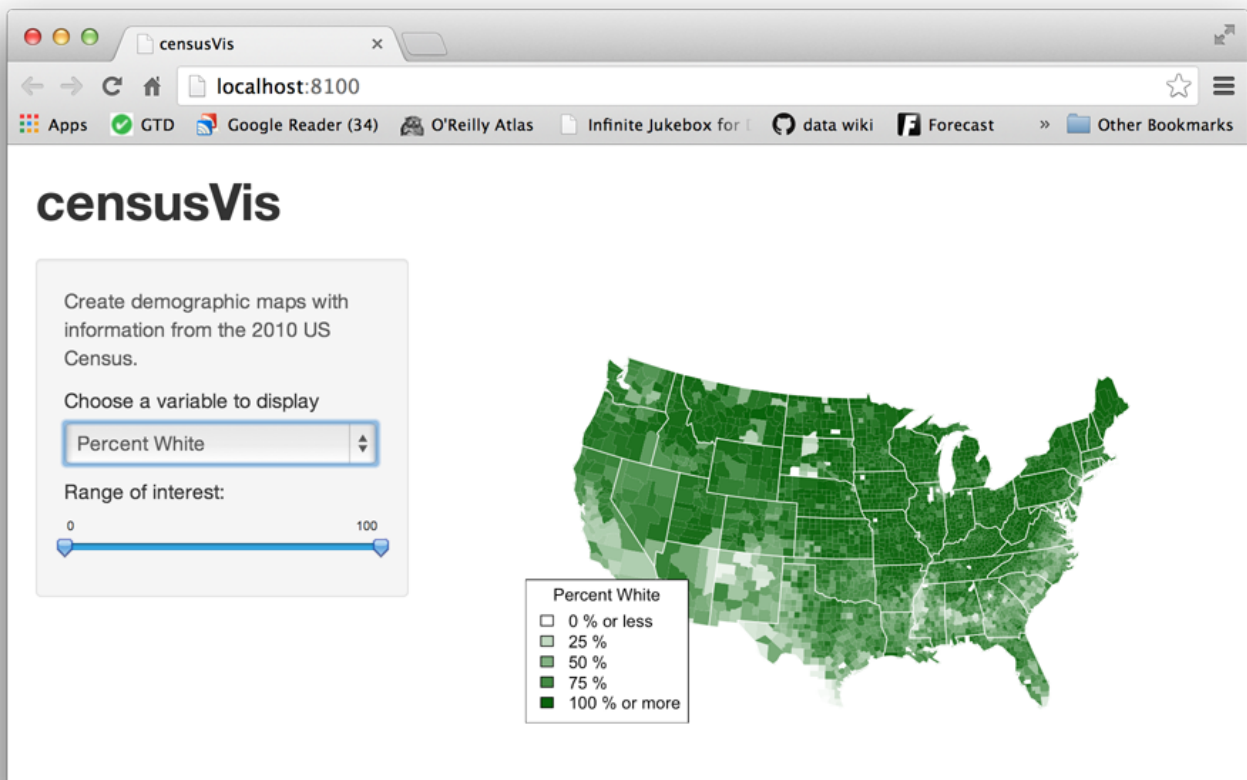
Complete the code to build a working censusVis app.

When you're ready to deploy your app, save your `server.R` and `ui.R` files and run `runApp("census-app")`. If everything works, your app should look like the picture below.

You'll need to decide

- how to create the argument values for `percent_map`, and
- where to put the code that creates these arguments.

Remember, you'll want the argument values to switch whenever a user changes the associated widget. When you are finished, or if you get stuck, read on below for a model answer.



## Model Answers 2

Reveal answer

## Recap

You can create more complicated Shiny apps by loading R Scripts, packages, and data sets.

Keep in mind:

- The directory that `server.R` appears in will become the working directory of the Shiny app
- Shiny will run code placed at the start of `server.R`, before `shinyServer`, only once during the life of the app.
- Shiny will run code placed inside `shinyServer` multiple times, which can slow the app down.

You also learned that `switch` is a useful companion to multiple choice Shiny widgets. Use `switch` to change the values of a widget into R expressions.

As your apps become more complex, they can become inefficient and slow. [Lesson 6](#) will show you how to build fast, modular apps with reactive expressions.

[Continue to lesson 6](#)

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Fernando Hernandez

---

The code for the legend for Asian in

Model Answers 2

A simple version of server.R:

```
says "Percent Asian" = "% Violet")
```

Awesome tutorial by the way! I particularly like the Your Turn sections.

Garrett

---

Thanks for catching that, Fernando. I'm glad you like the tutorial!

Ken Deal

---

I have a problem similar to one below by Mira and TyStudio but I've not found a solution.

Is there a trick to reading data into the server.r file?

comments powered by [Disqus](#)

## 1.9 LESSON 6: Use reactive expressions

□ 1 2 3 4 5 Lesson 6 7 □

### LESSON 6

# Use reactive expressions

Shiny apps wow your users by running fast, instantly fast. But what if your app needs to do a lot of slow computation?

This lesson will show you how to streamline your Shiny apps with reactive expressions. Reactive expressions let you control which parts of your app update when, which prevents unnecessary work.

To get started:

- Create a new folder named `stockVis` in your working directory.
- Download the following files and place them inside `stockVis`: `ui.R`, `server.R`, and `helpers.R`.
- Launch the app with `runApp("stockVis")`

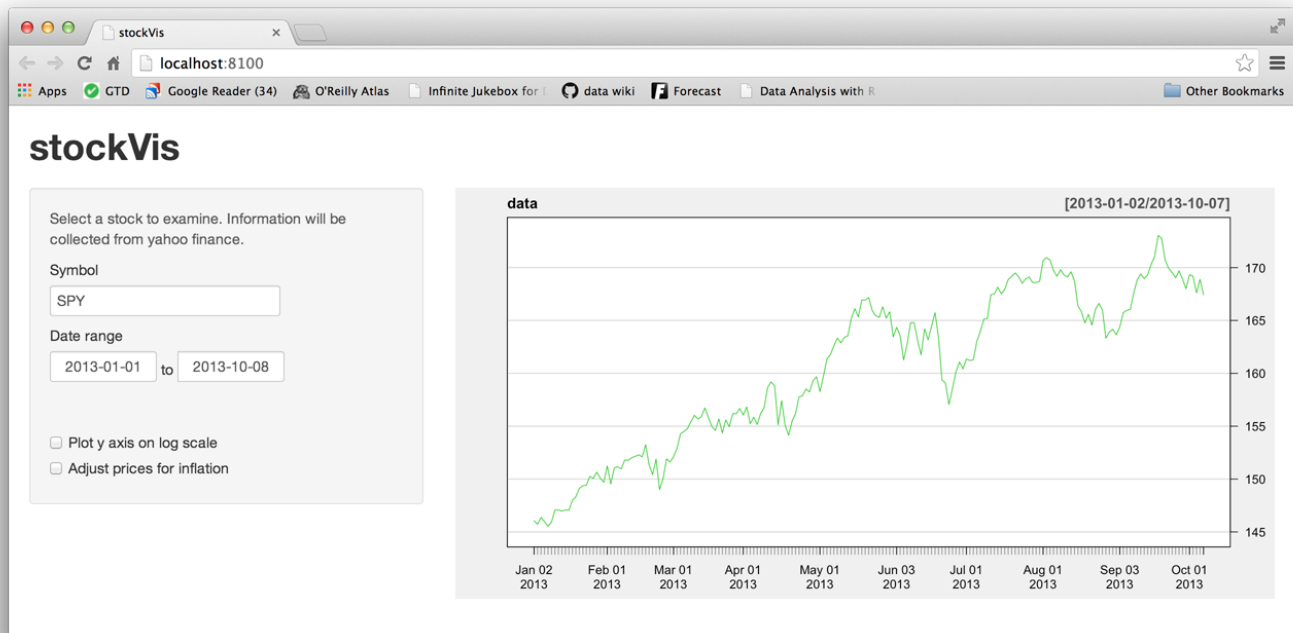
StockVis uses R's `quantmod` package, so you'll need to install `quantmod` with `install.packages("quantmod")` if you do not already have it.

```
runApp("stockVis")
```

## A new app: stockVis

The `stockVis` app looks up stock prices by ticker symbol and displays the results as a line chart. The app lets you

1. Select a stock to examine
2. Pick a range of dates to review
3. Choose whether to plot stock prices or the log of the stock prices on the y axis, and
4. Decide whether or not to correct prices for inflation.



Note that the “Adjust prices for inflation” check box doesn’t work yet. One of our tasks in this lesson is to fix this check box.

By default, stockVis displays the SPY ticker (an index of the entire S & P 500). To look up a different stock, type in a stock symbol that Yahoo finance will recognize. You can find a list of Yahoo’s stock symbols [here](#). Some common symbols are GOOG (Google), AAPL (Apple), and GS (Goldman Sachs).

StockVis relies heavily on two functions from the `quantmod` package:

1. It uses `getSymbols` to download financial data straight into R from websites like [Yahoo finance](#) and the [Federal Reserve Bank of St. Louis](#).
2. It uses `chartSeries` to display prices in an attractive chart.

StockVis also relies on an R script named `helpers.R`, which contains a function that adjusts stock prices for inflation.

## Check boxes and date ranges

The stockVis app uses a few new widgets.

- a date range selector, created with `dateRangeInput`, and
- a couple of check boxes made with `checkboxInput`. Check box widgets are very simple. They return a `TRUE` when the check box is checked, and a `FALSE` when the check box is not checked.

The check boxes are named `log` and `adjust` in the `ui.R` script, which means you can look them up as `input$log` and `input$adjust` in the `server.R` script. If you’d like to review how to use widgets and their values, check out [Lesson 3](#) and [Lesson 4](#).

## Streamline computation

The stockVis app has a problem.

Examine what will happen when you click “Plot y axis on the log scale.” The value of `input$log` will change, which will cause the entire expression in `renderPlot` to re-run:

```
output$plot <- renderPlot({
```



```

data <- getSymbols(input$symb, src = "yahoo",
  from = input$dates[1],
  to = input$dates[2],
  auto.assign = FALSE)

chartSeries(data, theme = chartTheme("white"),
  type = "line", log.scale = input$log, TA = NULL)
})

```

Each time `renderPlot` re-runs

1. it re-fetches the data from Yahoo finance with `getSymbols`, and
2. it re-draws the chart with the correct axis.

This is not good, because you do not need to re-fetch the data to re-draw the plot. In fact, Yahoo finance will cut you off if you re-fetch your data too often (because you begin to look like a bot). But more importantly, re-running `getSymbols` is unnecessary work, which can slow down your app and consume server bandwidth.

## Reactive expressions

You can limit what gets re-run during a reaction with reactive expressions.

A reactive expression is an R expression that uses widget input and returns a value. The reactive expression will update this value whenever the original widget changes.

To create a reactive expression use the `reactive` function, which takes an R expression surrounded by braces (just like the `render*` functions).

For example, here's a reactive expression that uses the widgets of `stockVis` to fetch data from Yahoo.

```

dataInput <- reactive({
  getSymbols(input$symb, src = "yahoo",
    from = input$dates[1],
    to = input$dates[2],
    auto.assign = FALSE)
})

```

When you run the expression, it will run `getSymbols` and return the results, a data frame of price data. You can use the expression to access price data in `renderPlot` by calling `dataInput()`.

```

output$plot <- renderPlot({
  chartSeries(dataInput(), theme = chartTheme("white"),
    type = "line", log.scale = input$log, TA = NULL)
})

```

Reactive expressions are a bit smarter than regular R functions. They cache their values and know when their values have become outdated. What does this mean? The first time that you run a reactive expression, the expression will save its result in your computer's memory. The next time you call the reactive expression, it can return this saved result without doing any computation (which will make your app faster).

The reactive expression will only return the saved result if it knows that the result is up-to-date. If the reactive expression has learned that the result is obsolete (because a widget has changed), the expression will recalculate the result. It then returns the new result and saves a new copy. The reactive expression will use this new copy until it too becomes out of date.

Let's summarize this behavior

A reactive expression saves its result the first time you run it.

- The next time the reactive expression is called, it checks if the saved value has become out of date (i.e., whether the widgets it depends on have changed).
- If the value is out of date, the reactive object will recalculate it (and then save the new result).
- If the value is up-to-date, the reactive expression will return the saved value without doing any computation.

You can use this behavior to prevent Shiny from re-running unnecessary code. Consider how a reactive expression will work in the new stockVis app below.

```
# server.R

library(quantmod)
source("helpers.R")

shinyServer(function(input, output) {

  dataInput <- reactive({
    getSymbols(input$symb, src = "yahoo",
              from = input$dates[1],
              to = input$dates[2],
              auto.assign = FALSE)
  })

  output$plot <- renderPlot({
    chartSeries(dataInput(), theme = chartTheme("white"),
               type = "line", log.scale = input$log, TA = NULL)
  })
})
```

When you click “Plot y axis on the log scale”, `input$log` will change and `renderPlot` will re-execute. Now

1. `renderPlot` will call `dataInput()`
2. `dataInput` will check that the dates and `symb` widgets have not changed
3. `dataInput` will return its saved data set of stock prices *without re-fetching data from Yahoo*
4. `renderPlot` will re-draw the chart with the correct axis.

## Dependencies

What if your user changes the stock symbol in the `symb` widget?

This will make the plot drawn by `renderPlot` out of date, but `renderPlot` no longer calls `input$symb`. Will Shiny know that `input$symb` has made plot out of date?

Yes, Shiny will know and will redraw the plot. Shiny keeps track of which reactive expressions an `output` object depends on, as well as which widget inputs. Shiny will automatically re-build an object if

- an `input` value in the object's `render*` function changes, or
- a reactive expression in the object's `render*` function becomes obsolete

Think of reactive expressions as links in a chain that connect `input` values to `output` objects. The objects in `output` will respond to changes made anywhere downstream in the chain. (You can fashion a long chain because reactive expressions can call other reactive expressions).

Only call a reactive expression from within a `reactive` or a `render*` function. Why? Only these R functions are equipped to deal with reactive output, which can change without warning. In fact, Shiny will prevent you from calling reactive expressions outside of these functions.

## Warm up

Time to fix the broken check box for “Adjust prices for inflation.” Your user should be able to toggle between prices adjusted for inflation and prices that have not been adjusted.

The `adjust` function in `helpers.R` uses the [Consumer Price Index](#) data provided by the Federal Reserve Bank of St. Louis to transform historical prices into present day values. But how can you implement this in the app?

Here’s one solution below, but it is not ideal. Can you spot why? Once again it has to do with `input$log`.

```
# server.R

library(quantmod)
source("helpers.R")

shinyServer(function(input, output) {

  dataInput <- reactive({
    getSymbols(input$symb, src = "yahoo",
              from = input$dates[1],
              to = input$dates[2],
              auto.assign = FALSE)
  })

  output$plot <- renderPlot({
    data <- dataInput()
    if (input$adjust) data <- adjust(dataInput())

    chartSeries(data, theme = chartTheme("white"),
                type = "line", log.scale = input$log, TA = NULL)
  })
})
```

Reveal answer

## Your Turn

Fix this problem by adding a new reactive expression to the app. The reactive expression should take the value of `dataInput` and return an adjusted (or not adjusted) copy of the data.

When you think you have it, compare your solution to the model answer below. Make sure you understand what calculations will happen and what calculations will not happen in your app when your user clicks “Plot y axis on the log scale”.

Reveal answer

## Recap

You can make your apps faster by modularizing your code with reactive expressions.

- A reactive expression takes `input` values, or values from other reactive expressions, and returns a new value
- Reactive expressions save their results, and will only re-calculate if their input has changed
- Create reactive expressions with `reactive({ })`
- Call reactive expressions with the name of the expression followed by parentheses `()`

- Only call reactive expressions from within other reactive expressions or `render*` functions

You can now create sophisticated, streamlined Shiny apps. The final lesson in this tutorial will show you how to share your apps with others.

[Continue to lesson 7](#)

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Nellster

---

Hello,

Great tutorial! I enjoy it very much.

I just have one problem: when I run the last step of this lesson, which uses the `adjust` function of `helper.R`, I get an error message:

```
Error in .func() : could not find function "adjust"
```

`helpers.R` was loaded correctly. I then checked inside `helpers.R`, to see where was this `adjust` function: I could not find it!

Did I miss something?

Thanks in advance

Garrett

---

Nellster, no wonder your code didn't work :) `adjust` is definitely in `helpers.R`, but you might have the wrong `helpers.R` file. Try downloading this [version](#).

If you see `percent_map` in your `helpers.R` file, it means that you are using the file from lesson 5. Lesson 6 has a new `helpers.R` file. I apologize about the name

comments powered by [Disqus](#)

## 1.10 LESSON 7: Share your apps

□ 1 2 3 4 5 6 Lesson 7 □

### LESSON 7

# Share your apps

You can now build a useful Shiny app, but can you share it with others? This lesson will show you several ways to share your Shiny apps.

When it comes to sharing Shiny apps, you have two basic options:

1. Share your Shiny app as two files: `server.R` and `ui.R`. This is the simplest way to share an app, but it works only if your users have R on their own computer (and know how to use it). Users can use these scripts to launch the app from their own R session, just like you've been launching the apps.
2. Share your Shiny app as a web page. This is definitely the most user friendly way to share a Shiny app. Your users can navigate to your app through the internet with a web browser. They will find your app fully rendered, up to date, and ready to go.

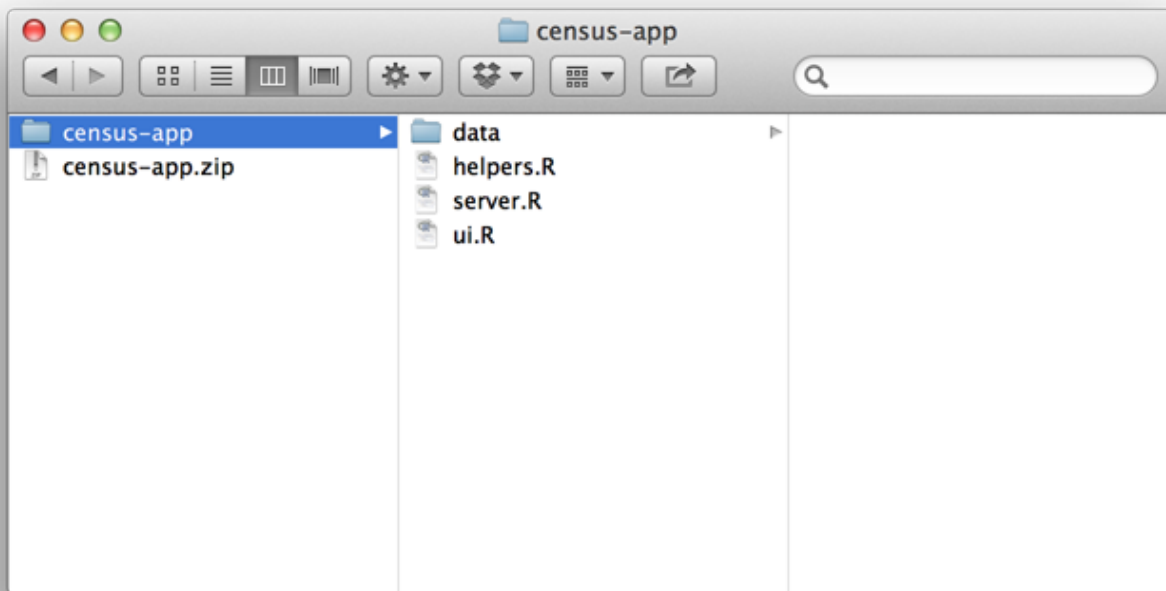
## Share as two R files

Anyone with R can run your Shiny app. They will need a copy of your `server.R` and `ui.R` files, as well as any supplementary materials used in your app (e.g., `www` folders or `helpers.R` files).

To send your files to another user, email the files (perhaps in a zip file) or host the files online.

Your user can place the files into an app directory in their working directory. They can launch the app in R with the same commands you used on your computer.

```
# install.packages("shiny")  
library(shiny)  
runApp("census-app")
```



Shiny has three built in commands that make it easy to use files that are hosted online: `runUrl` , `runGitHub` , and `runGist` .

## runUrl

`runUrl` will download and launch a Shiny app straight from a weblink.

To use `runURL` :

- Save your Shiny app's directory as a zip file
- Host that zip file at its own link on a web page. Anyone with access to the link can launch the app from inside R by running:

```
library(shiny)
runUrl("<the weblink>")
```

## runGitHub

If you don't have your own web page to host the files at, you can host your the files for free at [www.github.com](http://www.github.com).

Github is a popular project hosting site for R developers since it does more than just host files. Github provides many features to support collaboration, such as issue trackers, wikis, and close integration with the [git](https://git-scm.com/) version control system. To use Github, you'll need to sign up (it's free) and choose a user name.

To share an app through Github, create a project repository on Github. Then store your `server.R` and `ui.R` files in the repository, along with any supplementary files that the app uses.

Your users can launch your app by running:

```
runGitHub("<your repository name>", "<your user name>")
```

## runGist

If you want an anonymous way to post files online, Github offers a pasteboard service for sharing files at [gist.github.com](https://gist.github.com). You don't need to sign up for Github to use this service. Even if you have a Github account, Gist can be a simple, quick way to share Shiny projects.

To share your app as a Gist:

- Copy and paste your `server.R` and `ui.R` files to the Gist web page.
- Note the URL that Github gives the Gist.

Once you've made a Gist, your users can launch the app with `runGist("<gist number>")` where "`<gist number>`" is the number that appears at the end of your Gist's web address.

Here's an example of an app hosted as a Gist. You could launch this app with:

```
runGist("3239667")
```

## Share as a web page

All of the above methods share the same limitation. They require your user to have R and Shiny installed on their computer.

However, Shiny creates the perfect opportunity to share output with people who do *not* have R (and have no intention of getting it). Your Shiny app happens to be one of the most widely used communication tools in the world: a web page. If you host the app at its own URL, users can visit the app (and not need to worry about code).

If you are familiar with web hosting or have access to an IT department, you can host your Shiny apps yourself.

If you'd prefer an easier experience or need support, RStudio offers three ways to host your Shiny app as a web page:

1. Shinyapps.io.
2. Shiny Server, and
3. Shiny Server Pro

## Shinyapps.io

The easiest way to turn your Shiny app into a web page is to use [shinyapps.io](https://shinyapps.io), RStudio's hosting service for Shiny apps.

[shinyapps.io](https://shinyapps.io) lets you upload your app straight from your R session to a server hosted by RStudio. You complete control over your app including server administration tools. To find out more about [shinyapps.io](https://shinyapps.io) visit [shinyapps.io](https://shinyapps.io).

## Shiny Server

Shiny Server is a companion program to Shiny that builds a web server designed to host Shiny apps. It's free, open source, and available from Github.

Shiny Server is a server program that Linux servers can run to host a Shiny app as a web page. To use Shiny Server, you'll need a Linux server that has explicit support for Ubuntu 12.04 or greater (64 bit) and CentOS/RHEL 5 (64 bit). If you are not using an explicitly supported distribution, you can still use Shiny Server by building it from source.

You can host multiple Shiny applications on multiple web pages with the same Shiny Server, and you can deploy the apps from behind a firewall.

To see detailed instructions for installing and configuring a Shiny Server, visit the [Shiny Server guide](#).

## Shiny Server Pro

Shiny Server will get your app to the web and take care of all of your Shiny publishing needs. However, if you use Shiny in a for-profit setting, you may want to give yourself the server tools that come with most paid server programs, such as

- Password authentication
- SSL support
- Administrator tools
- Priority support
- and more.

If so, check out [Shiny Server Pro](#), RStudio's paid professional version of Shiny Server.

## Recap

Shiny apps are easy to share. You can share your app as a couple of R scripts, or as a fully functioning web app with its own URL. Each method has its own advantages.

You learned:

- Anyone can launch your app as long as they have a copy of R, Shiny, and a copy of your app's files.
- `runUrl`, `runGitHub`, and `runGit` make it simple to share and retrieve Shiny files from web links.
- You can turn your app into a live web app at its own URL with [shinyapps.io](#).
- You can use the open source Shiny Server to build a Linux server that hosts Shiny apps.
- If you need closer control, or want to manage large volumes of traffic, you can purchase [Shiny Server Pro](#) from RStudio.

Congratulations. You've worked through the entire Shiny development process. You can build a sophisticated, reactive app, deploy it, and share it with others. Users can interact with your data and follow your stories in a new way.

The next step is to practice, and then explore the advanced features of Shiny.

The [Shiny Dev Center](#) can help you along the way. It hosts a [gallery](#) of inspiring apps, along with the code that makes the apps.

The Shiny Dev Center also includes an [articles](#) section for continuing education. Each article examines an intermediate to advanced Shiny topic in depth.

You now know enough to build your own Shiny apps. See what you can do!

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Kalyana

---

Excellent help for the Shiny starters, thankyou



Camilo Mora

---

This is a great tutorial,

While loading my app into [ShinyApps.io](http://ShinyApps.io)  
using:

```
deployApp("01_hello")
```

I get the following error:

```
Error: Unhandled Exception: 'NoneType' object has no attribute '__getitem__'
```

and while the app shows up in the web-page of [ShinyApps.io](http://ShinyApps.io), it appears as  
"undeployed".

Any clue what may be causing this. I google this with no info available.

Yimi  
[comments powered by Disqus](#)

---

control over your app including server administration tools. To find out more about shinyapps.io visit [shinyapps.io](https://shinyapps.io).

## Shiny Server

Shiny Server is a companion program to Shiny that builds a web server designed to host Shiny apps. It's free, open source, and available from Github.

Shiny Server is a server program that Linux servers can run to host a Shiny app as a web page. To use Shiny Server, you'll need a Linux server that has explicit support for Ubuntu 12.04 or greater (64 bit) and CentOS/RHEL 5 (64 bit). If you are not using an explicitly supported distribution, you can still use Shiny Server by building it from source.

You can host multiple Shiny applications on multiple web pages with the same Shiny Server, and you can deploy the apps from behind a firewall.

=== [p]To see detailed instructions for installing and configuring a Shiny Server, visit the Shiny Server [a href="https://github.com/rstudio/shiny-server/blob/master/README.md"]guide[/a].[/p] ===

## Shiny Server Pro

Shiny Server will get your app to the web and take care of all of your Shiny publishing needs. However, if you use Shiny in a for-profit setting, you may want to give yourself the server tools that come with most paid server programs, such as

- Password authentication
- SSL support
- Administrator tools
- Priority support
- and more.

If so, check out [Shiny Server Pro](#), RStudio's paid professional version of Shiny Server.

## Recap

Shiny apps are easy to share. You can share your app as a couple of R scripts, or as a fully functioning web app with its own URL. Each method has its own advantages.

You learned:

- Anyone can launch your app as long as they have a copy of R, Shiny, and a copy of your app's files.
- `runUrl`, `runGitHub`, and `runGit` make it simple to share and retrieve Shiny files from web links.
- You can turn your app into a live web app at its own URL with [shinyapps.io](https://shinyapps.io).
- You can use the open source Shiny Server to build a Linux server that hosts Shiny apps.
- If you need closer control, or want to manage large volumes of traffic, you can purchase [Shiny Server Pro](#) from RStudio.

Congratulations. You've worked through the entire Shiny development process. You can build a sophisticated, reactive app, deploy it, and share it with others. Users can interact with your data and follow your stories in a new way.

The next step is to practice, and then explore the advanced features of Shiny.

The [Shiny Dev Center](#) can help you along the way. It hosts a [gallery](#) of inspiring apps, along with the code that

The Shiny Dev Center also includes an [articles](#) section for continuing education. Each article examines an intermediate to advanced Shiny topic in depth.

You now know enough to build your own Shiny apps. See what you can do!

---

Let us know what you think! If you have a sophisticated question, or want an in depth answer, please post at the [Shiny Discussion Forum](#), where we can respond at length. For help with code, check out [How to get help](#).

comments powered by [Disqus](#)

```
# Note: percent map is designed to work with the counties data set
# It may not work correctly with other data sets if their row order does
# not exactly match the order in which the maps package plots counties
percent_map <- function(var, color, legend.title, min = 0, max = 100) {

  # generate vector of fill colors for map
  shades <- colorRampPalette(c("white", color))(100)

  # constrain gradient to percents that occur between min and max
  var <- pmax(var, min)
  var <- pmin(var, max)
  percents <- as.integer(cut(var, 100,
    include.lowest = TRUE, ordered = TRUE))
  fills <- shades[percents]

  # plot choropleth map
  map("county", fill = TRUE, col = fills,
    resolution = 0, lty = 0, projection = "polyconic",
    myborder = 0, mar = c(0,0,0,0))

  # overlay state borders
  map("state", col = "white", fill = FALSE, add = TRUE,
    lty = 1, lwd = 1, projection = "polyconic",
    myborder = 0, mar = c(0,0,0,0))

  # add a legend
  inc <- (max - min) / 4
  legend.text <- c(paste0(min, " % or less"),
    paste0(min + inc, " %"),
    paste0(min + 2 * inc, " %"),
    paste0(min + 3 * inc, " %"),
    paste0(max, " % or more"))

  legend("bottomleft",
    legend = legend.text,
    fill = shades[c(1, 25, 50, 75, 100)],
    title = legend.title)
}
```

```
library(shiny)

shinyUI(fluidPage(
  titlePanel("stockVis"),

  sidebarLayout(
    sidebarPanel(
      helpText("Select a stock to examine.
        Information will be collected from yahoo finance."),

      textInput("symb", "Symbol", "SPY"),

      dateRangeInput("dates",
        "Date range",
        start = "2013-01-01",
        end = as.character(Sys.Date())),

      actionButton("get", "Get Stock"),

      br(),
      br(),

      checkboxInput("log", "Plot y axis on log scale",
        value = FALSE),

      checkboxInput("adjust",
        "Adjust prices for inflation", value = FALSE)
    ),

    mainPanel(plotOutput("plot"))
  )
))
```

```
# server.R
```

```
library(quantmod)
source("helpers.R")
```

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

  output$plot <- renderPlot({
    data <- getSymbols(input$symb, src = "yahoo",
      from = input$dates[1],
      to = input$dates[2],
      auto.assign = FALSE)

    chartSeries(data, theme = chartTheme("white"),
      type = "line", log.scale = input$log, TA = NULL)
  })

})
```

```
if (!exists(".inflation")) {
  .inflation <- getSymbols('CPIAUCNS', src = 'FRED',
    auto.assign = FALSE)
}

# adjusts yahoo finance data with the monthly consumer price index
# values provided by the Federal Reserve of St. Louis
# historical prices are returned in present values
adjust <- function(data) {

  latestcpi <- last(.inflation)[[1]]
  inf.latest <- time(last(.inflation))
  months <- split(data)

  adjust_month <- function(month) {
    date <- substr(min(time(month[1]), inf.latest), 1, 7)
    coredata(month) * latestcpi / .inflation[date][[1]]
  }

  adjs <- lapply(months, adjust_month)
  adj <- do.call("rbind", adjs)
  axts <- xts(adj, order.by = time(data))
  axts[, 5] <- Vo(data)
  axts
}
```

## 2 Articles

### 2.1 Articles

# Articles

## The basics

If you've been through the [tutorial](#) and need a refresher, these articles are a good place to start. They describe the lay of the land.

[The basic parts of a Shiny app](#)

[How to build a Shiny app](#)

[How to launch a Shiny app](#)

[How to get help](#)

[The Shiny cheat sheet](#)

[Single-file Shiny apps](#)

[App formats and launching methods](#)

[Persistent data storage in Shiny apps](#)

## Extend Shiny

These packages provide advanced features that can enhance your Shiny apps.

[shinythemes](#) - CSS themes ready to use with Shiny

[shinydashboard](#) - Shiny powered dashboards

[htmlwidgets](#) - A framework for embedding JavaScript visualizations into R. Ready to use examples include:

[leaflet](#) - Geo-spatial mapping ([article](#))

[dygraphs](#) - Time series charting ([article](#))

[MetricsGraphics](#) - Scatterplots and line charts with D3

[networkD3](#) - Graph data visualization with D3

[DataTables](#) - Tabular data display ([article](#))

[threejs](#) - 3D scatterplots and globes

[rCharts](#) - Multiple JavaScript charting libraries

[d3heatmap](#) - Heatmaps ([article](#))

[diagrammeR](#) - Graph and flowchart diagrams ([article](#))

## Layouts and UI

These articles explain how to control the layout, user-interface, and general appearance of your Shiny apps.

[Application layout guide](#)

[Display modes](#)

[Tabsets](#)

[Customize your UI with HTML](#)



- [Build your entire UI with HTML](#)
- [Build a dynamic UI that reacts to user input](#)
- [Shiny HTML Tags Glossary](#)
- [Progress indicators](#)

## Deploying apps

These articles describe the different ways to share your Shiny apps with users.

- [Getting started with shinyapps.io](#)
- [Setting up custom domains with shinyapps.io](#)
- [Scaling and Performance Tuning with shinyapps.io](#)
- [Share data across sessions with shinyapps.io](#)
- [Migrating shinyapps.io authentication](#)
- [Introduction to Shiny Server](#)
- [Save your app as a function](#)
- [Sharing apps to run locally](#)

## Interactive documents

These articles explain how to add Shiny components to R Markdown reports.

- [Introduction to R Markdown](#)
- [Introduction to interactive documents](#)
- [R Markdown integration in the RStudio IDE](#)
- [The R Markdown Cheat sheet](#)

## Widgets

These articles describe Shiny's pre-built widgets and provide ideas on how to use them. (See also [Lesson 3](#) in the tutorial, and the [Widgets](#) section in the [gallery](#).)

- [Using Action Buttons](#)
- [Using sliders](#)
- [Help users download data from your app](#)
- [Using selectize input](#)

## Outputs

These articles show you how to create and use different output objects, the parts of your app that display results and react to user input.

- [Render images in a Shiny app](#)
- [How to use DataTables in a Shiny App](#)

## Reactive programming

These articles describe reactivity from a conceptual level. Understanding reactivity will help you build apps that are more efficient, robust, and correct.

- [Reactivity: An overview](#)
- [Stop reactions with isolate\(\)](#)
- [Execution scheduling](#)
- [How to understand reactivity in R](#)

## Best practices

These articles contain ideas that can improve your Shiny workflow and help you create faster, more efficient apps.

[Write error messages for your UI with validate](#)

[Scoping rules for Shiny apps](#)

[Debugging techniques for Shiny apps](#)

[Learn about your user with session\\$clientData](#)

[Unicode characters in Shiny apps](#)

## Customizing Shiny

These articles suggest ways to create custom Shiny widgets, layouts and outputs; or to combine Shiny with other web technologies.

[Style your apps with CSS](#)

[Build custom input objects](#)

[Build custom output objects](#)

[Add Google Analytics to a Shiny app](#)

## Shiny Server Pro

Here are some of the unique things you can do when you deploy your apps with Shiny Server Pro

[How to create User Privileges](#)

[Allow different libraries for different apps](#)

## Interactive plots

Create interactive plots with base and ggplot2 graphics

[Interactive plots](#)

[Selecting rows of data](#)

[Interactive plots - advanced](#)

## Upgrade notes

Notes for upgrading to particular versions of Shiny

[Upgrade notes for Shiny 0.11](#)

[Upgrade notes for Shiny 0.12](#)

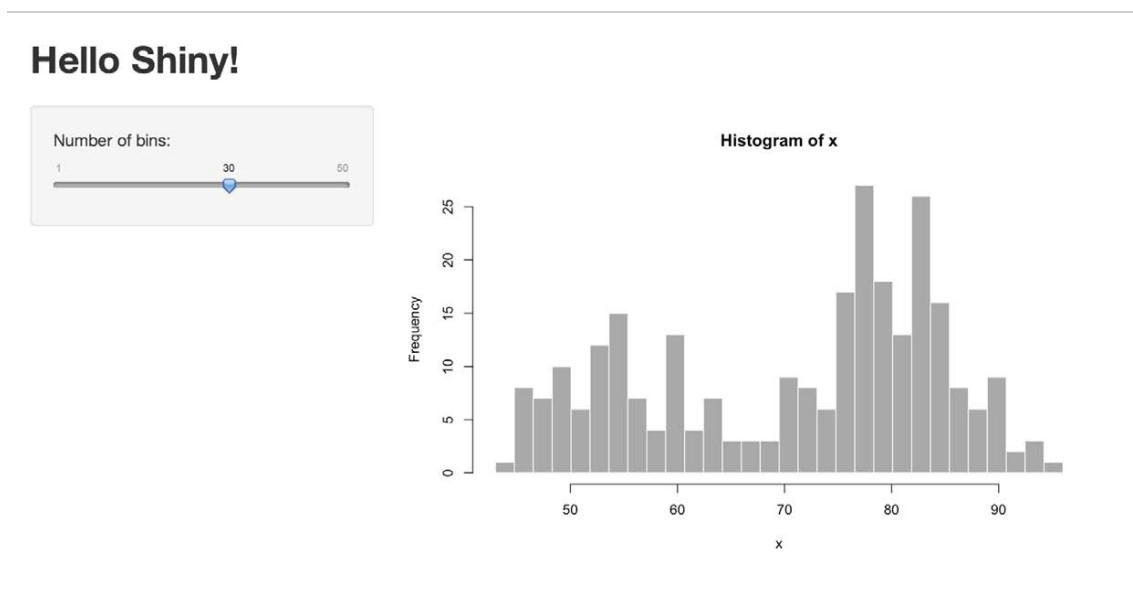
## 2.2 The basic parts of a Shiny app

# 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

### Shiny Text

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 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:

Choose a dataset:

Number of observations to view:

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

We've reviewed a lot of code and covered a lot of conceptual ground in the first three examples. The [next article](#) 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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

## 2.3 How to build a Shiny app

# How to build a Shiny app

ADDED: 06 JAN 2014

Let's walk through the steps of building a simple Shiny application. A Shiny application is simply a directory containing a user-interface definition, a server script, and any additional data, scripts, or other resources required to support the application.

## UI & Server

To get started building the application, create a new empty directory wherever you'd like, then create empty `ui.R` and `server.R` files within in. For purposes of illustration we'll assume you've chosen to create the application at `~/shinyapp`:

```
~/shinyapp
|-- ui.R
|-- server.R
```

Now we'll add the minimal code required in each source file. We'll first define the user interface by calling the function `pageWithSidebar` and passing its result to the `shinyUI` function:

`ui.R`

```
library(shiny)

# Define UI for miles per gallon application
shinyUI(pageWithSidebar(

  # Application title
  headerPanel("Miles Per Gallon"),

  sidebarPanel(),

  mainPanel()
))
```

The three functions `headerPanel`, `sidebarPanel`, and `mainPanel` define the various regions of the user-interface. The application will be called "Miles Per Gallon" so we specify that as the title when we create the header panel. The other panels are empty for now.

Now let's define a skeletal server implementation. To do this we call `shinyServer` and pass it a function that accepts two parameters, `input` and `output`:

`server.R`

```
library(shiny)
```

```
# Define server logic required to plot various variables against mpg
shinyServer(function(input, output) {

})
```

Our server function is empty for now but later we'll use it to define the relationship between our inputs and outputs.

We've now created the most minimal possible Shiny application. You can run the application by calling the `runApp` function as follows:

```
> library(shiny)
> runApp("~/shinyapp")
```

If everything is working correctly you'll see the application appear in your browser looking something like this:



We now have a running Shiny application however it doesn't do much yet. In the next section we'll complete the application by specifying the user-interface and implementing the server script.

## Inputs & Outputs

### Adding Inputs to the Sidebar

The application we'll be building uses the `mtcars` data from the `R datasets` package, and allows users to see a box-plot that explores the relationship between miles-per-gallon (MPG) and three other variables (Cylinders, Transmission, and Gears).

We want to provide a way to select which variable to plot MPG against as well as provide an option to include or exclude outliers from the plot. To do this we'll add two elements to the sidebar, a `selectInput` to specify the variable and a `checkboxInput` to control display of outliers. Our user-interface definition looks like this after adding these elements:

ui.R

```
library(shiny)
```

```
# Define UI for miles per gallon application
```

```

shinyUI(pageWithSidebar(

  # Application title
  headerPanel("Miles Per Gallon"),

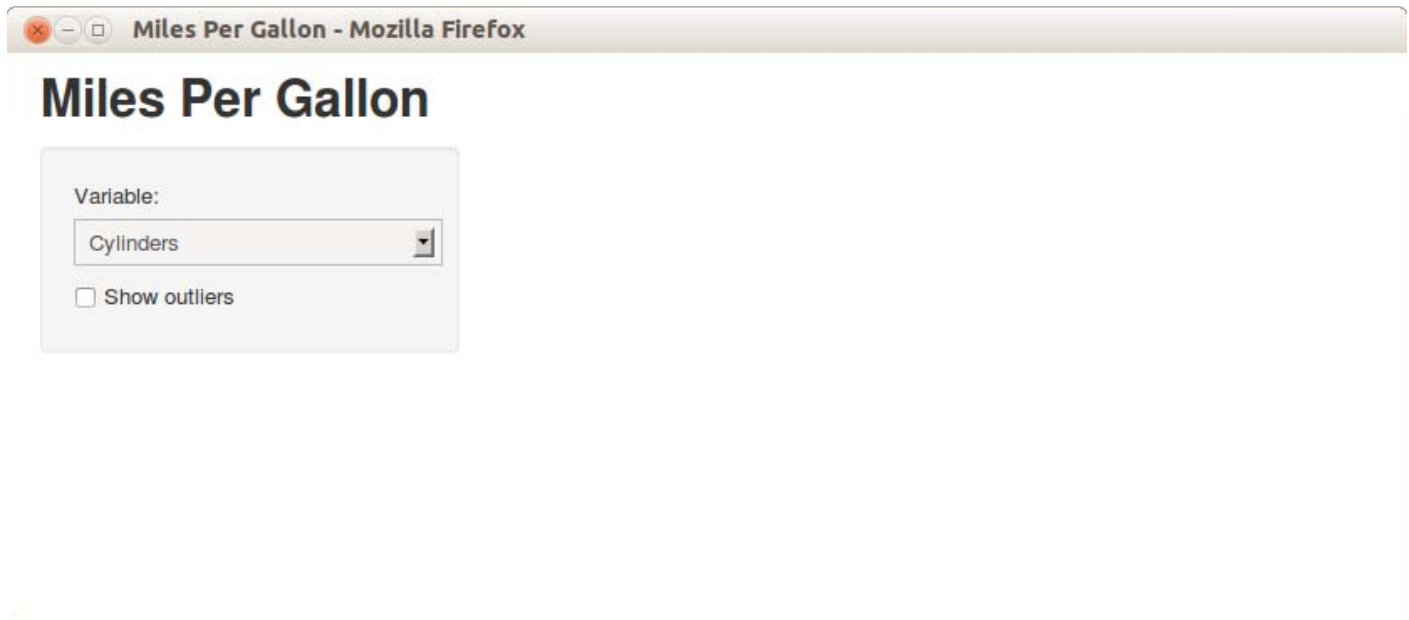
  # Sidebar with controls to select the variable to plot against mpg
  # and to specify whether outliers should be included
  sidebarPanel(
    selectInput("variable", "Variable:",
               list("Cylinders" = "cyl",
                    "Transmission" = "am",
                    "Gears" = "gear")),

    checkboxInput("outliers", "Show outliers", FALSE)
  ),

  mainPanel()
))

```

If you run the application again after making these changes you'll see the two user-inputs we defined displayed within the sidebar:



## Creating the Server Script

Next we need to define the server-side of the application which will accept inputs and compute outputs. Our server.R file is shown below, and illustrates some important concepts:

- Accessing input using slots on the `input` object and generating output by assigning to slots on the `output` object.
- Initializing data at startup that can be accessed throughout the lifetime of the application.
- Using a reactive expression to compute a value shared by more than one output.

The basic task of a Shiny server script is to define the relationship between inputs and outputs. Our script does this by accessing inputs to perform computations and by assigning reactive expressions to output slots.

Here is the source code for the full server script (the inline comments explain the implementation techniques in more detail):

```
server.R
```

```

library(shiny)
library(datasets)

# We tweak the "am" field to have nicer factor labels. Since this doesn't
# rely on any user inputs we can do this once at startup and then use the
# value throughout the lifetime of the application
mpgData <- mtcars
mpgData$am <- factor(mpgData$am, labels = c("Automatic", "Manual"))

# Define server logic required to plot various variables against mpg
shinyServer(function(input, output) {

  # Compute the formula text in a reactive expression since it is
  # shared by the output$caption and output$mpgPlot expressions
  formulaText <- reactive({
    paste("mpg ~", input$variable)
  })

  # Return the formula text for printing as a caption
  output$caption <- renderText({
    formulaText()
  })

  # Generate a plot of the requested variable against mpg and only
  # include outliers if requested
  output$mpgPlot <- renderPlot({
    boxplot(as.formula(formulaText()),
            data = mpgData,
            outline = input$outliers)
  })
})

```

The use of `renderText` and `renderPlot` to generate output (rather than just assigning values directly) is what makes the application reactive. These reactive wrappers return special expressions that are only re-executed when their dependencies change. This behavior is what enables Shiny to automatically update output whenever input changes.

## Displaying Outputs

The server script assigned two output values: `output$caption` and `output$mpgPlot`. To update our user interface to display the output we need to add some elements to the main UI panel.

In the updated user-interface definition below you can see that we've added the caption as an `h3` element and filled in its value using the `textOutput` function, and also rendered the plot by calling the `plotOutput` function:

ui.R

```

library(shiny)

# Define UI for miles per gallon application
shinyUI(pageWithSidebar(

  # Application title
  headerPanel("Miles Per Gallon"),

  # Sidebar with controls to select the variable to plot against mpg

```

```

# and to specify whether outliers should be included
sidebarPanel (
  selectInput ("variable", "Variable:",
    list ("Cylinders" = "cyl",
          "Transmission" = "am",
          "Gears" = "gear")),

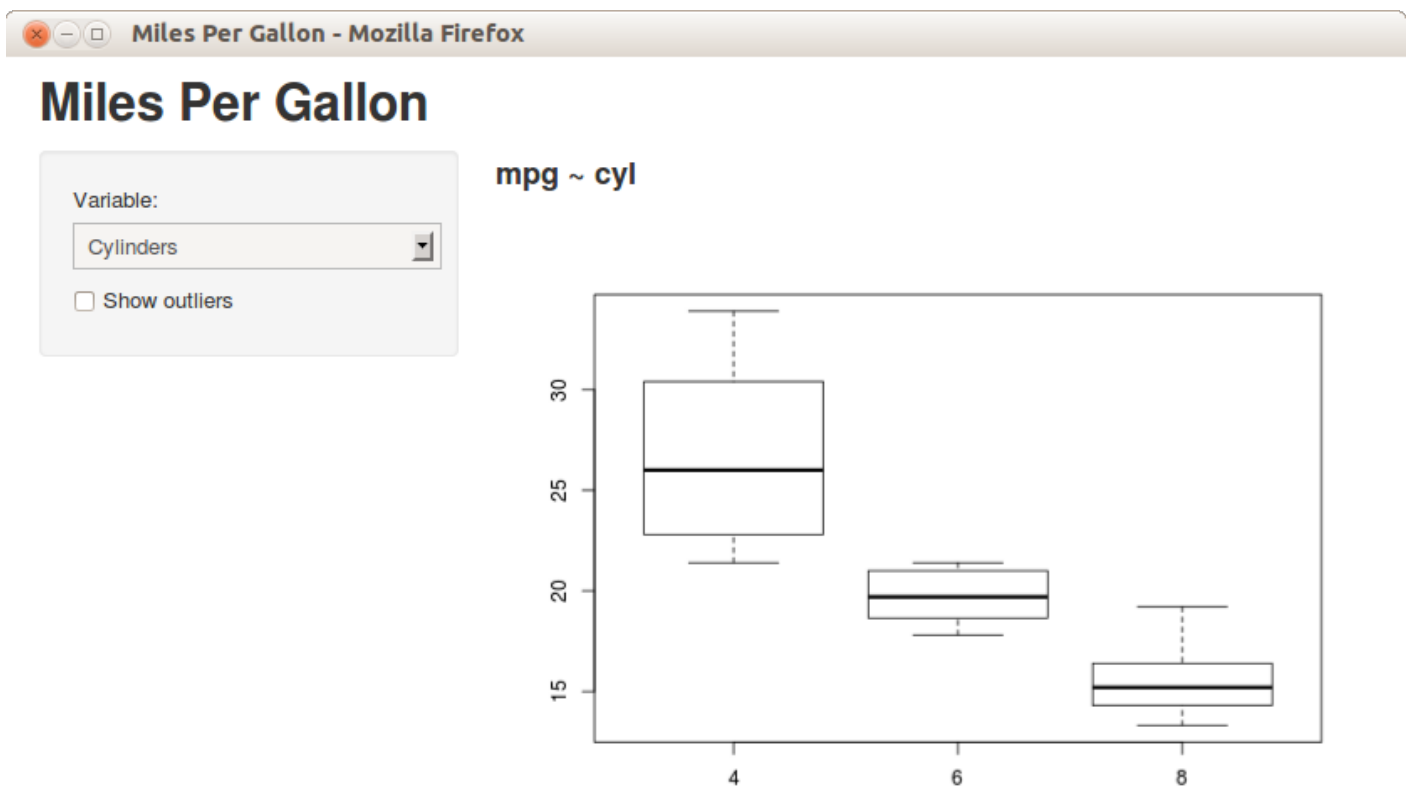
  checkboxInput ("outliers", "Show outliers", FALSE)
),

# Show the caption and plot of the requested variable against mpg
mainPanel (
  h3(textOutput ("caption")),

  plotOutput ("mpgPlot")
)
))

```

Running the application now shows it in its final form including inputs and dynamically updating outputs:



Now that we've got a simple application running we'll probably want to make some changes. The [next article](#) covers the basic cycle of editing, running, and debugging Shiny applications.

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

John Wandeto

---

Dear All. This is fantastic. I hope to build an application to analyse students tests performance, award grades, produce results with appropriate comments/suggestion. Thank you

Mark Cavanaugh

---

Thanks for this...very helpful...and I look forward to build with shiny.

comments powered by Disqus



## 2.4 How to launch a Shiny app

# How to launch a Shiny app

ADDED: 06 JAN 2014

In previous articles[1, 2] you've been calling `runApp` to run the example applications. This function starts the application and opens up your default web browser to view it. The call is blocking, meaning that it prevents traditional interaction with the console while the application is running.

To stop the application you simply interrupt R – you can do this by pressing the Ctrl-C in some R front ends, or the Escape key in RStudio, or by clicking the stop button if your R environment provides one.

## Running in a Separate Process

If you don't want to block access to the console while running your Shiny application you can also run it in a separate process. You can do this by opening a terminal or console window and executing the following, where `~/shinyapp` should be replaced with the path to your application:

```
R -e "shiny: runApp('~/shinyapp')"
```

By default `runApp` starts the application on a randomly selected port. For example, it might start on port 4700, in which case you can connect to the running application by navigating your browser to `http://localhost:4700`.

In other articles, we discuss some techniques for debugging Shiny applications, including the ability to stop execution and inspect the current environment. To combine these techniques with running your applications in a separate terminal session, you'll need to call `runApp` from an interactive R session, instead of with the method here.

## Live Reloading

When you make changes to your underlying user-interface definition or server script you don't need to stop and restart your application to see the changes. Simply save your changes and then reload the browser to see the updated application in action.

One qualification to this: when a browser reload occurs Shiny explicitly checks the timestamps of the `ui.R` and `server.R` files to see if they need to be re-sourced. Shiny isn't aware of other scripts or data files that change, so if you use those files and modify them, a full stop and restart of the application is needed.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

## 2.5 How to get help

# How to get help

ADDED: 22 MAY 2014

BY: GARRETT GROLEMUND

Writing code can be tricky. Sometimes you will want more advice than you can find in R's [help pages](#).

This article will show you where to seek help, how to get it.

Support for R and Shiny has developed like the languages themselves, organically and through the excellent work of volunteers. You can be a part of this process. Simply ask your questions in the right places and in a clear manner.

Not only will your question help you, it will create a record that will benefit everyone who has the same question later.

## Where to seek help

You will get the quickest response if you submit your question to a place that people already visit to ask and answer similar questions. I recommend these resources for questions on the following topics:

- R (general): [StackOverflow](#)
- Shiny (general): [StackOverflow](#)
- Shiny: The [Shiny discussion group](#)
- Shiny Server: The [Shiny Server support forum](#)
- Shiny Server Pro: email the Shiny Server Pro support team<sup>1</sup>
- ShinyApps.io: The [ShinyApps.io discussion group](#)
- RStudio IDE: The [RStudio IDE support forum](#)

<sup>1</sup>Since *Shiny Server Pro* is a paid product, it has a customer support team.

These sites all have archives that you can search to see if your question has already been answered. If answered, you can get an immediate solution!

Once answered your question will go into these archives and expand the knowledge base in the Shiny community.

## How to get help

You will get the most useful help if you do these simple things:

1. Search the archives and check if your question has an answer already.
2. Write a [minimal reproducible example](#) that illustrates your problem.
3. Be precise. Include the exact error messages that you see.
4. Run `sessionInfo()` in R and include the output with your question (*sessionInfo()* displays the versions of R and its packages that you are using, as well as your OS. This is important information for debugging).
5. Be friendly and appreciative.

These steps will make it easier for a mentor to help you.

## How to write a good reproducible example

A reproducible example is a snippet of R code that someone can run and recreate your problem. Many bugs cannot be diagnosed unless you include the code that causes them.

A *good* reproducible example is:

1. Minimal - It should contain just enough code to recreate the bug. This will help both you and your mentors zero in on the problem.
2. Complete - It should contain *everything* a person needs to recreate the bug. In other words, a person should be able to copy and paste the code into R and see the bug.

You do not need to share your own code and data (if you do not want to). Often you can create an example that reproduces your bug with toy code and a dummy data set.

If you need to create a reproducible example of a Shiny app, we recommend you [save your app as a gist](#). When you write your question, include the `runGist` command that will launch your app. A `runGist` command looks like this:

```
shiny::runGist("3239667")
```

Don't forget to explain how to create the error in your app. You may need to provide instructions such as "Click this checkbox" or "Select this value."

If you need to create a reproducible example of Rcpp code, I recommend you supply a .cpp file that can be executed through `Rcpp::sourceCpp`. It is the easiest way of testing or debugging C++ / Rcpp code.

If you want more details on writing a good example, Hadley gives some advice [here](#).

## What about the Shiny Dev Center?

RStudio wants to make the [Shiny Dev Center](#) as useful as possible. I hope it can be a one stop shop for [advice](#), [wisdom](#), and [inspiration](#). However the Shiny Dev Center is not designed to be an interactive resource.

If you ask a question *about an article or tutorial* in its comments section, I will try my best to answer it there. However, the comments section is *not* a good place to ask for specific help with your code. Why not?

- Practicality - Volunteers do not hang out in the comments sections waiting to answer your questions.
- Efficiency - The resources listed above are searchable and well known, which makes it easy for others to benefit from your questions and answers. Not so for the comments.
- Redundancy - There is a good chance someone already answered your question at one of the help sites. Why not find out?

## Recap

You can get the help you need and make Shiny better in the process by visiting a help site specific to your question. Do not forget to make your question clear with a reproducible example, an error message, and the output of `sessionInfo()`.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)  
[Chrome](#)

## 2.6 Single-file Shiny apps

# Single-file Shiny apps

ADDED: 10 SEP 2014

BY: WINSTON CHANG

As of version 0.10.2, Shiny supports single-file applications. You no longer need to build separate `server.R` and `ui.R` files for your app; you can just create a file called `app.R` that contains both the server and UI components.

## Example

To create a single-file app, create a new directory (for example, `newdir/`) and place a file called `app.R` in the directory. To run it, call `runApp("newdir")`.

Your `app.R` file should call `shinyApp()` with an appropriate UI object and server function, as demonstrated below:

```
server <- function(input, output) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs), col = 'darkgray', border = 'white')
  })
}

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      sliderInput("obs", "Number of observations:", min = 10, max = 500, value = 100)
    ),
    mainPanel(plotOutput("distPlot"))
  )
)

shinyApp(ui = ui, server = server)
```

One nice feature about single-file apps is that you can copy and paste the entire app into the R console, which makes it easy to quickly share code for others to experiment with. For example, if you copy and paste the code above into the R command line, it will start a Shiny app.

## Details

The `shinyApp()` function returns an object of class `shiny.appobj`. When this is returned to the console, it is printed using the `print.shiny.appobj()` function, which launches a Shiny app from that object.

You can also use a similar technique to create and run files that aren't named `app.R` and don't reside in their own directory. If, for example, you create a file called `test.R` and have it call `shinyApp()` at the end, you could then run it from the console with:

```
print(source("test.R"))
```

When the file is sourced, it returns a `shiny.appobj` —but by default, the return value from `source()` isn't printed. Wrapping it in `print()` causes Shiny to launch it.

This method is handy for quick experiments, but it lacks some advantages that you get from having an `app.R` in its own directory. When you do `runApp("newdir")`, Shiny will monitor the file for changes and reload the app if you reload your browser, which is useful for development. This doesn't happen when you simply source the file. Also, Shiny Server and shinyapps.io expect an app to be in its own directory. So if you want to deploy your app, it should go in its own directory.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Günter Lutz-Misof

---

Would it be possible to create nested shiny apps. What I want to do is to realize a tabbed interface and each tab represents a single shiny app. all these apps should use the same dataset. Tested this and it works partially. If i integrate the histogram app then its shown in an iframe and works fine. If I want to use data from the main app it seems that I have lost the environment. And during loading the complete app it shows ERROR: [on\_request\_read] connection reset by peer. Is it possible that you provide an example how to do this, if this is possible?

ImAndy

---

For examples that require additional packages, how do I add, say, `library(dplyr)` in the above code?

Bk

---

Just add your library in front of the code:

```
library(dplyr)
```

comments powered by Disqus

## 2.7 App formats and launching apps

# App formats and launching apps

ADDED: 10 FEB 2015  
BY: WINSTON CHANG

You may have noticed that there are several different ways that Shiny apps are defined and launched. Sometimes you'll see the `shinyServer()` in the `server.R` file, sometimes not, and the same goes for `shinyUI()` in `ui.R`. Sometimes there isn't even a `server.R` file at all.

This article provides an overview of the different ways of defining and launching Shiny applications.

---

## server.R and ui.R

Most examples will include a `server.R` and `ui.R` file like the following:

```
## server.R ##
function(input, output) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs), col = 'darkgray', border = 'white')
  })
}

## ui.R ##
fluidPage(
  sidebarLayout(
    sidebarPanel(
      sliderInput("obs", "Number of observations:", min = 10, max = 500, value = 100)
    ),
    mainPanel(plotOutput("distPlot"))
  )
)
```

For applications defined this way, the `server.R` file must return the server function, and the `ui.R` file must return the UI object (in this case, the UI object is created by `fluidPage()`). In other words, if the files contained other code (like utility functions) you must make sure that the last expression in the file is the server function or UI object.

---

## shinyServer() and shinyUI()

Prior to Shiny 0.10, the `server.R` and `ui.R` files required calls to `shinyServer()` and `shinyUI()` respectively. Older Shiny application examples might look like the following. These are the same as in the previous example, except that the code is wrapped in `shinyServer()` and `shinyUI()`:

```
## server.R ##
shinyServer(function(input, output) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs), col = 'darkgray', border = 'white')
  })
})

## ui.R ##
shinyUI(fluidPage(
  sidebarLayout(
    sidebarPanel(
      sliderInput("obs", "Number of observations:", min = 10, max = 500, value = 100)
    ),
    mainPanel(plotOutput("distPlot"))
  )
))
```

As of Shiny 0.10, calling these functions is no longer needed.

---

## app.R

As of Shiny 0.10.2, applications can be created with a single file, `app.R`, which contains both the UI and server code. This file must return an object created by the `shinyApp()` function.

```
## app.R ##
server <- function(input, output) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs), col = 'darkgray', border = 'white')
  })
}

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      sliderInput("obs", "Number of observations:", min = 10, max = 500, value = 100)
    ),
    mainPanel(plotOutput("distPlot"))
  )
)

shinyApp(ui = ui, server = server)
```

This method is more appropriate for smaller applications; for larger applications, you may find that having separate `ui.R` and `server.R` files makes your code easier to manage.

For more information, see the article about [single-file apps](#).

---

## session and clientData

In the server code for some examples, you might see code like this:

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

In other examples, you might `session` as a third argument to the server function:

```
function(input, output, session) { ... }
```

The `session` argument is optional. It's only needed if you want to use advanced features of Shiny – some functions in Shiny take the `session` variable as an argument.

You may also see some older examples that take `clientId` as an argument to the server function. `clientId` provides information about the connection and the visibility of various components on the web page (see the [client data article](#) for more).

However, it is no longer necessary to use `clientId` as an argument, because if you have `session`, you can access the same information client data with `session$clientId`. For the sake of consistency, we recommend using `session$clientId`:

```
# These two server functions do the same thing

# Using the clientId argument directly (older examples)
function(input, output, clientId) {
  output$txt <- renderPrint({
    clientId
  })
}

# Using the session argument
function(input, output, session) {
  output$txt <- renderPrint({
    session$clientId
  })
}
```

---

## Ways of calling `runApp()`

There are several different things that may be passed to `runApp()` to launch an application.

---

### App directory

If your application resides in a directory `myapp/`, you could launch it with:

```
runApp("myapp")
```

---

### Shiny app object

If you've created a Shiny app object at the console by calling `shinyApp()`, you can pass that app object to `runApp()`:

```
# Create app object (assume ui and server are defined above)
app <- shinyApp(ui, server)

runApp(app)
```



Additionally, if you simply type `app` at the console and press Enter, R will launch the app. This is because when you run code at the console, R will call `print()` on the return value, and for a Shiny app object, the `print()` method calls `runApp()` on the object. So you could do the following to launch the app:

```
app <- shinyApp(ui, server)
app
```

---

## `list(ui, server)`

Another way to launch an app is by giving `runApp()` a list with the ui and server components. This is an older style that predates the Shiny app object method above.

```
# (Assume ui and server are defined above)
runApp(list(ui, server))
```

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

## 2.8 Persistent data storage in Shiny apps

# Persistent data storage in Shiny apps

ADDED: 01 JUL 2015

BY: DEAN ATTALI

Shiny apps often need to save data, either to load it back into a different session or to simply log some information. However, common methods of storing data from R may not work well with Shiny. Functions like `write.csv()` and `saveRDS()` save data locally, but consider how [shinyapps.io](http://shinyapps.io) works.

Shinyapps.io is a popular server for hosting Shiny apps. It is designed to distribute your Shiny app across different servers, which means that if a file is saved during one session on some server, then loading the app again later will probably direct you to a different server where the previously saved file doesn't exist.

On other occasions, you may use data that is too big to store locally with R in an efficient manner.

This guide will explain seven methods for storing persistent data remotely with a Shiny app. You will learn how to store:

- Arbitrary data can be stored as a file in some sort of a file system ([local file system](#), [Dropbox](#), [Amazon S3](#))
- Structured rectangular data can be stored as a table in a relational database or table-storage service ([SQLite](#), [MySQL](#), [Google Sheets](#))
- Semi-structured data can be stored as a collection in a NoSQL database ([MongoDB](#))

The article explains the theory behind each method, and augments the theory with working examples that will make it clear and easy for you to use these methods in your own apps.

As a complement to this article, you can see a [live demo of a Shiny app](#) that uses each of the seven storage methods to save and load data ([source code on GitHub](#)). This article expands on Jeff Allen's [article regarding sharing data across sessions](#).

## Basic Shiny app without data storage

To demonstrate how to store data using each storage type, we'll start with a simple form-submission Shiny app that

1. collects some information from the user
2. stores their response, and
3. shows all previous responses

Initially the app will only save responses within its R session. We will learn later how to modify the app to use each different storage type.

Here is the code for the basic app that we will be using as our starting point—copy it into a file named `app.R`. (In case you didn't know: Shiny apps don't *have to* be broken up into separate `ui.R` and `server.R` files, they can be completely defined in one file as [this Shiny article](#) explains)

```
library(shiny)
```

```

# Define the fields we want to save from the form
fields <- c("name", "used_shiny", "r_num_years")

# Shiny app with 3 fields that the user can submit data for
shinyApp(
  ui = fluidPage(
    DT::dataTableOutput("responses", width = 300), tags$hr(),
    textInput("name", "Name", ""),
    checkboxInput("used_shiny", "I've built a Shiny app in R before", FALSE),
    sliderInput("r_num_years", "Number of years using R", 0, 25, 2, ticks = FALSE),
    actionButton("submit", "Submit")
  ),
  server = function(input, output, session) {

    # Whenever a field is filled, aggregate all form data
    formData <- reactive({
      data <- sapply(fields, function(x) input[[x]])
      data
    })

    # When the Submit button is clicked, save the form data
    observeEvent(input$submit, {
      saveData(formData())
    })

    # Show the previous responses
    # (update with current response when Submit is clicked)
    output$responses <- DT::renderDataTable({
      input$submit
      loadData()
    })
  }
)

```

The above code is taken from a [guide on how to mimic a Google form with Shiny](#).

The above app is very simple—there is a table that shows all responses, three input fields, and a Submit button that will take the data in the input fields and save it. You might notice that there are two functions that are not defined but are used in the app: `saveData(data)` and `loadData()`. These two functions are the only code that affects how the data is stored/retrieved, and we will redefine them for each data storage type. In order to make the app work for now, here's a trivial implementation of the save and load functions that simply stores responses in the current R session.

```

saveData <- function(data) {
  data <- as.data.frame(t(data))
  if (exists("responses")) {
    responses <<- rbind(responses, data)
  } else {
    responses <<- data
  }
}

loadData <- function() {
  if (exists("responses")) {
    responses
  }
}

```

Before continuing further, make sure this basic app works for you and that you understand every line in it—it is not difficult, but take the two minutes to go through it. The code for this app is also available as a [gist](#) and you can run it either by copying all the code to your RStudio IDE or by running `shiny:runGist("c4db11d81f3c46a7c4a5")`.

# Local vs remote storage

Before diving into the different storage methods, one important distinction to understand is *local storage vs remote storage*.

Local storage means saving a file on the same machine that is running the Shiny application. Functions like `write.csv()`, `write.table()`, and `saveRDS()` implement local storage because they will save a file on the machine running the app. Local storage is generally faster than remote storage, but it should only be used if you always have access to the machine that saves the files.

Remote storage means saving data on another server, usually a reliable hosted server such as Dropbox, Amazon, or a hosted database. One big advantage of using hosted remote storage solutions is that they are much more reliable and can generally be more trusted to keep your data alive and not corrupted.

When going through the different storage type options below, keep in mind that if your Shiny app is hosted on [shinyapps.io](#), you will have to use a remote storage method for the time being. RStudio plans to implement persistent storage on [shinyapps.io](#) soon. In the meantime, using local storage is only an option if you're hosting your own Shiny Server, though that comes at the price of having to manage a server and should only be done if you're comfortable with administering a server.

# Persistent data storage methods

Using the above Shiny app, we can store and retrieve responses in many different ways. Here we will go through seven ways to achieve data persistence that can be easily integrated into Shiny apps. For each method, we will explain the method and provide a version of `saveData()` and `loadData()` that implements the method. To use a method as the storage type in the example app, run the app with the appropriate version of `saveData()` and `loadData()`.

As a reminder, you can see all the seven different storage types being used, along with the exact code used, in [this live Shiny app](#).

Here is a summary of the different storage types we will learn to use.

Method	Data type	Local storage	Remote storage	R package
Local file system	Arbitrary data	YES		-
Dropbox	Arbitrary data		YES	<code>rdrop2</code>
Amazon S3	Arbitrary data		YES	<code>RAmazonS3</code>
SQLite	Structured data	YES		<code>RSQLite</code>
MySQL	Structured data	YES	YES	<code>RMySQL</code>
Google Sheets	Structured data		YES	<code>googlesheets</code>
MongoDB	Semi-structured data	YES	YES	<code>rmongodb</code>

## Store arbitrary data in a file

This is the most flexible option to store data since files allow you to store any type of data, whether it is a single value, a big *data.frame*, or any arbitrary data. There are two common cases for using files to store data:

1. you have one file that gets repeatedly overwritten and used by all sessions (like the example in [Jeff Allen's article](#)), or

2. you save a new file every time there is new data

In our case we'll use the latter because we want to save each response as its own file. We can use the former option, but then we would introduce the potential for [race conditions](#) which will overcomplicate the app. A race condition happens when two users submit a response at the exact same time, but since the file cannot deal with multiple edits simultaneously, one user will overwrite the response of the other user.

When saving multiple files, it is important to save each file with a different file name to avoid overwriting files. There are many ways to do this. For example, you can simply use the current timestamp and an *md5 hash* of the data being saved as the file name to ensure that no two form submissions have the same file name.

Arbitrary data can be stored in a file either on the local file system or on remote services such as Dropbox or Amazon S3.

## 1. Local file system (local)

The most trivial way to save data from Shiny is to simply save each response as its own file on the current server. To load the data, we simply load all the files in the output directory. In our specific example, we also want to concatenate all of the data files together into one *data.frame*.

Setup: The only setup required is to create an output directory (responses in this case) and to ensure that the Shiny app has file permissions to read/write in that directory.

Code:

```
outputDir <- "responses"

saveData <- function(data) {
  data <- t(data)
  # Create a unique file name
  fileName <- sprintf("%s_%s.csv", as.integer(Sys.time()), digest::digest(data))
  # Write the file to the local system
  write.csv(
    x = data,
    file = file.path(outputDir, fileName),
    row.names = FALSE, quote = TRUE
  )
}

loadData <- function() {
  # Read all the files into a list
  files <- list.files(outputDir, full.names = TRUE)
  data <- lapply(files, read.csv, stringsAsFactors = FALSE)
  # Concatenate all data together into one data.frame
  data <- do.call(rbind, data)
  data
}
```

## 2. Dropbox (remote)

If you want to store arbitrary files with a remote hosted solution instead of the local file system, you can store files on [Dropbox](#). Dropbox is a file storing service which allows you to host any file, up to a certain maximum usage. The free account provides plenty of storage space and should be enough to store most data from Shiny apps.

This approach is similar to the previous approach that used the local file system. The only difference is that now that files are being saved to and loaded from Dropbox. You can use the [rdrop2](#) package to interact with Dropbox from R. Note that [rdrop2](#) can only move existing files onto Dropbox, so we still need to create a local file before storing it on Dropbox.

Setup: You need to have a Dropbox account and create a folder to store the responses. You will also need to add authentication to `rdrop2` with any approach [suggested in the package README](#). The authentication approach I chose was to authenticate manually once and to copy the resulting `.httr-oauth` file that gets created into the Shiny app's folder.

Code:

```
library(rdrop2)
outputDir <- "responses"

saveData <- function(data) {
  data <- t(data)
  # Create a unique file name
  fileName <- sprintf("%s_%s.csv", as.integer(Sys.time()), digest::digest(data))
  # Write the data to a temporary file locally
  filePath <- file.path(tempdir(), fileName)
  write.csv(data, filePath, row.names = FALSE, quote = TRUE)
  # Upload the file to Dropbox
  drop_upload(filePath, dest = outputDir)
}

loadData <- function() {
  # Read all the files into a list
  filesInfo <- drop_dir(outputDir)
  filePaths <- filesInfo$path
  data <- lapply(filePaths, drop_read_csv, stringsAsFactors = FALSE)
  # Concatenate all data together into one data.frame
  data <- do.call(rbind, data)
  data
}
```

### 3. Amazon S3 (remote)

Another popular alternative to Dropbox for hosting files online is [Amazon S3](#), or S3 in short. Just like with Dropbox, you can host any type of file on S3, but instead of placing files inside directories, in S3 you place files inside of *buckets*. You can use the [RAmazonS3](#) package to interact with S3 from R. Note that the package is a few years old and is not under active development, so use it at your own risk.

Setup: You need to have an [Amazon Web Services](#) account and to create an S3 bucket to store the responses. As the [package documentation](#) explains, you will need to set the `AmazonS3` global option to enable authentication.

Code:

```
library(RAmazonS3)

s3BucketName <- "my-unique-s3-bucket-name"
options(AmazonS3 = c('login' = "secret"))

saveData <- function(data) {
  # Create a unique file name
  fileName <- sprintf("%s_%s.csv", as.integer(Sys.time()), digest::digest(data))
  # Upload the data to S3
  addFile(
    I(paste0(
      paste(names(data), collapse = ", "),
      "\n",
      paste(data, collapse = ", ")
    ))
  )
}
```

```

    )),
    s3BucketName,
    fileName,
    virtual = TRUE
  )
}

loadData <- function() {
  # Get a list of all files
  files <- listBucket(s3BucketName)$Key
  files <- as.character(files)
  # Read all files into a list
  data <- lapply(files, function(x) {
    raw <- getFile(s3BucketName, x, virtual = TRUE)
    read.csv(text = raw, stringsAsFactors = FALSE)
  })
  # Concatenate all data together into one data.frame
  data <- do.call(rbind, data)
  data
}

```

## Store structured data in a table

If the data you want to save is structured and rectangular, storing it in a table would be a good option. Loosely defined, structured data means that each observation has the same fixed fields, and rectangular data means that all observations contain the same number of fields and fit into a nice 2D matrix. A *data.frame* is a great example of such data, and thus *data.frames* are ideal candidates to be stored in tables such as relational databases.

Structured data must have some *schema* that defines what the data fields are. In a *data.frame*, the number and names of the columns can be thought of as the schema. In tables with a header row, the header row can be thought of as the schema.

Structured data can be stored in a table either in a relational database (such as SQLite or MySQL) or in any other table-hosting service such as Google Sheets. If you have experience with database interfaces in other languages, you should note that R does not currently have support for prepared statements, so any SQL statements have to be constructed manually. One advantage of using a relational database is that with most databases it is safe to have multiple users using the database concurrently without running into race conditions thanks to [transaction support](#).

### 4. SQLite (local)

SQLite is a very simple and light-weight relational database that is very easy to set up. SQLite is serverless, which means it stores the database locally on the same machine that is running the shiny app. You can use the [RSQLite](#) package to interact with SQLite from R. To connect to a SQLite database in R, the only information you need to provide is the location of the database file.

To store data in a SQLite database, we loop over all the values we want to add and use a [SQL INSERT](#) statement to add the data to the database. It is essential that the schema of the database matches exactly the names of the columns in the Shiny data, otherwise the SQL statement will fail. To load all previous data, we use a plain [SQL SELECT \\*](#) statement to get all the data from the database table.

Setup: First, you must have SQLite installed on your server. Installation is fairly easy; for example, on an Ubuntu machine you can install SQLite with `sudo apt-get install sqlite3 libsqlite3-dev`. If you use shinyapps.io, SQLite is already installed on the shinyapps.io server, which will be a handy feature in future versions of shinyapps.io, which will include persistent local storage.

You also need to create a database and a table that will store all the responses. When creating the table, you need to set up the schema of the table to match the columns of your data. For example, if you want to save data with

columns "name" and "email" then you can create the SQL table with

```
CREATE TABLE responses(name TEXT, email TEXT);
```

. Make sure the shiny app has write permissions on the database file and its parent directory.

Code:

```
library(RSQLite)
sqlitePath <- "/path/to/sqlite/database"
table <- "responses"

saveData <- function(data) {
  # Connect to the database
  db <- dbConnect(SQLite(), sqlitePath)
  # Construct the update query by looping over the data fields
  query <- sprintf(
    "INSERT INTO %s (%s) VALUES ('%s')",
    table,
    paste(names(data), collapse = ", "),
    paste(data, collapse = "', '")
  )
  # Submit the update query and disconnect
  dbGetQuery(db, query)
  dbDisconnect(db)
}

loadData <- function() {
  # Connect to the database
  db <- dbConnect(SQLite(), sqlitePath)
  # Construct the fetching query
  query <- sprintf("SELECT * FROM %s", table)
  # Submit the fetch query and disconnect
  data <- dbGetQuery(db, query)
  dbDisconnect(db)
  data
}
```

## 5. MySQL (local or remote)

MySQL is a very popular relational database that is similar to SQLite but is more powerful. MySQL databases can either be hosted locally (on the same machine as the Shiny app) or online using a hosting service.

This method is very similar to the previous SQLite method, with the main difference being where the database is hosted. You can use the [RMySQL](#) package to interact with MySQL from R. Since MySQL databases can be hosted on remote servers, the command to connect to the server involves more parameters, but the rest of the saving/loading code is identical to the SQLite approach. To connect to a MySQL database, you need to provide the following parameters: host, port, dbname, user, password.

Setup: You need to create a MySQL database (either locally or using a web service that hosts MySQL databases) and a table that will store the responses. As with the setup for SQLite, you need to make sure the table schema is properly set up for your intended data.

Code:

```
library(RMySQL)

options(mysql = list(
  "host" = "127.0.0.1",
```



```

"port" = 3306,
"user" = "myuser",
"password" = "mypassword"
))
databaseName <- "myshinydatabase"
table <- "responses"

saveData <- function(data) {
  # Connect to the database
  db <- dbConnect(MySQL(), dbname = databaseName, host = options()$mysql$host,
    port = options()$mysql$port, user = options()$mysql$user,
    password = options()$mysql$password)
  # Construct the update query by looping over the data fields
  query <- sprintf(
    "INSERT INTO %s (%s) VALUES ('%s')",
    table,
    paste(names(data), collapse = ", "),
    paste(data, collapse = "', '")
  )
  # Submit the update query and disconnect
  dbGetQuery(db, query)
  dbDisconnect(db)
}

loadData <- function() {
  # Connect to the database
  db <- dbConnect(MySQL(), dbname = databaseName, host = options()$mysql$host,
    port = options()$mysql$port, user = options()$mysql$user,
    password = options()$mysql$password)
  # Construct the fetching query
  query <- sprintf("SELECT * FROM %s", table)
  # Submit the fetch query and disconnect
  data <- dbGetQuery(db, query)
  dbDisconnect(db)
  data
}

```

## 6. Google Sheets (remote)

If you don't want to deal with the formality and rigidity of a database, another option for storing tabular data is in a Google Sheet. One nice advantage of Google Sheets is that they are easy to access from anywhere; but unlike with databases, with Google Sheets data can be overwritten with multiple concurrent users.

You can use the `googlesheets` package to interact with Google Sheets from R. To connect to a specific sheet, you will need either the sheet's title or key (preferably key, as it is unique). It is very easy to store or retrieve data from a Google Sheet, as the code below shows.

Setup: All you need to do is create a Google Sheet and set the top row with the names of the fields. You can do that either via a web browser or by using the `googlesheets` package. You also need to have a Google account. The `googlesheets` package uses a similar approach to authentication as `rdrop2`, and thus you also need to authenticate in a similar fashion, such as by copying a valid `.httr-oauth` file to your Shiny directory.

Code:

```

library(googlesheets)

table <- "responses"

```

```

saveData <- function(data) {
  # Grab the Google Sheet
  sheet <- gs_title(table)
  # Add the data as a new row
  gs_add_row(sheet, input = data)
}

loadData <- function() {
  # Grab the Google Sheet
  sheet <- gs_title(table)
  # Read the data
  gs_read_csv(sheet)
}

```

## Store semi-structured data in a NoSQL database

If you have data that is not fully structured but is also not completely free-form, a good middle ground can be using a NoSQL database. NoSQL databases can also be referred to as schemaless databases because they do not use a formal schema. NoSQL databases still offer some of the benefits of a traditional relational database, but are more flexible because every entry can use different fields. If your Shiny app needs to store data that has several fields but there is no unifying schema for all of the data to use, then using a NoSQL database can be a good option.

There are many NoSQL databases available, but here we will only show how to use mongoDB.

### 7. MongoDB (local or remote)

MongoDB is one of the most popular NoSQL databases, and just like MySQL it can be hosted either locally or remotely. There are many web services that offer mongoDB hosting, including [MongoLab](#) which gives you free mongoDB databases. In mongoDB, entries (in our case, responses) are stored in a *collection* (the equivalent of an S3 bucket or a SQL table).

You can use either the `rmongodb` or `mongolite` package to interact with mongoDB from R. In this example we will use `rmongodb`, but `mongolite` is a perfectly good alternative. As with the relational database methods, all we need to do in order to save/load data is connect to the database and submit the equivalent of an update or select query. To connect to the database you need to provide the following: db, host, username, password. When saving data to mongoDB, the data needs to be converted to BSON (binary JSON) in order to be inserted into a mongoDB collection. MongoDB automatically adds a unique “id” field to every entry, so when retrieving data, we manually remove that field.

Setup: All you need to do is create a mongoDB database—either locally or using a web service such as MongoLab. Since there is no schema, it is not mandatory to create a collection before populating it.

Code:

```

library(rmongobd)

options(mongodb = list(
  "host" = "ds012345.mongolab.com:61631",
  "username" = "myuser",
  "password" = "mypassword"
))
databaseName <- "myshinydatabase"
collectionName <- "myshinydatabase.responses"

saveData <- function(data) {
  # Connect to the database

```

```

db <- mongo.create(db = databaseName, host = options()$mongodb$host,
  username = options()$mongodb$username, password = options()$mongodb$password)
# Convert the data to BSON (Binary JSON)
data <- mongo.bson.from.list(as.list(data))
# Insert the data into the mongo collection and disconnect
mongo.insert(db, collectionName, data)
mongo.disconnect(db)
}

loadData <- function() {
  # Connect to the database
  db <- mongo.create(db = databaseName, host = options()$mongodb$host,
    username = options()$mongodb$username, password = options()$mongodb$password)
  # Get a list of all entries
  data <- mongo.find.all(db, collectionName)
  # Read all entries into a list
  data <- lapply(data, data.frame, stringsAsFactors = FALSE)
  # Concatenate all data together into one data.frame
  data <- do.call(rbind, data)
  # Remove the ID variable
  data <- data[, -1, drop = FALSE]
  # Disconnect
  mongo.disconnect(db)
  data
}

```

## Conclusion

Persistent storage lets you do more with your Shiny apps. You can even use persistent storage to access and write to remote data sets that would otherwise be too big to manipulate in R.

The following table can serve as a reminder of the different storage types and when to use them. Remember that any method that uses local storage can only be used on Shiny Server, while any method that uses remote storage can be also used on shinyapps.io.

Method	Data type	Local storage	Remote storage	R package
Local file system	Arbitrary data	YES		-
Dropbox	Arbitrary data		YES	rdrop2
Amazon S3	Arbitrary data		YES	RAmazonS3
SQLite	Structured data	YES		RSQLite
MySQL	Structured data	YES	YES	RMySQL
Google Sheets	Structured data		YES	googlesheets
MongoDB	Semi-structured data	YES	YES	rmongodb

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

MarkeDAtHome

---

We can also add to this list Google's BigQuery, via Hadley's <https://github.com/hadley/bigr...> or within dplyr()

Ludovic Kuty

---

Great article ! It proved really useful to explore a few things and in particular to build custom-made questionnaires for live statistical lessons

Ben Porter

---

Another clear and thorough post from Dean Attali.

comments powered by Disqus

## 2.9 Application layout guide

# Application layout guide

ADDED: 24 JAN 2014

BY: JJ ALLAIRE

## Overview

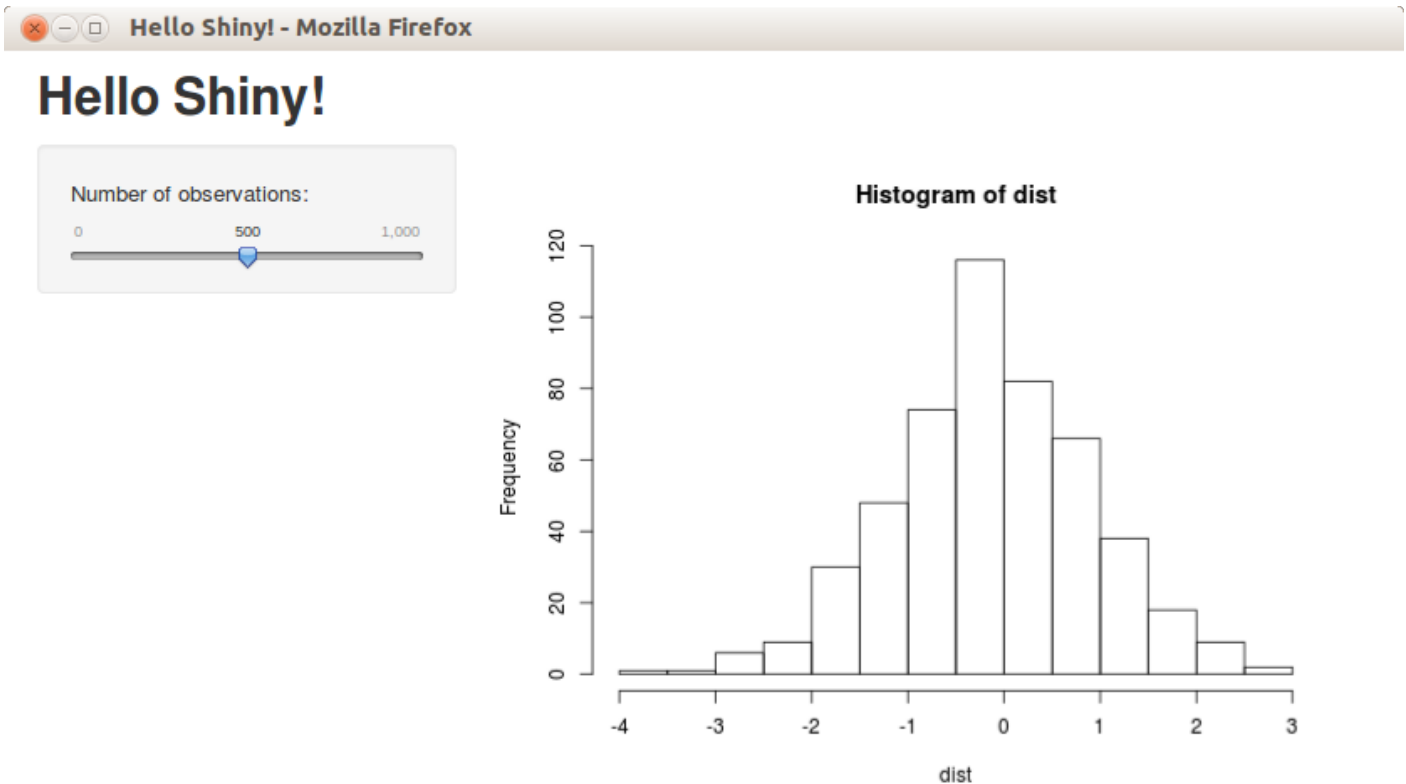
Shiny includes a number of facilities for laying out the components of an application. This guide describes the following application layout features:

1. The simple default layout with a sidebar for inputs and a large main area for output.
2. Custom application layouts using the Shiny grid layout system.
3. Segmenting layouts using the `tabsetPanel()` and `navlistPanel()` functions.
4. Creating applications with multiple top-level components using the `navbarPage()` function.

These features were implemented using the layout features available in [Bootstrap 2](#), an extremely popular HTML/CSS framework (though no prior experience with Bootstrap is assumed).

## Sidebar Layout

The sidebar layout is a useful starting point for most applications. This layout provides a sidebar for inputs and a large main area for output:



Here's the code used to create this layout:

```
shinyUI(fluidPage(
  titlePanel("Hello Shiny!"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("obs", "Number of observations:",
        min = 1, max = 1000, value = 500)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
))
```

Note that the sidebar can be positioned to the left (the default) or right of the main area. For example, to position the sidebar to the right you would use this code:

```
sidebarLayout(position = "right",
  sidebarPanel(
    # Inputs excluded for brevity
  ),
  mainPanel(
    # Outputs excluded for brevity
  )
)
```

## Grid Layout

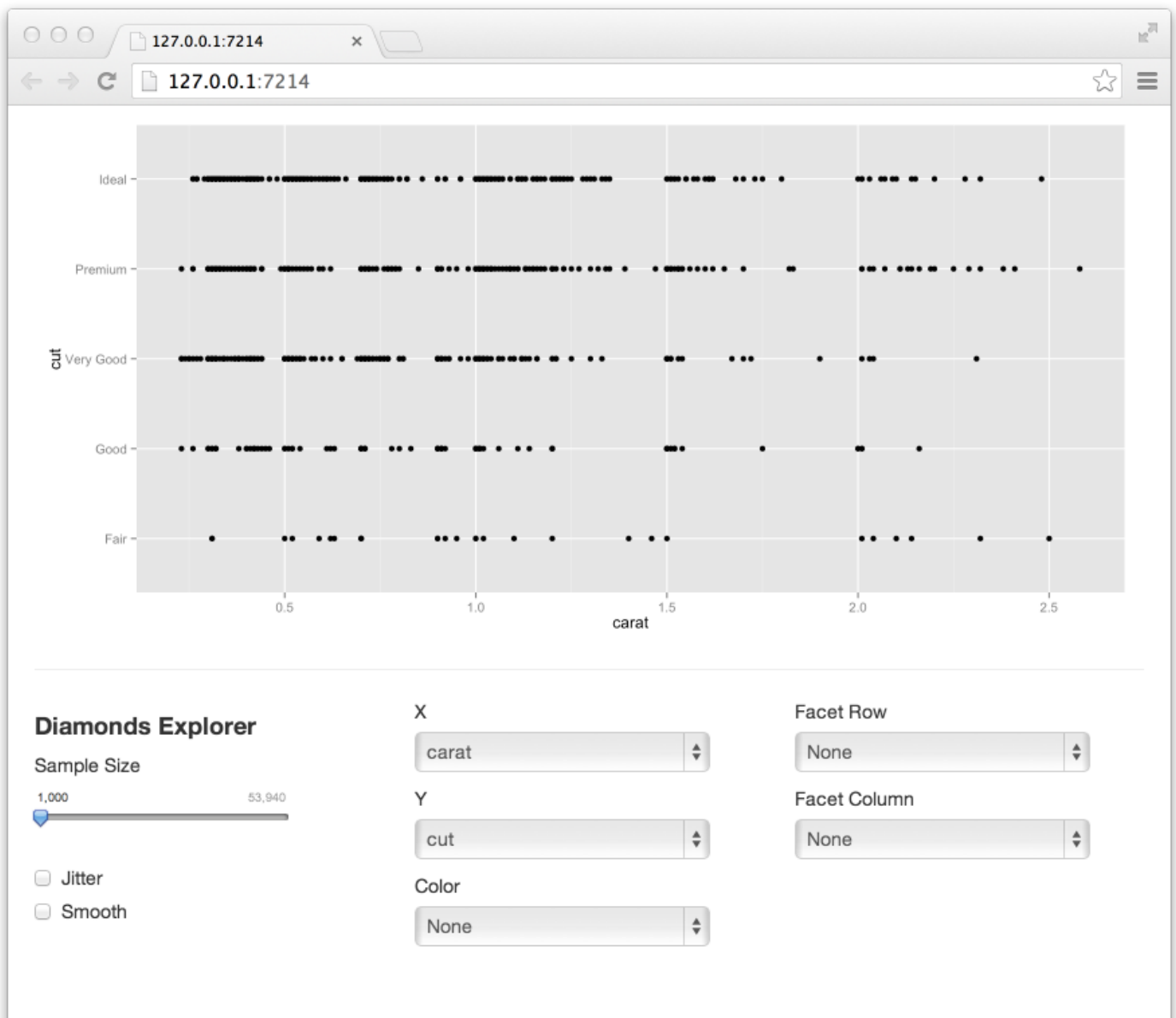
The familiar `sidebarLayout()` described above makes use of Shiny's lower-level grid layout functions. Rows are created by the `fluidRow()` function and include columns defined by the `column()` function. Column widths are based on the Bootstrap 12-wide grid system, so should add up to 12 within a `fluidRow()` container.

To illustrate, here's the sidebar layout implemented using the `fluidRow()`, `column()` and `wellPanel()` functions:

```
shinyUI(fluidPage(  
  titlePanel("Hello Shiny!"),  
  fluidRow(  
    column(4,  
      wellPanel(  
        sliderInput("obs", "Number of observations:",  
                    min = 1, max = 1000, value = 500)  
      )  
    ),  
    column(8,  
      plotOutput("distPlot")  
    )  
  )  
))
```

The first parameter to the `column()` function is its width (out of a total of 12 columns). It's also possible to offset the position of columns to achieve more precise control over the location of UI elements. You can move columns to the right by adding the `offset` parameter to the `column()` function. Each unit of offset increases the left-margin of a column by a whole column.

Here's an example of a UI with a plot at the top and three columns at the bottom that contain the inputs that drive the plot:



The code required to implement this UI is as follows:

```
library(shiny)
library(ggplot2)

dataset <- diamonds

shinyUI(fluidPage(

  title = "Diamonds Explorer",

  plotOutput('plot'),

  hr(),

  fluidRow(
    column(3,
      h4("Diamonds Explorer"),
      sliderInput('sampleSize', 'Sample Size',
        min=1, max=nrow(dataset), value=min(1000, nrow(dataset)),
        step=500, round=0),
```



```
    br(),
    checkboxInput('jitter', 'Jitter'),
    checkboxInput('smooth', 'Smooth')
  ),
  column(4, offset = 1,
    selectInput('x', 'X', names(dataset)),
    selectInput('y', 'Y', names(dataset), names(dataset)[[2]]),
    selectInput('color', 'Color', c('None', names(dataset)))
  ),
  column(4,
    selectInput('facet_row', 'Facet Row', c(None='.', names(dataset))),
    selectInput('facet_col', 'Facet Column', c(None='.', names(dataset)))
  )
)
))
```

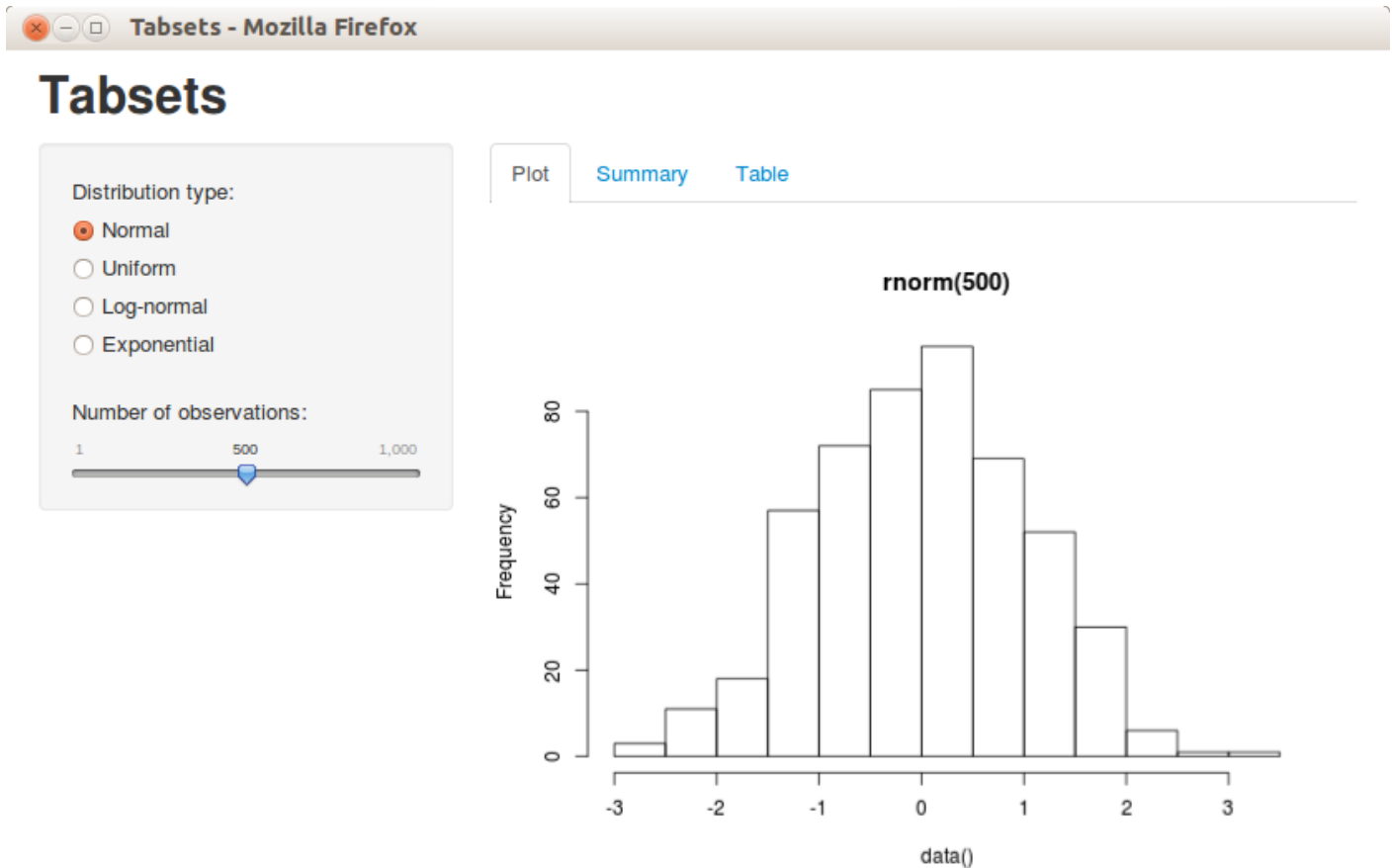
There are a few important things to note here:

1. The inputs are at the bottom and broken into three columns of varying widths.
2. The `offset` parameter is used on the center input column to provide custom spacing between the first and second columns.
3. The page doesn't include a `titlePanel()` so the title is specified as an explicit argument to `fluidPage()`.

Grid layouts can be used anywhere within a `fluidPage()` and can even be nested within each other. You can find out more about grid layouts in the [Grid Layouts in Depth](#) section below.

## Tabsets

Often applications need to subdivide their user-interface into discrete sections. This can be accomplished using the `tabsetPanel()` function. For example:



The code required to create this UI is:

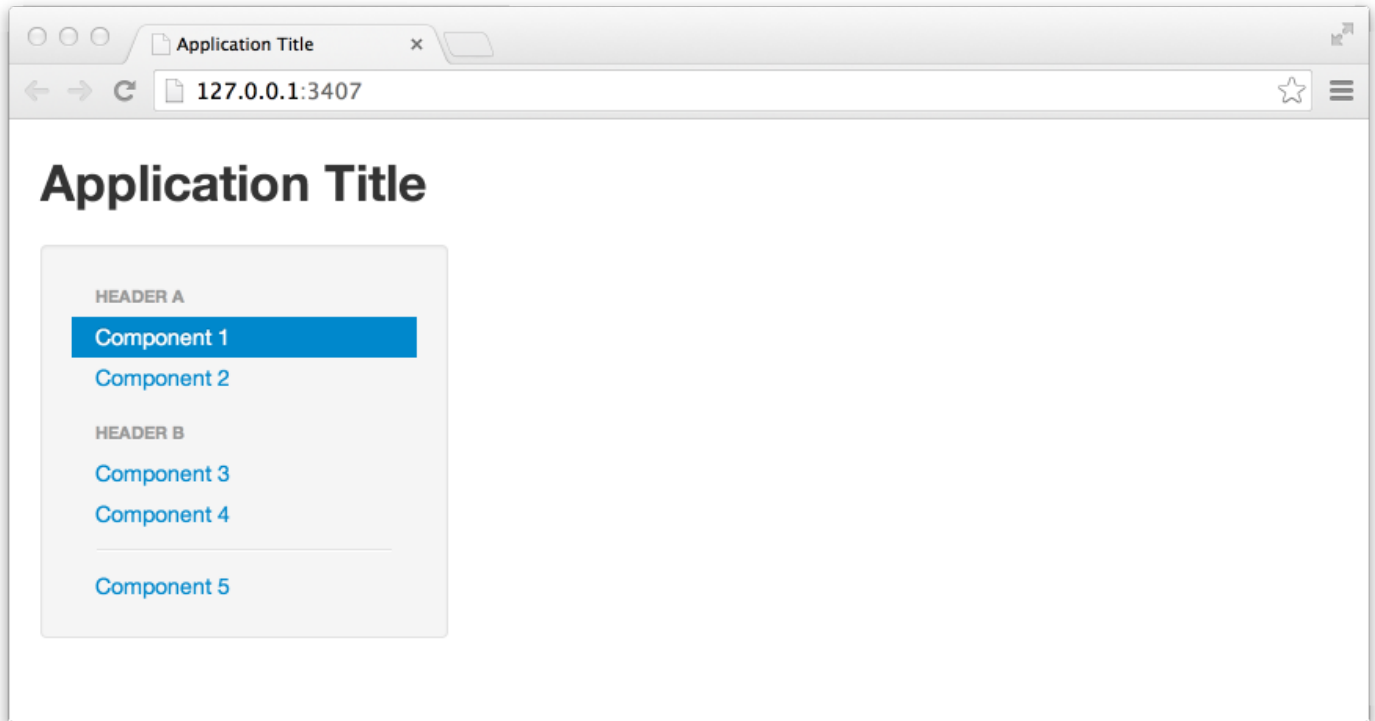
```
shinyUI(fluidPage(
  titlePanel("Tabsets"),
  sidebarLayout(
    sidebarPanel(
      # Inputs excluded for brevity
    ),
    mainPanel(
      tabsetPanel(
        tabPanel("Plot", plotOutput("plot")),
        tabPanel("Summary", verbatimTextOutput("summary")),
        tabPanel("Table", tableOutput("table"))
      )
    )
  )
))
```

Tabs can be located above (the default), below, left, or to the right of tab content. For example, to position the tabs below the tab content you would use this code:

```
tabsetPanel(position = "below",
  tabPanel("Plot", plotOutput("plot")),
  tabPanel("Summary", verbatimTextOutput("summary")),
  tabPanel("Table", tableOutput("table"))
```

## Navlists

When you have more than a handful of `tabPanels` the `navlistPanel()` may be a good alternative to `tabsetPanel()`. A navlist presents the various components as a sidebar list rather than using tabs. It also supports section heading and separators for longer lists. Here's an example of a `navlistPanel()`:

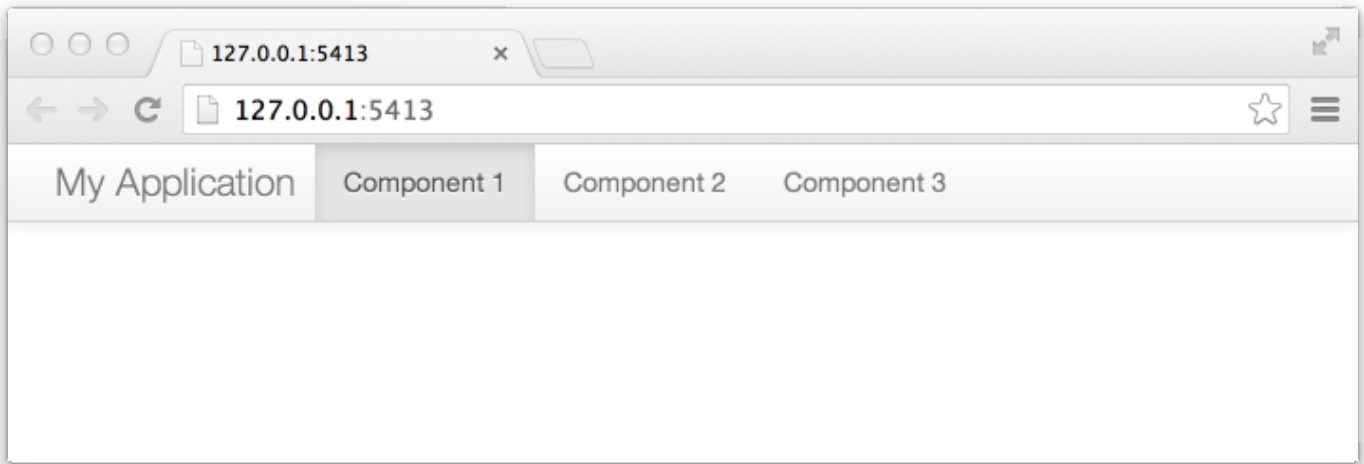


The code required to implement this is as follows (note that the `tabPanels` are empty to keep the example uncluttered, typically they'd include additional UI elements):

```
shinyUI(fluidPage(
  titlePanel("Application Title"),
  navlistPanel(
    "Header A",
    tabPanel("Component 1"),
    tabPanel("Component 2"),
    "Header B",
    tabPanel("Component 3"),
    tabPanel("Component 4"),
    "-----",
    tabPanel("Component 5")
  )
))
```

## Navbar Pages

You may want to create a Shiny application that consists of multiple distinct sub-components (each with their own sidebar, tabsets, or other layout constructs). The `navbarPage()` function creates an application with a standard Bootstrap [Navbar](#) at the top. For example:

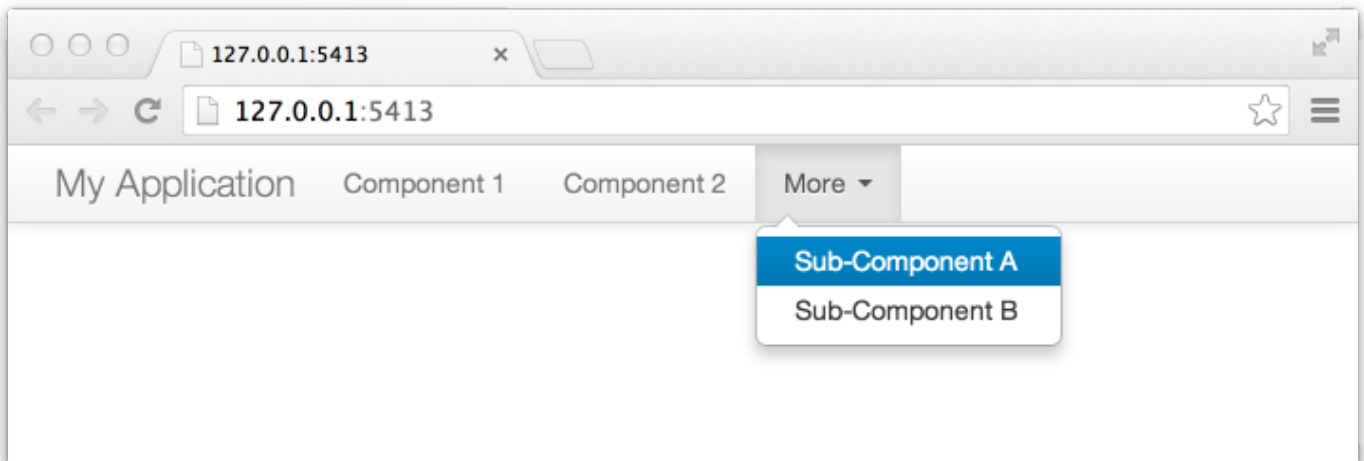


```
shinyUI(navbarPage("My Application",
  tabPanel("Component 1"),
  tabPanel("Component 2"),
  tabPanel("Component 3")
))
```

Note that the Shiny `tabPanel()` is used to specify the navigable components.

## Secondary Navigation

You can add a second level of navigation to the page by using the `navbarMenu()` function. This adds a menu to the top level navbar which can in turn refer to additional `tabPanels`.



```
shinyUI(navbarPage("My Application",
  tabPanel("Component 1"),
  tabPanel("Component 2"),
  navbarMenu("More",
    tabPanel("Sub-Component A"),
    tabPanel("Sub-Component B"))
))
```

## Additional Options

There are several other arguments to `navbarPage()` that provide additional measures of customization:

Argument Description

header	Tag or list of tags to display as a common header above all tabPanels.
footer	Tag or list of tags to display as a common footer below all tabPanels
inverse	<b>TRUE</b> to use a dark background and light text for the navigation bar
collapsible	<b>TRUE</b> to automatically collapse the navigation elements into a menu when the width of the browser is less than 940 pixels (useful for viewing on smaller touchscreen device)

## Grid Layouts in Depth

There are two types of Bootstrap grids, fluid and fixed. The examples so far have used the fluid grid system exclusively and that's the system that's recommended for most applications (and the default for Shiny functions like `navbarPage()` and `sidebarLayout()`).

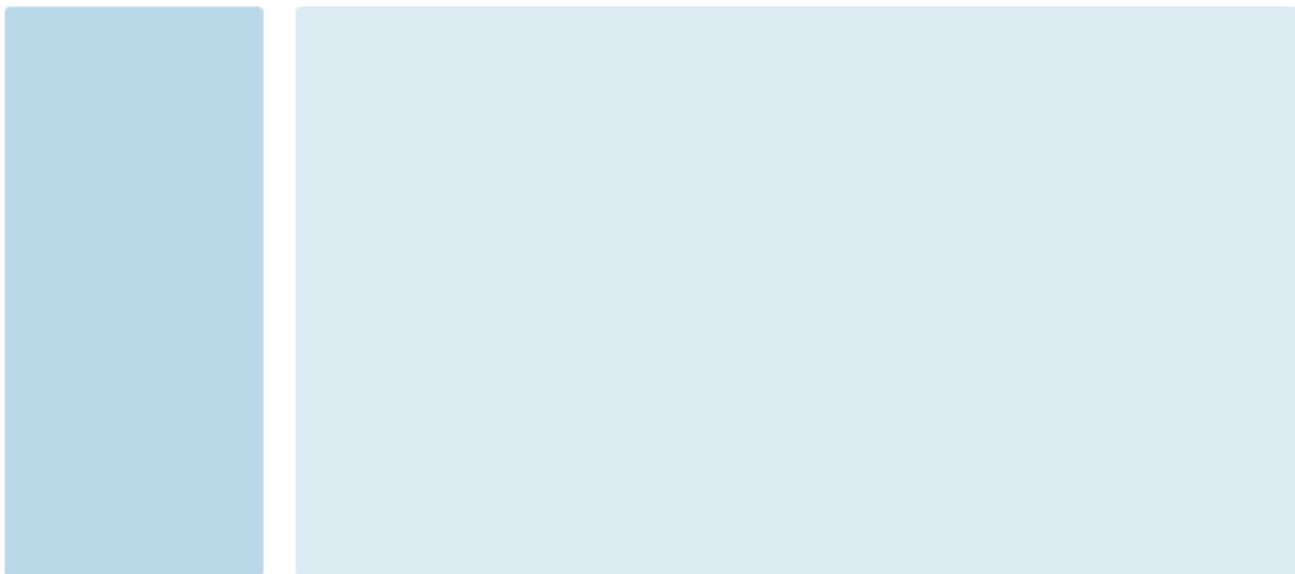
Both grid systems use a flexibly sub-dividable 12-column grid for layout. The fluid system always occupies the full width of the web page and re-sizes it's components dynamically as the size of the page changes. The fixed system occupies a fixed width of 940 pixels by default and may assume other widths when Bootstrap responsive layout kicks in (e.g. when on a tablet).

The following sections are a translation of the official [Bootstrap 2 grid system](#) documentation, with HTML code replaced by R code.

## Fluid Grid System

The Bootstrap grid system utilizes 12 columns which can be flexibly subdivided into rows and columns. To create a layout based on the fluid system you use the `fluidPage()` function. To create rows within the grid you use the `fluidRow()` function; to create columns within rows you use the `column()` function.

For example, consider this high level page layout (the numbers displayed are columns out of a total of 12):



To create this layout in a Shiny application you'd use the following code (note that the column widths within the fluid row add up to 12):

```
shinyUI(fluidPage(
  fluidRow(
    column(2,
      "sidebar"
    ),
    column(10,
      "main"
    )
  )
)
```

))

## Column Offsetting

It's also possible to offset the position of columns to achieve more precise control over the location of UI elements. Move columns to the right by adding the `offset` parameter to the `column()` function. Each unit of offset increases the left-margin of a column by a whole column. Consider this layout:

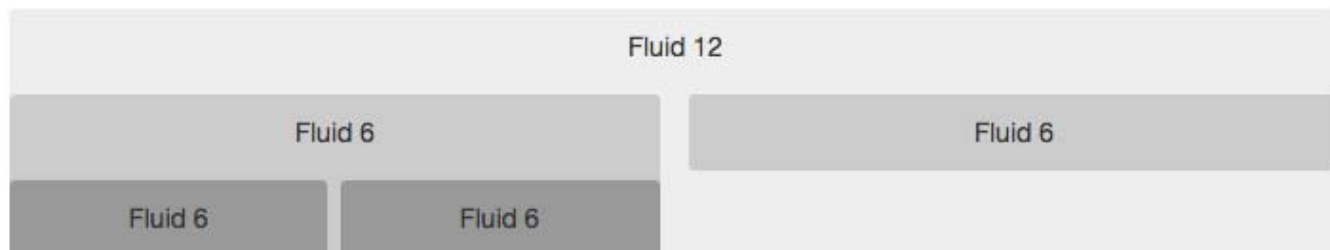


To create this layout in a Shiny application you'd use the following code:

```
shinyUI(fluidPage(
  fluidRow(
    column(4,
      "4"
    ),
    column(4, offset = 4,
      "4 offset 4"
    )
  ),
  fluidRow(
    column(3, offset = 3,
      "3 offset 3"
    ),
    column(3, offset = 3,
      "3 offset 3"
    )
  )
))
```

## Column Nesting

When you nest columns within a fluid grid, each nested level of columns should add up to 12 columns. This is because the fluid grid uses percentages, not pixels, for setting widths. Consider this page layout:



To create this layout in a Shiny application you'd use the following code:

```
shinyUI(fluidPage(
  fluidRow(
    column(12,
      "Fluid 12",
      fluidRow(
        column(6,
          "Fluid 6",

```

```

    fluidRow(
      column(6,
        "Fluid 6"),
      column(6,
        "Fluid 6")
    )
  ),
  column(width = 6,
    "Fluid 6")
)
)
)
))

```

Note that each time a `fluidRow()` is introduced the columns within the row add up to 12.

## Fixed Grid System

The fixed grid system also utilizes 12 columns, and maintains a fixed width of 940 pixels by default. If Bootstrap responsive features are enabled (they are by default in Shiny) then the grid will also adapt to be 724px or 1170px wide depending on your viewport (e.g. when on a tablet).

The main benefit of a fixed grid is that it provides stronger guarantees about how users will see the various elements of your UI laid out (this is because it's not being dynamically laid out according to the width of the browser). The main drawback is that it's a bit more complex to work with. In general we recommend using fluid grids unless you absolutely require the lower level layout control afforded by a fixed grid.

### Using Fixed Grids

Using fixed grids in Shiny works almost identically to fluid grids. Here are the differences to keep in mind:

1. You use the `fixedPage()` and `fixedRow()` functions to build the grid.
2. Rows can nest, but should always include a set of columns that add up to the number of columns of their parent (rather than resetting to 12 at each nesting level as they do in fluid grids).

Here's the code for a fixed grid version of the simple sidebar layout shown earlier:

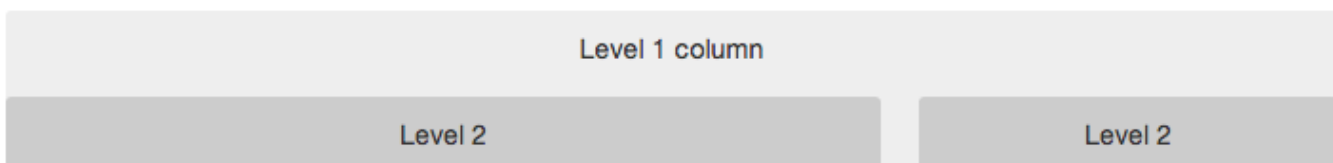
```

shinyUI(fixedPage(
  fixedRow(
    column(2,
      "sidebar"
    ),
    column(10,
      "main"
    )
  )
)
)
))

```

### Column Nesting

In fixed grids the width of each nested column must add up to the number of columns in their parent. Here's a `fixedRow()` with a 9-wide column that contains two other columns of width 6 and 3:



The create this row within a Shiny application you'd use the following code:

```
fixedRow(
  column(9,
    "Level 1 column",
    fixedRow(
      column(6,
        "Level 2"
      ),
      column(3,
        "Level 2"
      )
    )
  )
)
```

Note that the total size of the nested columns is 9, the same as their parent column.

## Responsive Layout

The Bootstrap grid system supports responsive CSS, which enables your application to automatically adapt its layout for viewing on different sized devices. Responsive layout includes the following:

1. Modifying the width of columns in the grid
2. Stack elements instead of float wherever necessary
3. Resize headings and text to be more appropriate for devices

Responsive layout is enabled by default for all Shiny page types. To disable responsive layout you should pass `responsive = FALSE` to the `fluidPage()` or `fixedPage()` function.

### Supported Devices

When responsive layout is enabled here is how the Bootstrap grid system adapts to various devices:

	Layout width	Column width	Gutter width
Large display	1200px and up	70px	30px
Default	980px and up	60px	20px
Portrait tablets	768px and above	42px	20px
Phones to tablets	767px and below	Fluid (no fixed widths)	Fluid (no fixed widths)
Phones	480px and below	Fluid (no fixed widths)	Fluid (no fixed widths)

Note that on smaller screen sizes fluid columns widths are used automatically even if the page uses fixed grid layout.

## Application Themes

Shiny applications inherit the default visual theme of the Bootstrap web framework upon which Shiny is based. If you want to change the look of your application it's possible to specify an alternate Bootstrap theme. You can do this using the `theme` parameter to the `fluidPage()`, `fixedPage()`, or `navbarPage()` function, which specifies an alternative Bootstrap CSS stylesheet to use for the application.

Bootstrap themes are typically specified using a single CSS source file (although it's possible for them to have associated images, css, or fonts as well). If you've saved a theme at the location `www/bootstrap.css` within your application directory then you would link it in using this code:

```
shinyUI(fluidPage(theme = "bootstrap.css",
  titlePanel("My Application")
```



When importing a theme it's important to make sure that it's compatible with Bootstrap 3. One popular source of Bootstrap themes is [Bootswatch](#), but there are many others.

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

aquacalc

Thanks for the well organized and very useful summary.

Ian

hi,I work with shiny ,thanks for your documents . I have trouble when i want display few charts in one page. for example i have 12 plot in same page,3 rows and 4 columnsâ€now I just try fluidrow and box ,failed!!  
(R code " par(mfrow=c(3,4))",it's ok ,but charts not great )

Dave

This is great, thanks. Quick question: how do I add additional content to a single tab? So could I have plotOutput and verbatimTextOutput appear within a single tab?

hi im weasel

Im trying to get the FluidRow Column Nesting code to work so I can play with it to understand what's going on. I'm talking about:

comments powered by Disqus

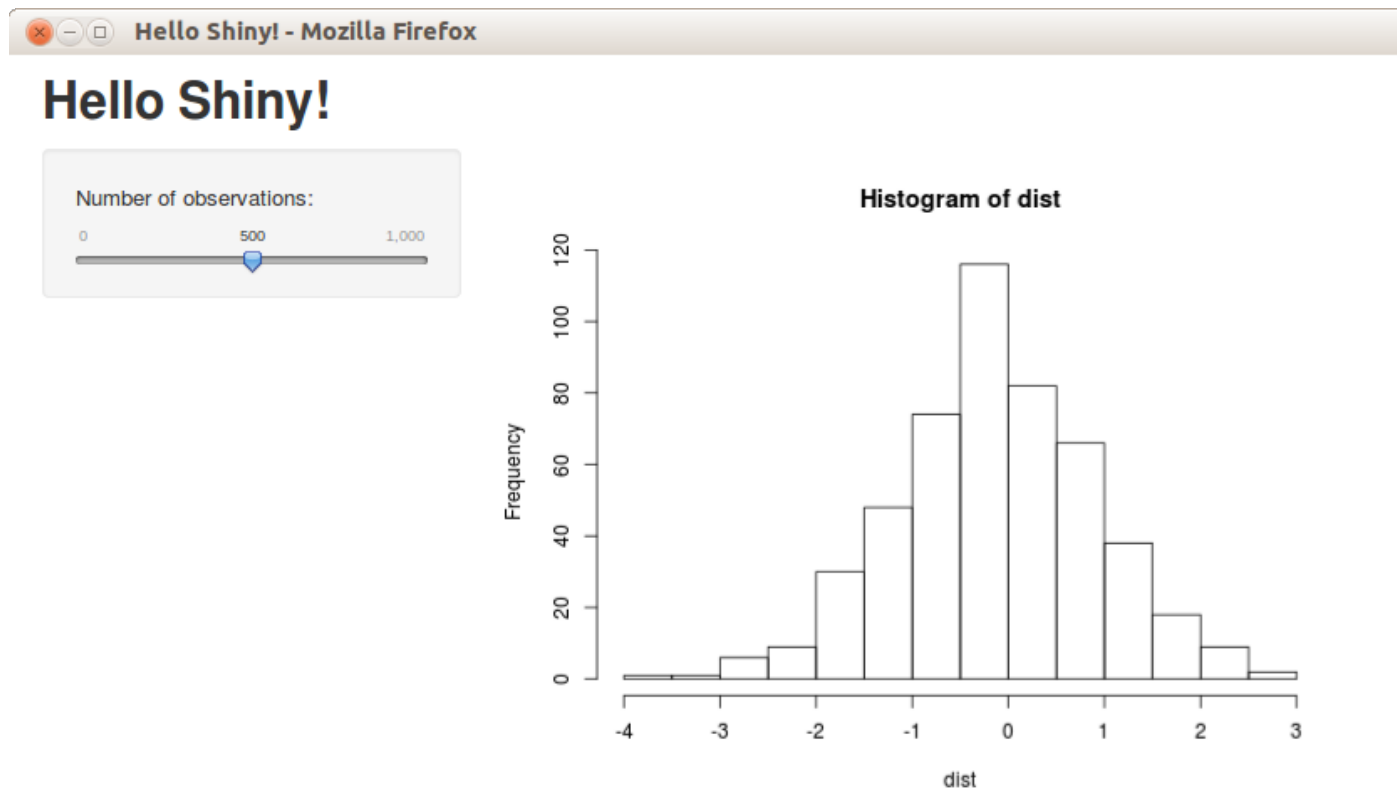
## 2.10 Display modes

# Display modes

ADDED: 03 FEB 2014

BY: GARRETT GROLEMUND

Shiny can display your apps in two different ways. Your app can appear in normal display mode, as pictured below.



Or your app can appear in showcase mode. Showcase mode displays your app alongside the code that generates it; showcase mode also displays a title, author and description for your app.

The screenshot shows a web browser window displaying a Shiny application. The browser's address bar shows the URL `127.0.0.1:3361`. The application title is "Hello Shiny!".

The application interface includes:

- A slider control labeled "Number of observations:" with a value of 500, ranging from 1 to 1,000.
- A histogram titled "Histogram of dist" showing the frequency distribution of a random variable. The x-axis is labeled "dist" and ranges from -3 to 3. The y-axis is labeled "Frequency" and ranges from 0 to 100. The histogram shows a bell-shaped curve centered at 0.
- A code editor at the bottom with two tabs: "server.R" and "ui.R". The "server.R" tab is active, showing the following R code:
 

```
library(shiny)
# Define server logic required to generate and plot a random
# distribution
shinyServer(function(input, output) {
  # Expression that generates a plot of the distribution. 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$distPlot <- renderPlot({
    # generate an rnorm distribution and plot it
    dist <- rnorm(input$obs)
    hist(dist)
  })
})
```
- A button labeled "show with app" next to the code editor.

Below the code editor, there is a small text block: "Hello Shiny! by RStudio, Inc. This small Shiny application demonstrates Shiny's automatic UI updates. Move the Number of observations slider and notice how the `renderPlot` expression is automatically re-evaluated when its dependant, `input$obs`, changes, causing a new distribution to be generated and the plot to be rendered." A "Code license: GPL-3" link is also visible.

To view an app in showcase mode, launch it with the argument `display.mode = "showcase"`, e.g.

```
> runApp("MyApp", display.mode = "showcase")
```

Apps can be set to open in showcase mode by default. If you would prefer to view such an app in normal mode, use the argument `display.mode = "normal"`.

```
> runApp("MyApp", display.mode = "showcase")
```

Shiny's built in example apps will automatically open in showcase mode when you call `runExample`, e.g.

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

## Showcase mode features

When you display your app in showcase mode, Shiny presents the app along with

- the R files in the app's directory, placed in a shared tabset that your user can place next to the app, or below it. These files will always contain server.R and ui.R.
- a title
- an author
- a license
- explanatory text
- code highlighting

## Code highlighting

Shiny showcase will highlight lines of code in server.R as it runs them. The highlight will appear in yellow and fade out after a few moments. This helps reveal how Shiny creates reactivity; when your user manipulates an app, Shiny reruns parts of server.R to create updated output.

The screenshot shows a web browser window displaying a Shiny application titled "Hello Shiny!". The application features a slider control for "Number of observations" with a value of 379. To the right is a histogram titled "Histogram of dist" showing a normal distribution. Below the histogram is a code editor showing the server.R script with syntax highlighting. The code includes a `renderPlot` call that generates a random normal distribution and plots it. A footer section explains that the application demonstrates automatic UI updates based on reactive dependencies.

**Hello Shiny!**

Number of observations: 1 379 1,000

**Histogram of dist**

Frequency

dist

```
server.R  ui.R  show below
library(shiny)
# Define server logic required to generate and plot a random
# distribution
shinyServer(function(input, output) {
  # Expression that generates a plot of the distribution. 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$distPlot <- renderPlot({
    # generate an rnorm distribution and plot it
    dist <- rnorm(input$obs)
    hist(dist)
  })
})
```

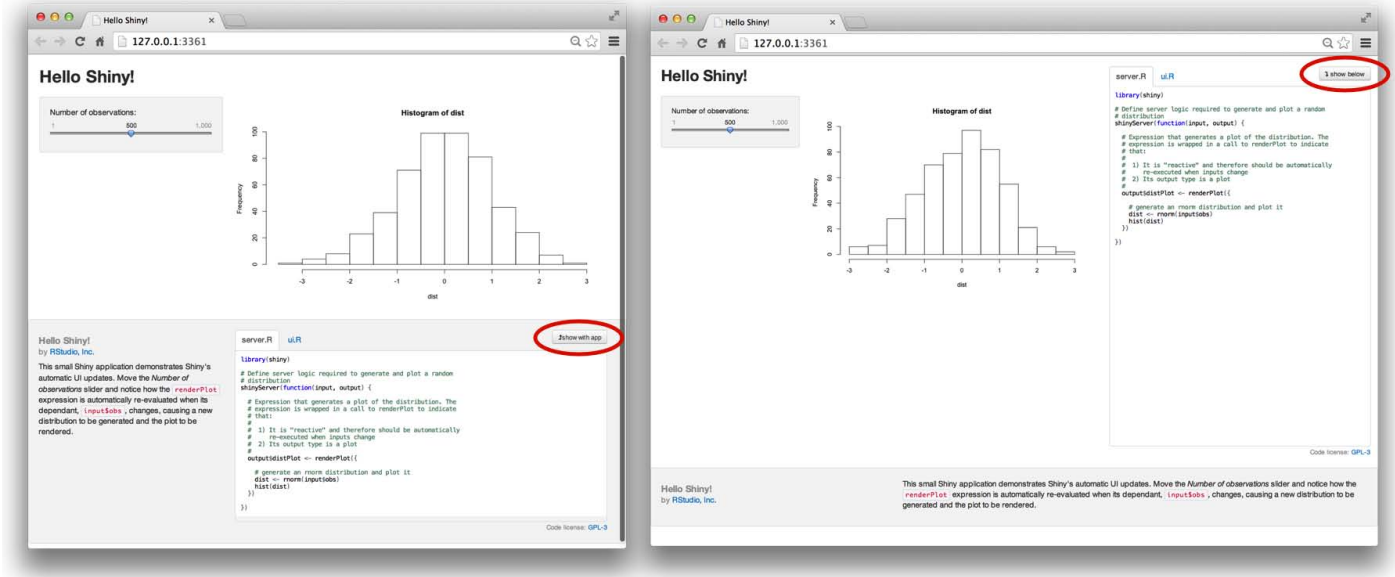
Code license: GPL-3

Hello Shiny!  
by RStudio, Inc.

This small Shiny application demonstrates Shiny's automatic UI updates. Move the *Number of observations* slider and notice how the `renderPlot` expression is automatically re-evaluated when its dependant, `input$obs`, changes, causing a new distribution to be generated and the plot to be rendered.

## Showcase layout

Once an app is open, you can change its layout the buttons labelled `show with app` and `show below`. These will place the app's R scripts either next to the app or below it. The app will automatically scale to fit nicely with the code in your browser window.



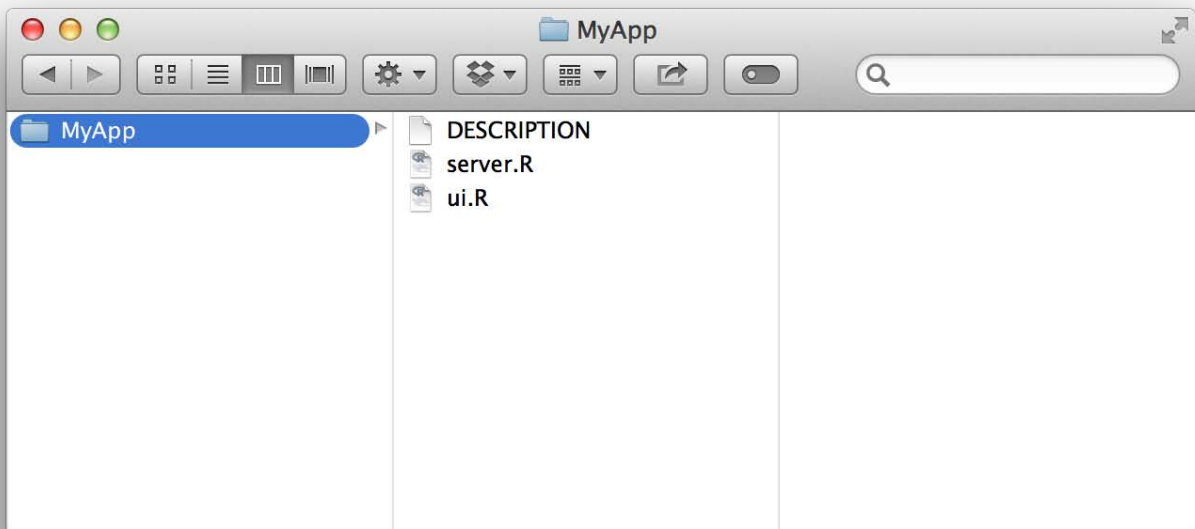
## Writing for Showcase mode

You can provide information about your app that Shiny showcase will use by creating a DESCRIPTION file. The file should be written in plain text and contain Title, Author, and DisplayMode fields in [Debian Control File \(DCF\)](#) format. You can also include other optional fields, such as AuthorUrl, License, and Tags. The description file of Shiny's built in 01\_hello example is displayed below

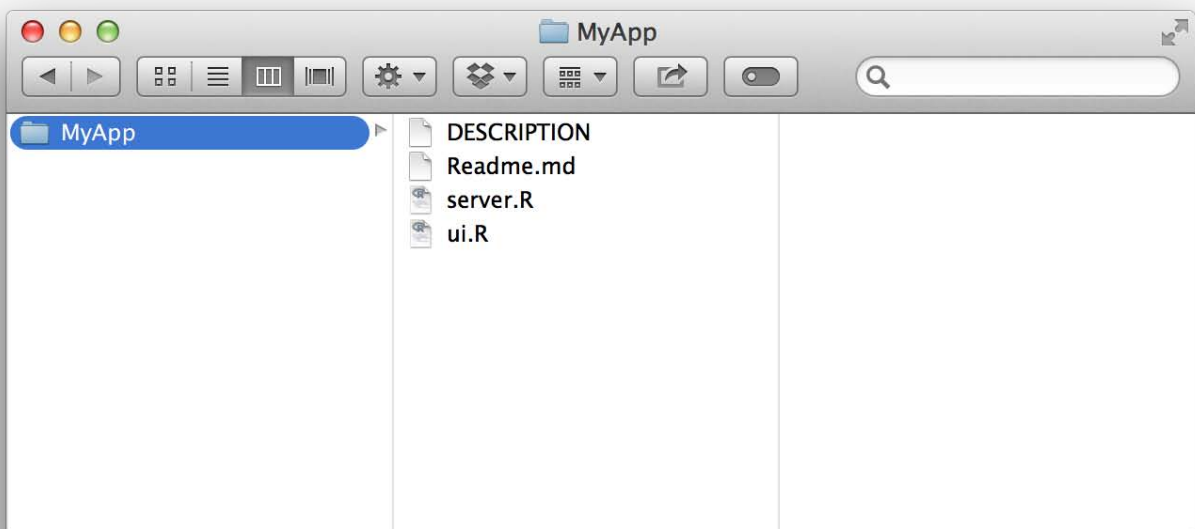
```
Title: Hello Shiny!
Author: RStudio, Inc.
AuthorUrl: http://www.rstudio.com/
License: GPL-3
DisplayMode: Showcase
Tags: getting-started
Type: Shiny
```

Shiny will use the DisplayMode field to determine the default display mode for your app. If you set the field to Showcase, Shiny will open your app in showcase mode. If you set it to Normal, Shiny will open your app in Normal mode. Your users can override this default by using the `display.mode` argument of `runApp`.

Once you've written your DESCRIPTION file, place it alongside the server.R and ui.R files in your app's directory.



You can also create a readme file for your app. Shiny will display the text of the readme beneath the app in showcase mode. Write your readme in markdown and save it in your app directory as `Readme.md`. Shiny will automatically use any `DESCRIPTION` and `Readme.md` files that you place in your app.



Here's an example of the short readme for `01_hello`.

This small Shiny application demonstrates Shiny's automatic UI updates. Move the `*Number of observations*` slider and notice how the ``renderPlot`` expression is automatically re-evaluated when its dependant, ``input$obs``, changes, causing a new distribution to be generated and the plot to be rendered.

# Showcase and privacy

Some developers would prefer not to expose their code with showcase mode. That's not a problem. Your users will not be able to turn on showcase mode unless you let them. It is impossible to force an app into Showcase mode unless (a) you manually launch the app in showcase mode, or (b) the DESCRIPTION file explicitly states that the app should be shown in showcase mode.

However, it is possible for users to turn off showcase mode if they do not like it. A user can turn off showcase mode for an app if they add `?showcase=0` to the end of the app's URL. This won't affect how other users see the app.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

anchitkolla

---

how do you put the display mode equal to showcase on [shinyapps.io??](#)

Ezriah

---

You need to create a text file with the filename DESCRIPTION and no file extension. This file should have the following in it (from above). I don't think you actually need it all, but DO include the line DisplayMode: Showcase. Save the file in the same directory as your shiny application and when you publish it, it publishes in showcase mode.

Title: Hello Shiny!

Author: RStudio, Inc.

AuthorUrl: <http://www.rstudio.com/>

License: GPL-3

DisplayMode: Showcase

Tags: getting-started

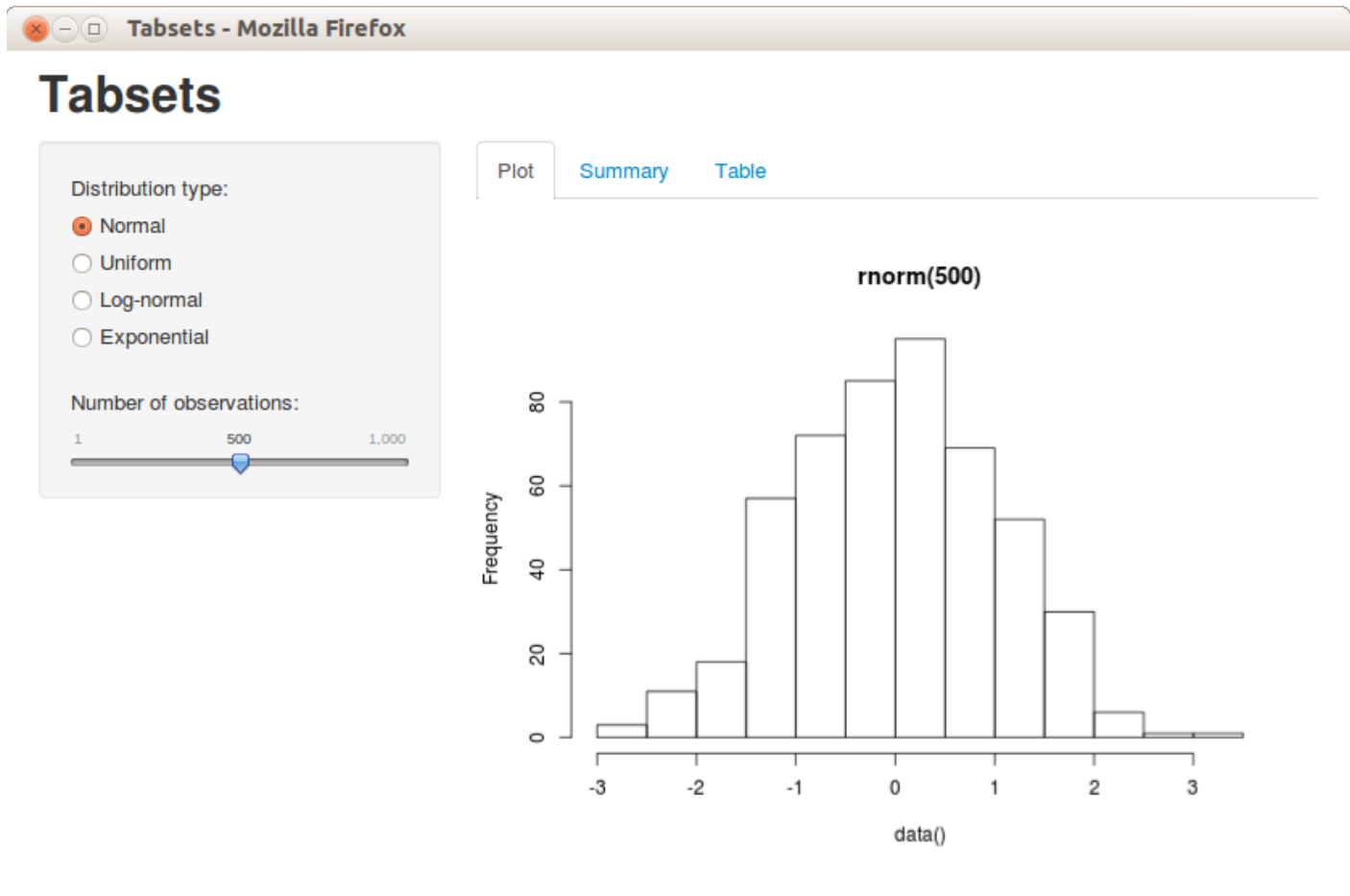
Type: Shiny

comments powered by [Disqus](#)

## 2.11 Tabsets

# Tabsets

ADDED: 06 JAN 2014



The Tabsets application demonstrates using tabs to organize output. To run the example type:

```
> library(shiny)
> runExample("06_tabsets")
```

## Tab Panels

Tabsets are created by calling the `tabsetPanel` function with a list of tabs created by the `tabPanel` function. Each tab panel is provided a list of output elements which are rendered vertically within the tab.

In this example we updated our Hello Shiny application to add a summary and table view of the data, each rendered on their own tab. Here is the revised source code for the user-interface:

```
ui.R
```



```

library(shiny)

# Define UI for random distribution application
shinyUI(pageWithSidebar(

  # Application title
  headerPanel("Tabsets"),

  # Sidebar with controls to select the random distribution type
  # and number of observations to generate. Note the use of the br()
  # element to introduce extra vertical spacing
  sidebarPanel(
    radioButtons("dist", "Distribution type:",
      list("Normal" = "norm",
           "Uniform" = "unif",
           "Log-normal" = "lnorm",
           "Exponential" = "exp")),
    br(),

    sliderInput("n",
      "Number of observations:",
      value = 500,
      min = 1,
      max = 1000)
  ),

  # Show a tabset that includes a plot, summary, and table view
  # of the generated distribution
  mainPanel(
    tabsetPanel(
      tabPanel("Plot", plotOutput("plot")),
      tabPanel("Summary", verbatimTextOutput("summary")),
      tabPanel("Table", tableOutput("table"))
    )
  )
))

```

## Tabs and Reactive Data

Introducing tabs into our user-interface underlines the importance of creating reactive expressions for shared data. In this example each tab provides its own view of the dataset. If the dataset is expensive to compute then our user-interface might be quite slow to render. The server script below demonstrates how to calculate the data once in a reactive expression and have the result be shared by all of the output tabs:

server.R

```

library(shiny)

# Define server logic for random distribution application
shinyServer(function(input, output) {

  # Reactive expression to generate the requested distribution. This is
  # called whenever the inputs change. The renderers defined
  # below then all use the value computed from this expression
  data <- reactive({
    dist <- switch(input$dist,

```

```
norm = rnorm,
unif = runif,
lnorm = rlnorm,
exp = rexp,
rnorm)

dist(input$n)
})

# Generate a plot of the data. Also uses the inputs to build the
# plot label. Note that the dependencies on both the inputs and
# the 'data' reactive expression are both tracked, and all expressions
# are called in the sequence implied by the dependency graph
output$plot <- renderPlot({
  dist <- input$dist
  n <- input$n

  hist(data(),
        main=paste('r', dist, '(', n, ')', sep=''))
})

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

# Generate an HTML table view of the data
output$table <- renderTable({
  data.frame(x=data())
})
})
```

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Phillip Manning

---

Is there a way to hide the scroll bars in a tabset? I am using a tabset to switch between two charts. The charts fit into the tabset but scroll bars still appear. Any suggestions?

Garrett

Philip, this would be a great question for the [Shiny Discuss list](#) with a reproducible example. As far as I know, scroll bars are a feature of your browser, not the Shiny tabset.

comments powered by Disqus

Shiny is an RStudio project. © 2014 RStudio, Inc.

## 2.12 Customize your UI with HTML

# Customize your UI with HTML

ADDED: 16 APR 2014

BY: GARRETT GROLEMUND WITH JOE CHENG

In this article, you will learn how to supplement the functions in `ui.R` with raw HTML to create highly customized Shiny apps. You do not need to know HTML to use Shiny, but if you do, you can use the methods in this article to enhance your app.

The user-interface (UI) of a Shiny app is web document. Shiny developers can provide this document as an `index.html` file or assemble it from R code in a `ui.R` file.

`ui.R` calls R functions that output HTML code. Shiny turns this code into a web app.

I will use the `01_hello` app throughout this article as an example. You can access this app by running:

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

## shinyUI

Many Shiny apps come with a `ui.R` script that determines the layout of the app. The `ui.R` script for `01_example` looks like the following code:

```
library(shiny)

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

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

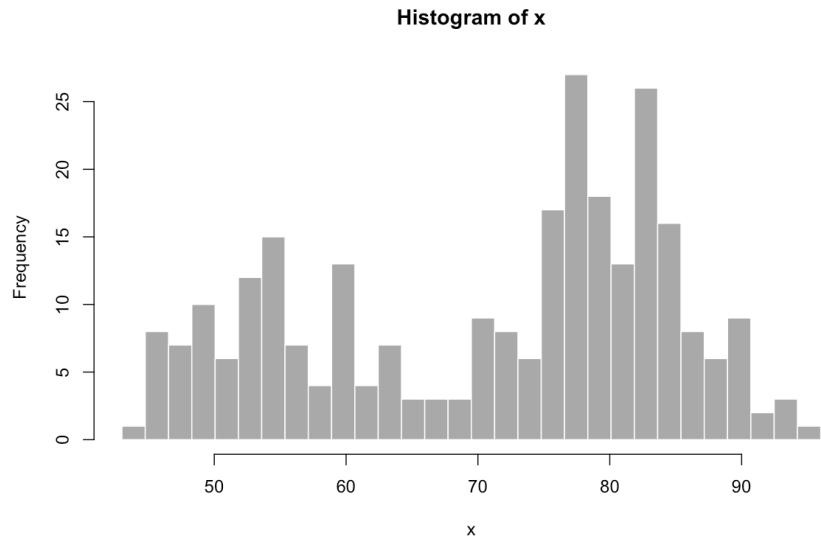
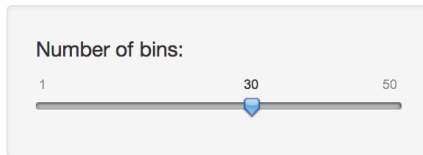
  # Sidebar with a slider input for the number of bins
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),

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

```
)
))
```

The code creates this app:

## Hello Shiny!



Each `ui.R` script calls the function `shinyUI`. `shinyUI` then calls R functions that return HTML. In other words, Shiny lets you generate HTML with R. This is why you do not need to know HTML to use Shiny.

You can see that the functions inside of `shinyUI` return HTML if you run them. `fluidPage` returns a chunk of HTML as does every function inside of `fluidPage`. For example, the following code returns the HTML output in the comments below.

```
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")
    )
  )
)

## <div class="container-fluid">
```

```
## <nz style="padding: 10px 0px;">Hello Shiny!</nz>
## <div class="row-fluid">
## <div class="span4">
## <form class="well">
## <div>
## <label class="control-label" for="bins">Number of bins:</label>
## <input id="bins" type="slider" name="bins" value="30" class="jslider" data-##
from="1" data-to="50" data-step="1" data-skin="plastic" data-round="FALSE## " data-local-e=
"us" data-format="#,##0.#####" data-smooth="FALSE"/>
## </div>
## </form>
## </div>
## <div class="span8">
## <div id="distPlot" class="shiny-plot-output" style="width: 100% ; height: ## 400p
x"></div>
## </div>
## </div>
## </div>
```

```
titlePanel("Hello Shiny!")
## <h2 style="padding: 10px 0px;">Hello Shiny!</h2>
```

In R terminology, the output is a list of character strings with a special class that tells Shiny the contents contain HTML.

```
class(titlePanel("Hello Shiny!"))
## [1] "shiny.tag.list" "list"
```

Shiny's UI functions are sufficient for creating most Shiny apps. In 90% of your Shiny apps, you will probably never think of using anything more complicated. However in some apps, you may want to add custom HTML that is not provided by the usual Shiny functions. You can do this by passing HTML tags to `shinyUI` with the `tags` object.

## tags

`shiny:tags` is a list of 110 functions. Each function builds a specific HTML tag. If you are familiar with HTML, you will recognize these tags by their names. You can learn what the most common tags do in the [Shiny HTML tags glossary](#).

```
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. To create a `div` tag, you can run:

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

You can call some of the most popular tags with helper functions (that wrap the appropriate `tags` functions). For example, the helper function `code` calls the `tags$code` and creates text formatted as computer code. The helper functions that can call their equivalent tags without using the tag syntax (`tags$`) 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$` syntax. For example, to embed a plug-in or third party application call it with `tags$embed`.

Every tag function will treat its arguments in a special way: it will treat named arguments as HTML *attributes* and unnamed arguments as HTML *children*.

## Attributes

A tag function will use each named argument to add an HTML *attribute* to the tag. The argument name becomes the attribute name, and the argument value becomes the attribute value. So for example, if you want to create a `div` with a `class` attribute, use:

```
tags$div(class = "header")
## <div class="header"></div>
```

To add an attribute without a value, set the attribute to `NA`:

```
tags$div(class = "header", checked = NA)
## <div class="header" checked></div>
```

## Children

Each tag function will add unnamed arguments to your tag as HTML *children*. This addition lets you nest tags inside of each other (just as in HTML).

```
tags$div(class = "header", checked = NA,
  tags$p("Ready to take the Shiny tutorial? If so"),
  tags$a(href = "shiny.rstudio.com/tutorial", "Click Here!"))
## <div class="header" checked>
## <p>Ready to take the Shiny tutorial? If so</p>
## <a href="shiny.rstudio.com/tutorial">Click Here!</a>
## </div>
```

## withTags

You can save typing by wrapping your HTML objects with `withTags`. `withTags` is similar to R's regular `with`

function. R will lookup each tag function mentioned inside `withTags` in the `tags` object, even if you do not specify `tags$`.

```
withTags({
  div(class="header", checked=NA,
    p("Ready to take the Shiny tutorial? If so"),
    a(href="shiny.rstudio.com/tutorial", "Click Here!")
  )
})
## <div class="header" checked>
##   <p>Ready to take the Shiny tutorial? If so</p>
##   <a href="shiny.rstudio.com/tutorial">Click Here!</a>
## </div>
```

Once you have a complete tag, you can add it directly to your app's `shinyUI` function. For example, you could add the tag above to the `ui.R` file of `01-hello`:

```
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),

      # adding the new div tag to the sidebar
      tags$div(class="header", checked=NA,
        tags$p("Ready to take the Shiny tutorial? If so"),
        tags$a(href="shiny.rstudio.com/tutorial", "Click Here!")
      )
    ),

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

Your updated app will contain the new HTML element.



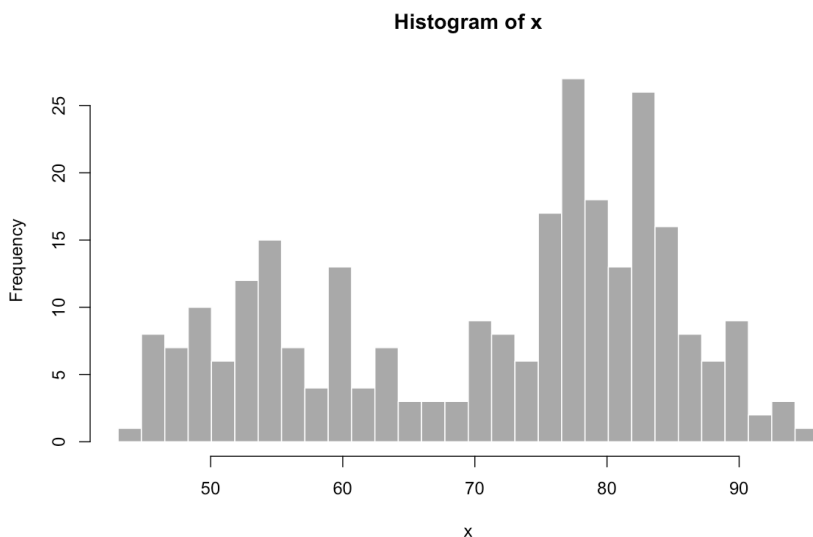
## Hello Shiny!

Number of bins:

1 30 50

Ready to take the Shiny tutorial? If so

[Click Here!](#)



## Conditional attributes and children

If you set an argument of a tag function to `NULL`, the argument will not appear in the HTML output. `NULL` gives you a way to build attributes and children that will appear only under certain conditions.

```
tags$div(class = "header", id = NULL,
         NULL,
         "line 2"
        )
## <div class="header">line 2</div>
```

```
tags$div(class = "header", id = if (FALSE) 100,
         if (FALSE) "line 1",
         "line 2"
        )
## <div class="header">line 2</div>
```

## Lists

You can pass a list of children to a tag with R's list function. The tag function will add each element of the list as a child of the tag.

```
tags$div(class="header", checked=NA,
         list(
           tags$sp("Ready to take the Shiny tutorial? If so"),
           tags$a(href="shiny.rstudio.com/tutorial", "Click Here!"),
           "Thank you"
         )
        )
## <div class="header" checked>
## <p>Ready to take the Shiny tutorial? If so</p>
## <a href="shiny.rstudio.com/tutorial">Click Here!</a>
## Thank you
```

```
## </div>
```

## Raw HTML

You cannot put raw HTML directly into a tag object or into `shinyUI`. Shiny will treat raw HTML as a character string, adding HTML as text to your UI document.

```
tags$div(
  "<strong>Raw HTML!</strong>"
)
## <div>&lt;strong&gt;Raw HTML!&lt;/strong&gt;</div>
```

To add raw HTML, use the `HTML` function. `HTML` takes a character string and returns it as HTML (a special class of object in Shiny).

```
tags$div(
  HTML("<strong>Raw HTML!</strong>")
)
## <div><strong>Raw HTML!</strong></div>
```

Shiny will assume that the code you pass to `HTML` is correctly written HTML. Be sure to double check it.

## Warning

It is a bad idea to pass an input object to `HTML`:

```
tags$div(
  HTML(input$text)
)
```

This allows the user to add their own HTML to your app, which creates a security vulnerability. What you user enters could be added to the web document or seen by other users, which might break the app. In the worse case scenario, a user may try to deploy malicious [Cross Site Scripting \(XSS\)](#), an undesirable security vulnerability.

## Recap

You can use `HTML` to customize your Shiny apps. Every Shiny app is built on an HTML document that creates the apps' user interface. Usually, Shiny developers create this document by giving `shinyUI` R functions that build HTML output. However, you can supply HTML output directly with Shiny's `tags` object.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Linus

---

I'm have some HTML code that I'm calling from within Shiny using `includeHTML(...)`. It's a simple HTML page with a text bar and a button perfectly aligned (I don't want to recreate this in Shiny, although I know how to do that).

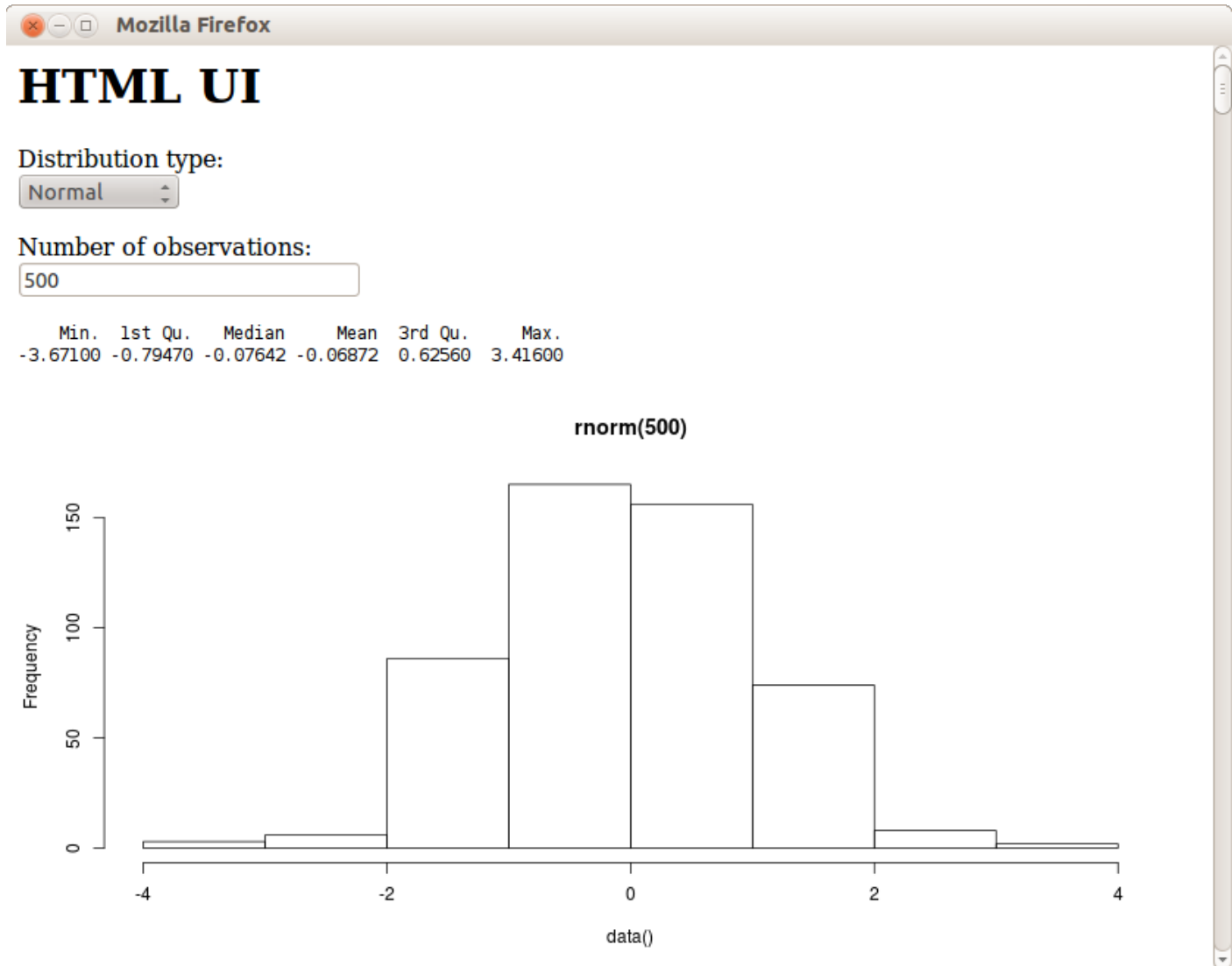
Now I'd like to add reactive functions attached to the button using the text in the box. How can I reference the pure HTML widgets from with Shiny? Any help will be much appreciated.

comments powered by [Disqus](#)

## 2.13 Build your entire UI with HTML

# Build your entire UI with HTML

ADDED: 06 JAN 2014



The HTML UI application demonstrates defining a Shiny user-interface using a standard HTML page rather than a ui.R script. To run the example type:

```
> library(shiny)
> runExample("08_html ")
```

## Defining an HTML UI

Many Shiny apps use a ui.R file to build their user-interfaces. While this is a fast and convenient way to build user-

interfaces, some applications will inevitably require more flexibility. For this type of application, you can define your user-interface directly in HTML. In this case there is no ui.R file and the directory structure looks like this:

```
<application-dir>
|-- www
    |-- index.html
|-- server.R
```

In this example we re-write the front-end of the Tabsets application ( `runExample("06_tabsets")` ) using HTML directly. Here is the source code for the new user-interface definition:

www/index.html

```
<html >

<head>
  <script src="shared/jquery.js" type="text/javascript"></script>
  <script src="shared/shiny.js" type="text/javascript"></script>
  <link rel="stylesheet" type="text/css" href="shared/shiny.css"/>
</head>

<body>
  <h1>HTML UI</h1>

  <p>
    <label>Distribution type:</label><br />
    <select name="dist">
      <option value="norm">Normal</option>
      <option value="uni f">Uniform</option>
      <option value="lnorm">Log-normal</option>
      <option value="exp">Exponential</option>
    </select>
  </p>

  <p>
    <label>Number of observations:</label><br />
    <input type="number" name="n" value="500" min="1" max="1000" />
  </p>

  <pre id="summary" class="shiny-text-output"></pre>

  <div id="plot" class="shiny-plot-output"
    style="width: 100%; height: 400px"></div>

  <div id="table" class="shiny-html-output"></div>
</body>

</html >
```

There are few things to point out regarding how Shiny binds HTML elements back to inputs and outputs:

- HTML form elements (in this case a select list and a number input) are bound to input slots using their `name` attribute.
- Output is rendered into HTML elements based on matching their `id` attribute to an output slot and by specifying the requisite CSS class for the element (in this case either `shiny-text-output`, `shiny-plot-output`, or `shiny-html-output`).

With this technique you can create highly customized user-interfaces using whatever HTML, CSS, and JavaScript you like.

## Server Script

All of the changes from the original Tabsets application were to the user-interface, the server script remains the same:

server.R

```
library(shiny)

# Define server logic for random distribution application
shinyServer(function(input, output) {

  # Reactive expression to generate the requested distribution. This is
  # called whenever the inputs change. The output renderers defined
  # below then all used the value computed from this expression
  data <- reactive({
    dist <- switch(input$dist,
                  norm = rnorm,
                  unif = runif,
                  lnorm = rlnorm,
                  exp = rexp,
                  rnorm)

    dist(input$n)
  })

  # Generate a plot of the data. Also uses the inputs to build the
  # plot label. Note that the dependencies on both the inputs and
  # the data reactive expression are both tracked, and all expressions
  # are called in the sequence implied by the dependency graph
  output$plot <- renderPlot({
    dist <- input$dist
    n <- input$n

    hist(data(),
         main=paste('r', dist, '(', n, ')', sep=''))
  })

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

  # Generate an HTML table view of the data
  output$table <- renderTable({
    data.frame(x=data())
  })
})
```

you'd like to get specific help, we recommend the [Shiny Discussion Forum](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Eddy F

---

It's common web dev best practice defer loading of JavaScript blocks by a) using async/defer attributes (modern approach) or by placing JavaScript Script blocks within the Body just prior to the <body> tag (slightly older best practice). Does this cause any issues with the shiny.js file?

Nirmalya

---

I am hosting a website. In one page I want to take input from the user process it online using shiny and reflect the graph back on the webpage. How to do it? Do I have to put both of my file index.html and server.R in my hosting server space then it will start working ( it is not working I tried). Can you guide me on this.

Guest

---

What would be the recommended way to have a standard shiny input slider widget when creating the entire UI with html?

comments powered by [Disqus](#)

## 2.14 Build a dynamic UI that reacts to user

# Build a dynamic UI that reacts to user input

ADDED: 06 JAN 2014

## Dynamic UI

Shiny apps are often more than just a fixed set of controls that affect a fixed set of outputs. Inputs may need to be shown or hidden depending on the state of another input, or input controls may need to be created on-the-fly in response to user input.

Shiny currently has three different approaches you can use to make your interfaces more dynamic. From easiest to most difficult, they are:

- The `conditionalPanel` function, which is used in `ui.R` and wraps a set of UI elements that need to be dynamically shown/hidden
- The `renderUI` function, which is used in `server.R` in conjunction with the `htmlOutput` function in `ui.R`, lets you generate calls to UI functions and make the results appear in a predetermined place in the UI
- Use JavaScript to modify the webpage directly.

Let's take a closer look at each approach.

## Showing and Hiding Controls With `conditionalPanel`

`conditionalPanel` creates a panel that shows and hides its contents depending on the value of a JavaScript expression. Even if you don't know any JavaScript, simple comparison or equality operations are extremely easy to do, as they look a lot like R (and many other programming languages).

Here's an example for adding an optional smoother to a ggplot, and choosing its smoothing method:

```
# Partial example
checkboxInput("smooth", "Smooth"),
conditionalPanel (
  condition = "input.smooth == true",
  selectInput("smoothMethod", "Method",
    list("lm", "glm", "gam", "loess", "rlm"))
)
```

In this example, the select control for `smoothMethod` will appear only when the `smooth` checkbox is checked. Its condition is `"input.smooth == true"`, which is a JavaScript expression that will be evaluated whenever any inputs/outputs change.

The condition can also use `output` values; they work in the same way (`output.foo` gives you the value of the output `foo`). If you have a situation where you wish you could use an R expression as your `condition` argument, you can create a reactive expression in `server.R` and assign it to a new output, then refer to that output in your



ui.R

```
# Partial example
selectInput("dataset", "Dataset", c("diamonds", "rock", "pressure", "cars")),
conditionalPanel(
  condition = "output.nrows",
  checkboxInput("headonly", "Only use first 1000 rows"))
```

server.R

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

output$nrows <- reactive({
  nrow(datasetInput())
})
```

However, since this technique requires server-side calculation (which could take a long time, depending on what other reactive expressions are executing) we recommend that you avoid using `output` in your conditions unless absolutely necessary.

## Creating Controls On the Fly With `renderUI`

*Note: This feature should be considered experimental. Let us know whether you find it useful.*

Sometimes it's just not enough to show and hide a fixed set of controls. Imagine prompting the user for a latitude/longitude, then allowing the user to select from a checklist of cities within a certain radius. In this case, you can use the `renderUI` expression to dynamically create controls based on the user's input.

ui.R

```
# Partial example
numericInput("lat", "Latitude"),
numericInput("long", "Longitude"),
uiOutput("cityControls")
```

server.R

```
# Partial example
output$cityControls <- renderUI({
  cities <- getNearestCities(input$lat, input$long)
  checkboxGroupInput("cities", "Choose Cities", cities)
})
```

`renderUI` works just like `renderPlot`, `renderText`, and the other output rendering functions you've seen before, but it expects the expression it wraps to return an HTML tag (or a list of HTML tags, using `tagList`). These tags can include inputs and outputs.

In `ui.R`, use a `uiOutput` to tell Shiny where these controls should be rendered.

## Use JavaScript to Modify the Page

*Note: This feature should be considered experimental. Let us know whether you find it useful.*

You can use JavaScript/jQuery to modify the page directly. General instructions for doing so are outside the scope of this tutorial, except to mention an important additional requirement. Each time you add new inputs/outputs to the DOM, or remove existing inputs/outputs from the DOM, you need to tell Shiny. Our current recommendation is:

- Before making changes to the DOM that may include adding or removing Shiny inputs or outputs, call `Shiny.unbindAll()`.
- After such changes, call `Shiny.bindAll()`.

If you are adding or removing many inputs/outputs at once, it's fine to call `Shiny.unbindAll()` once at the beginning and `Shiny.bindAll()` at the end – it's not necessary to put these calls around each individual addition or removal of inputs/outputs.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Christopher Peters

---

Just want to offer feedback that the uiOutput / renderUI feature is great --- consider this a vote for further development of the feature.

Garrett

---

Thank you, Christopher. We're glad to hear that it is useful!

Mona Jalal

---

Hi Garrett, How can I find a working dynamic UI where in based on the selection of first menu other menus appear or something like that? Your examples are partial and I could not run them. Thanks

Alex Lemm

---

renderUI() + uiOutput() are great. I usually use both to create widgets whose values are updated regularly by a Cron Job. After using this feature for over a year now, I hope it will lose its "experimental" status soon. :-)

## 2.15 Shiny HTML Tags Glossary

# Shiny HTML Tags Glossary

ADDED: 16 APR 2014

BY: GARRETT GROLEMUND

In [Customize your Shiny UI with HTML](#) you saw that Shiny provides a list of functions named `tags`. Each function in the list creates an HTML tag that you can use to layout your Shiny App. But what if you are unfamiliar with HTML tags? The glossary below explains what the most popular tags in `tags` do.

## tags

The `shiny::tags` object contains R functions that recreate 110 HTML tags.

```
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"
```

You can use any of these functions by subsetting the `tags` list. For example to create an `h1` header in HTML, run:

```
tags$h1("My header")
## <h1>My header</h1>
```

Some tags functions come with a helper function that makes accessing them easier. For example, the `shiny::h1` function is a wrapper for `tags$h1`.

However, you can access many of the functions in tags only through `tags` because they share a name with a common R function.

The glossary below explains what the most popular tag functions do. The tag functions that do not appear here are less popular, but still useful. You can look them up at an HTML documentation site such as [w3schools](#).

## a

Creates a link to a web page. You can access the “a” tag with the helper function `a`.

Common Attributes Description

`href` the address of the web page to link to

```
tags$a(href="www.rstudio.com", "Click here!")
## <a href="www.rstudio.com">Click here!</a>
```

## audio

Adds an audio element (e.g., a sound to your app).

Common Attributes Description

`autoplay` A valueless attribute. If present, audio starts playing automatically when loaded

`controls` A valueless attribute. If present, play controls are displayed

`src` The location of the audio file to play

`type` The type of file to play

```
tags$audio(src = "sound.mp3", type = "audio/mp3", autoplay = NA, controls = NA)
## <audio src="sound.mp3" type="audio/mp3" autoplay controls></audio>
```

## b

Creates bold text.

```
tags$b("This text is bold.")
## <b>This text is bold.</b>
```

## blockquote

Creates a block of quoted text. Usually it is displayed in a special way.

Common Attributes Description

`cite` the source of the quote

```
tags$blockquote("Tidy data sets are all the same. Each messy data set is messy in its own way.", cite = "Hadley Wickham")
## <blockquote cite="Hadley Wickham">Tidy data sets are all the same. Each messy data set is messy in its own way.</blockquote>
```

## br

Creates a line break. You can use the helper function `br`.

```
tags$div(
  "Some text followed by a break",
  tags$br(),
  "Some text following a break"
```

```
)
## <div>
##   Some text followed by a break
##   <br/>
##   Some text following a break
## </div>
```

## code

Creates text formatted as computer code. You can use the helper function `code`.

```
tags$code("This text will be displayed as computer code.")
## <code>This text will be displayed as computer code.</code>
```

## div

Creates a section (e.g., “division”) of an HTML document. divs provide a useful hook for CSS styling. You can use the helper function `div`.

Common Attributes Description

class	The class of the div, a useful way to style the div with CSS
id	The ID of the div, a useful way to style the div with CSS
style	CSS styling to apply to the div

```
tags$code("This text will be displayed as computer code.")
## <code>This text will be displayed as computer code.</code>
```

## em

Creates emphasized (e.g., italicized) text. You can use the helper function `em`.

```
tags$em("This text is emphasized.")
## <em>This text is emphasized.</em>
```

## embed

Embed a plug-in or third party application.

Common Attributes Description

src	The source of the file to embed
type	The MIME type of the embedded content
height	The height of the embedded content
width	The width of the embedded content

```
tags$embed(src = "animation.swf")
## <embed src="animation.swf"/>
```

## h1, h2, h3, h4, h5, h6

Adds hierarchical headings. `h1` creates a first level heading and `h2` through `h6` create a hierarchy of decreasing subheadings. You can use the helper functions `h1`, `h2`, `h3`, `h4`, `h5`, `h6`.

```
tags$div(
  tags$h1("Heading"),
```

```

tags$sh2("Subheading"),
tags$sh3("Subsubheading"),
tags$sh4("Subsubsubheading"),
tags$sh5("Subsubsubsubheading"),
tags$sh6("Subsubsubsubsubheading")
)
## <div>
##   <h1>Heading</h1>
##   <h2>Subheading</h2>
##   <h3>Subsubheading</h3>
##   <h4>Subsubsubheading</h4>
##   <h5>Subsubsubsubheading</h5>
##   <h6>Subsubsubsubsubheading</h6>
## </div>

```

## hr

Adds a horizontal line (e.g., horizontal rule). You can use the helper function `hr`.

```

tags$hr()
## <hr/>

```

## i

Creates italicized text.

```

tags$i("This text is italicized.")
## <i>This text is italicized.</i>

```

## iframe

Creates an inline frame to embed an HTML document in.

Common Attributes Description

src	The URL of the HTML document to embed
srcdoc	A raw HTML document to embed
scrolling	Should iframe display scrollbars ( yes , no , auto )
seamless	A valueless attribute. Should the iframe seem like part of the web page?
height	The height of the iframe
width	The width of the iframe
name	The name of the iframe

```

tags$iframe(src = "www.rstudio.com", seamless=NA)
## <iframe src="www.rstudio.com" seamless></iframe>

```

## img

Creates an image. You can use the helper function `img`.

Common Attributes Description

src	The source of the image to embed
height	The height of the image
width	The width of the image

```
tags$img(src = "www.rstudio.com", width = "100px", height = "100px")
## 
```

## link

Creates a link to a separate document. Normally used with CSS style sheets.

## ol and li

Create an ordered list (i.e., a numbered list).

```
tags$ol (
  tags$li("First list item"),
  tags$li("Second list item"),
  tags$li("Third list item")
)
## <ol>
## <li>First list item</li>
## <li>Second list item</li>
## <li>Third list item</li>
## </ol>
```

## p

Create a paragraph (a block of text that begins on its own line). You can access the p tag with the helper function `p` too.

```
tags$div(
  tags$p("First paragraph"),
  tags$p("Second paragraph"),
  tags$p("Third paragraph")
)
## <div>
## <p>First paragraph</p>
## <p>Second paragraph</p>
## <p>Third paragraph</p>
## </div>
```

## pre

Create pre-formatted text, text that looks like computer code. You can use the helper function `pre`.

```
tags$pre("This text is preformatted.")
## <pre>This text is preformatted.</pre>
```

## script

Add a client-side script such as javascript. You must wrap the actual script in `HTML` to prevent it from being passed as text.

```
tags$script(HTML("if (window.innerHeight < 400) alert('Screen too small');"))
```

## span

Create a group of inline elements. Normally used to style a string of text. You can use the helper function `span`.

```
tags$div(
  HTML(paste("This text is ", tags$span(style="color:red", "red"), sep = ""))
)
## <div>This text is <span style="color:red">red</span></div>
```

## strong

Create bold text. You can use the helper function `strong`.

```
tags$strong("This text is strongly emphasized.")
## <strong>This text is strongly emphasized.</strong>
```

## style

Create style specifications. A way to add CSS styles directly to your Shiny App.

## sub, sup

Create subscript or super script.

```
tags$div(
  HTML(paste("E = mc", tags$sup(2), sep = "")),
  HTML(paste("H", tags$sub(2), "0", sep = ""))
)
## <div>
##   E = mc<sup>2</sup>
##   H<sub>2</sub>0
## </div>
```

## ul and li

Create an unordered list (i.e., a list of bullet points).

```
tags$ul(
  tags$li("First list item"),
  tags$li("Second list item"),
  tags$li("Third list item")
)
## <ul>
##   <li>First list item</li>
##   <li>Second list item</li>
##   <li>Third list item</li>
## </ul>
```

## video

Add a video.

Common Attributes Description

<code>autoplay</code>	A valueless attribute. If present, video starts playing automatically when loaded
<code>controls</code>	A valueless attribute. If present, Shiny will display play controls.



src	The location of the video file to play
height	The height of the video
width	The width of the video

```
tags$video(src = "video.mp4", type = "video/mp4", autoplay = NA, controls = NA)
## <video src="video.mp4" type="video/mp4" autoplay controls></video>
```

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Spencer Boucher

---

Might be better to link to a more respected source of information than [w3schools](#), like Mozilla Developer Network for example.

ImAndy

---

the tags\$div example is a repeat of the tags\$code

Nat Condit-Schultz

---

Is there a way to have the audio src be controlled by a widget? I'd like the user to be able to select from a list of audio files to hear.

Shrimp Fish

---

I have a sound file, named "testaudio.mp3", in the subdirectory "www". I used the following syntax:

```
tags$audio(src = "testaudio.mp3", type = "audio/mp3", autoplay = NA, controls = NA)
```

[comments powered by Disqus](#)

## 2.16 Progress indicators

# Progress indicators

ADDED: 10 SEP 2014  
BY: WINSTON CHANG

If your Shiny app contains computations that take a long time to complete, a progress bar can improve the user experience by communicating how far along the computation is, and how much is left. Progress bars were added in Shiny 0.10.2.

To see progress bars in action, see [this app](#) in the gallery.

## Adding a progress indicator

The simplest way to add a progress indicator is to put `withProgress()` inside of the `reactive()`, `observer()`, or `renderXx()` function that contains the long-running computation. In this example, we'll simulate a long computation by creating an empty data frame and then adding one row to it every 0.1 seconds. (Note that this example is written as a [single-file app](#)). To run this, you can copy and paste the code into the R console.)

```
server <- function(input, output) {  
  output$plot <- renderPlot({  
    input$goPlot # Re-run when button is clicked  
  
    # Create 0-row data frame which will be used to store data  
    dat <- data.frame(x = numeric(0), y = numeric(0))  
  
    withProgress(message = 'Making plot', value = 0, {  
      # Number of times we'll go through the loop  
      n <- 10  
  
      for (i in 1:n) {  
        # Each time through the loop, add another row of data. This is  
        # a stand-in for a long-running computation.  
        dat <- rbind(dat, data.frame(x = rnorm(1), y = rnorm(1)))  
  
        # Increment the progress bar, and update the detail text.  
        incProgress(1/n, detail = paste("Doing part", i))  
  
        # Pause for 0.1 seconds to simulate a long computation.  
        Sys.sleep(0.1)  
      }  
    })  
  
    plot(dat$x, dat$y)  
  })  
}
```

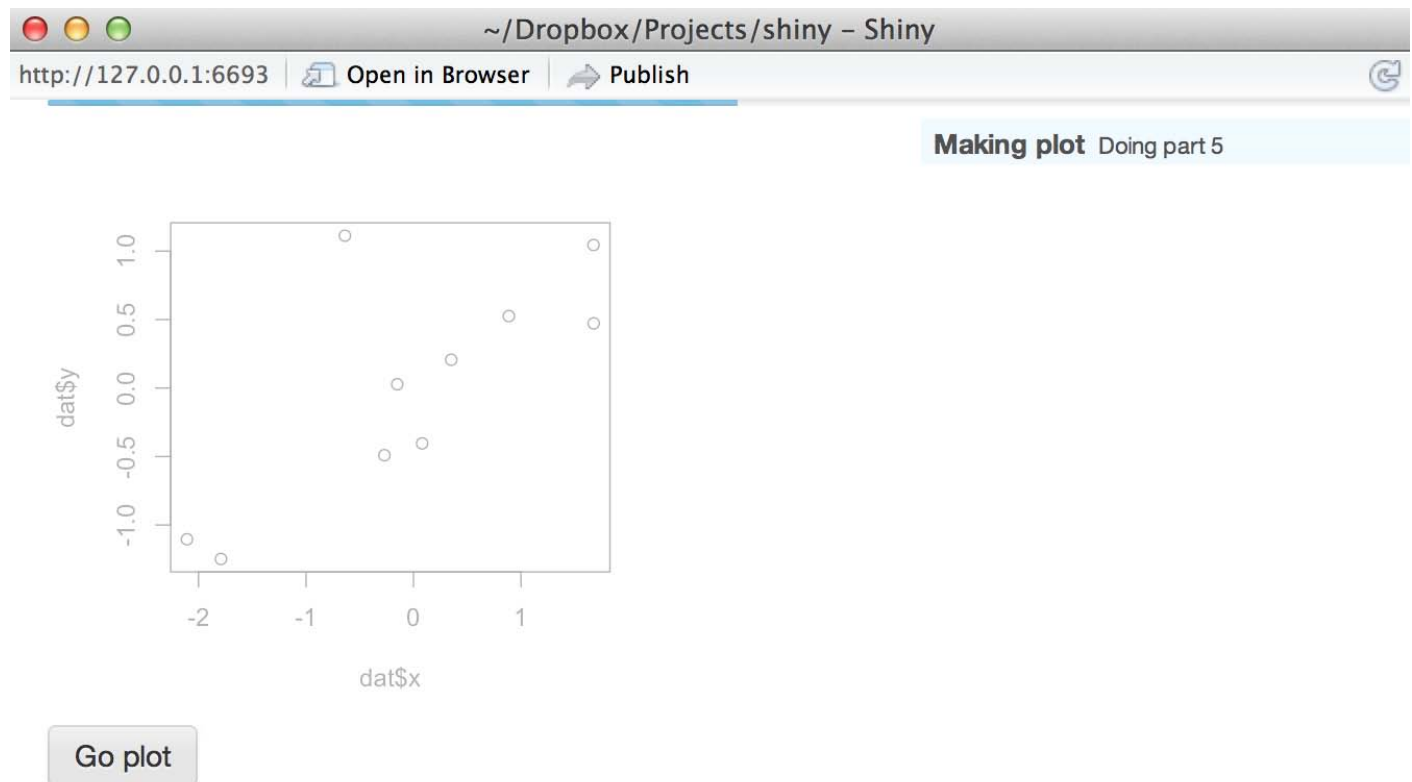
```
ui <- shinyUI(basicPage(
  plotOutput('plot', width = "300px", height = "300px"),
  actionButton('goPlot', 'Go plot')
))

shinyApp(ui = ui, server = server)
```

This is what will happen:

The `withProgress()` function is used to start a progress bar, and then the value is incremented with `incProgress()`. By default, the range of values for the bar goes from 0 to 1, although this can be changed with the `min` and `max` arguments.

There are two levels of messages: `message`, and `detail`. The `message` is presented in bold, and the `detail` is presented in normal-weight text.



In the example above, `withProgress()` is used inside of `renderPlot()`, but it could also be used inside of any other render function, like `renderTable()`, or inside of a `reactive()`.

It's possible to nest calls to `withProgress`; if you do this, the second-level progress bar will appear directly under the top-level progress bar, and the second-level text will appear under the top-level text. Further levels of nesting will have a similar pattern.

## Using a Progress object

The `withProgress()` function is a convenient interface around a `Progress` object. In most cases, it's simpler and easier to use `withProgress`, but in some cases, you may need the greater level of control provided by the `Progress` object. Before we delve into a more complex example, we'll simply convert the example above from using `withProgress` to using a `Progress` object.

```
server <- function(input, output) {
  output$plot <- renderPlot({
    input$goPlot # Re-run when button is clicked
```

```

# Create 0-row data frame which will be used to store data
dat <- data.frame(x = numeric(0), y = numeric(0))

# Create a Progress object
progress <- shiny::Progress$new()
# Make sure it closes when we exit this reactive, even if there's an error
on.exit(progress$close())

progress$set(message = "Making plot", value = 0)

# Number of times we'll go through the loop
n <- 10

for (i in 1:n) {
  # Each time through the loop, add another row of data. This is
  # a stand-in for a long-running computation.
  dat <- rbind(dat, data.frame(x = rnorm(1), y = rnorm(1)))

  # Increment the progress bar, and update the detail text.
  progress$inc(1/n, detail = paste("Doing part", i))

  # Pause for 0.1 seconds to simulate a long computation.
  Sys.sleep(0.1)
}

plot(dat$x, dat$y)
})
}

ui <- shinyUI(basicPage(
  plotOutput('plot', width = "300px", height = "300px"),
  actionButton('goPlot', 'Go plot')
))

shinyApp(ui = ui, server = server)

```

Notice that we need to explicitly create the `progress` object and make sure that it closes properly, using `on.exit()`.

## A more complex Progress example

In the example below, the `renderTable()` calls out to another function, `compute_data()`, to do the long-running computation. If we were to just update the progress indicator before and after `compute_data()` were called, then it would only be updated at the beginning, when nothing has been done yet, and at the end, when the computation is completed. In some cases, the best we can do may be to set it to a starting value of, say, 0.3, and then move it to 1 at completion. This may be true if, for example, the function is in an external package.

However, if you do have control over the function doing the computation, you may want to modify it to accept either a `Progress` object which it will update directly, or to accept a function which it calls each time it does some part of the computation.

In the example below, we'll take the latter approach. The `compute_data()` function accepts an optional `updateProgress` function, which it calls periodically as it does the computation. The `updateProgress` function is a closure that captures the `Progress` object; each time it's called, it updates the progress indicator.

Again, you can copy and paste this code in your R console to see it in action:

```

# This function computes a new data set. It can optionally take a function,
# updateProgress, which will be called as each row of data is added.
compute_data <- function(updateProgress = NULL) {
  # Create 0-row data frame which will be used to store data
  dat <- data.frame(x = numeric(0), y = numeric(0))

  for (i in 1:10) {
    Sys.sleep(0.25)

    # Compute new row of data
    new_row <- data.frame(x = rnorm(1), y = rnorm(1))

    # If we were passed a progress update function, call it
    if (is.function(updateProgress)) {
      text <- paste0("x:", round(new_row$x, 2), " y:", round(new_row$y, 2))
      updateProgress(detail = text)
    }

    # Add the new row of data
    dat <- rbind(dat, new_row)
  }

  dat
}

server <- function(input, output) {
  output$table <- renderTable({
    input$goTable

    # Create a Progress object
    progress <- shiny::Progress$new()
    progress$set(message = "Computing data", value = 0)
    # Close the progress when this reactive exits (even if there's an error)
    on.exit(progress$close())

    # Create a callback function to update progress.
    # Each time this is called:
    # - If `value` is NULL, it will move the progress bar 1/5 of the remaining
    #   distance. If non-NULL, it will set the progress to that value.
    # - It also accepts optional detail text.
    updateProgress <- function(value = NULL, detail = NULL) {
      if (is.null(value)) {
        value <- progress$getValue()
        value <- value + (progress$getMax() - value) / 5
      }
      progress$set(value = value, detail = detail)
    }

    # Compute the new data, and pass in the updateProgress function so
    # that it can update the progress indicator.
    compute_data(updateProgress)
  })
}

ui <- shinyUI(basicPage(
  tableOutput('table'),

```

```
actionbutton('goTable', 'Go Table')
))
```

```
shinyApp(ui = ui, server = server)
```

It's possible to use other constructions for the `updateProgress` function that have different behavior. In the example above, each time `updateProgress()` is called, the progress bar moves 1/5 of the remaining distance. This tells the user that something is happening, and it's simple because you don't need to know ahead of time how many times it's going to run. However, it's not the most accurate representation of progress, since it approaches the end asymptotically, whereas a linear approach would be more accurate.

One alternative is to have the external function call `updateProgress()` with a specific value. If, for example, the external function knows that it will iterate over the loop 100 times, it could call `updateProgress()` with `value=0.01`, then `value=0.02`, and so on.

Another alternative is to construct a different `updateProgress` callback, one which increments by a fixed amount each time. To do this, before you call `compute_data()`, you must know how many times it will call `updateProgress()` in the loop. Let's assume that it will be called 20 times. Then `updateProgress` could be defined like so:

```
n <- 20
updateProgress <- function(detail = NULL) {
  progress$inc(amount = 1/n, detail = detail)
}
```

Each time this version of `updateProgress()` is called, it moves the bar 1/20th of the total distance.

## Recap

You can add progress indicators to your app, using the simpler `withProgress()` interface, or the `Progress` object if you need more control. These progress indicators can provide feedback to the user that will make their experience more satisfying.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Xiushi Le

---

let's say I want to join two big dataset, how to I slice the join operation so that it works with the progress feature? Also, is there any function in R that can estimate the execution time of an operation in R?

Christopher Peters

I'm curious if there's a simple way to change the color? This progress bar doesn't feel prominent enough, so I was hoping to make it red.

Dean Attali

---

If you know a bit of CSS, it should be easy. If you make the shiny progress take a long enough time (ie. 1 minute), then you'll have enough time to look at the page DOM to see what the progress bar HTML looks like and to extract which elements you need to change.

For example, it looks like adding the following CSS (read the article about customizing CSS if you're not sure how to add it) would make the progress bar [comments powered by Disqus](#)

## 2.17 Getting started with shinyapps.io

# Getting started with shinyapps.io

ADDED: 18 MAR 2014

BY: ANDY KIPP

Shinyapps.io is a platform as a service (PaaS) for hosting Shiny web apps (applications). This guide will show you how to create a shinyapps.io account and deploy your first application to the cloud.

Before you get started with shinyapps.io, you will need:

- An R development environment, such as the RStudio IDE
- (for Windows users only) [RTools](#) for building packages
- (for Mac users only) XCode Command Line Tools for building packages
- (for Linux users only) GCC
- The [devtools](#) R package (version 1.4 or later)
- The latest version of the [shinyapps](#) R package

## How to install devtools

Shinyapps.io uses the latest improvements to the `devtools` package. To use shinyapps.io, you must update `devtools` to version 1.4 or later. To install `devtools` from CRAN, run the code below. Then restart your R session.

```
install.packages('devtools')
```

## How to install shinyapps

The `shinyapps` package deploys applications to the shinyapps.io service. Currently, you need to install the `shinyapps` package from its development page at Github. You can do this by running the R command:

```
devtools::install_github('rstudio/shinyapps')
```

After the `shinyapps` package has been installed, load it into your R session:

```
library(shinyapps)
```

## Create a shinyapps.io account

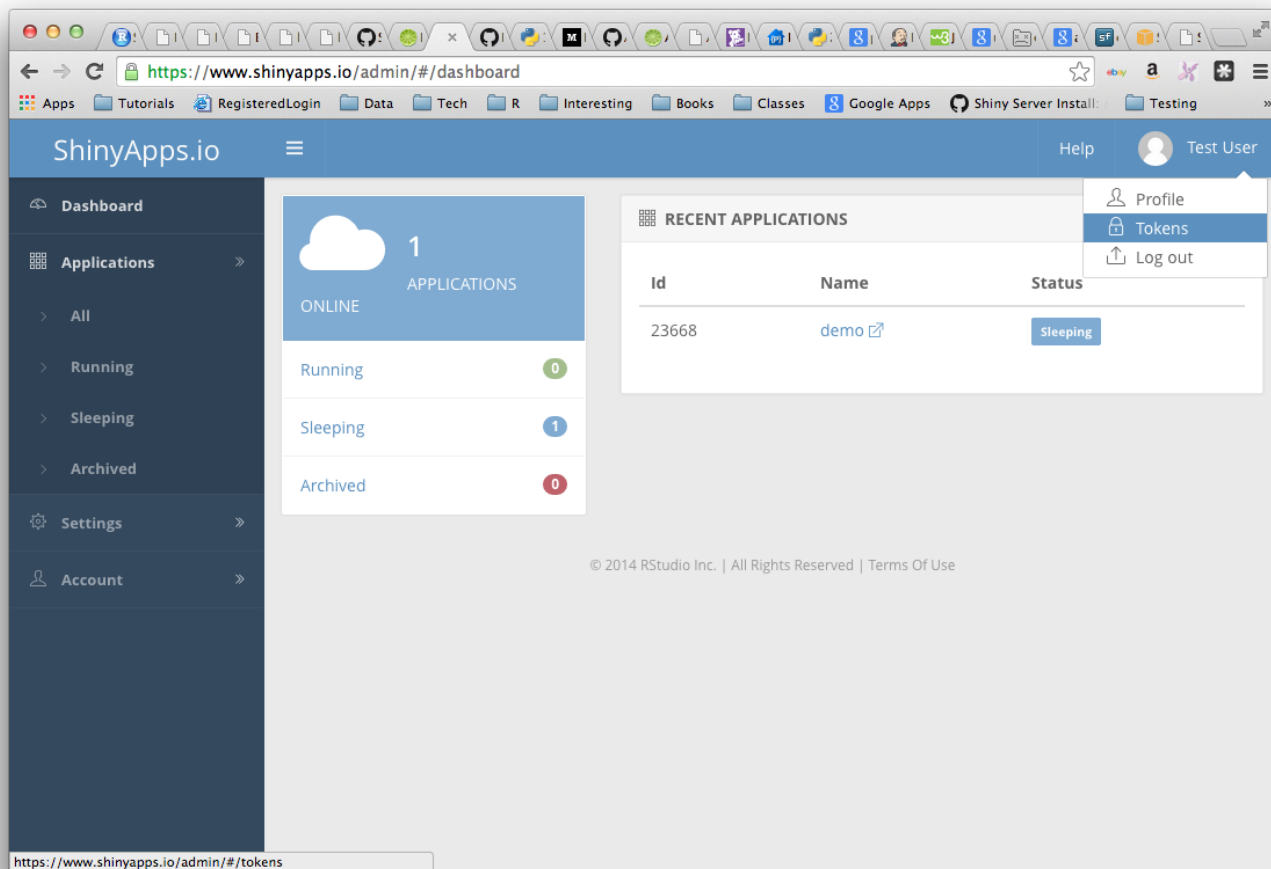
Go to [shinyapps.io](#) and click “Log In.” The site will ask you to sign in using your Google Account.

The first time you sign in, shinyapps.io prompts you to setup your account. Shinyapps.io uses the account name as the domain name for all your apps. Account names must be between four and 63 characters and can contain only letters, numbers, and dashes (-). Account names may not begin with a number or a dash, and they can not end with a dash (see [RFC 952](#)). Some account names may be reserved.



## Configure shinyapps

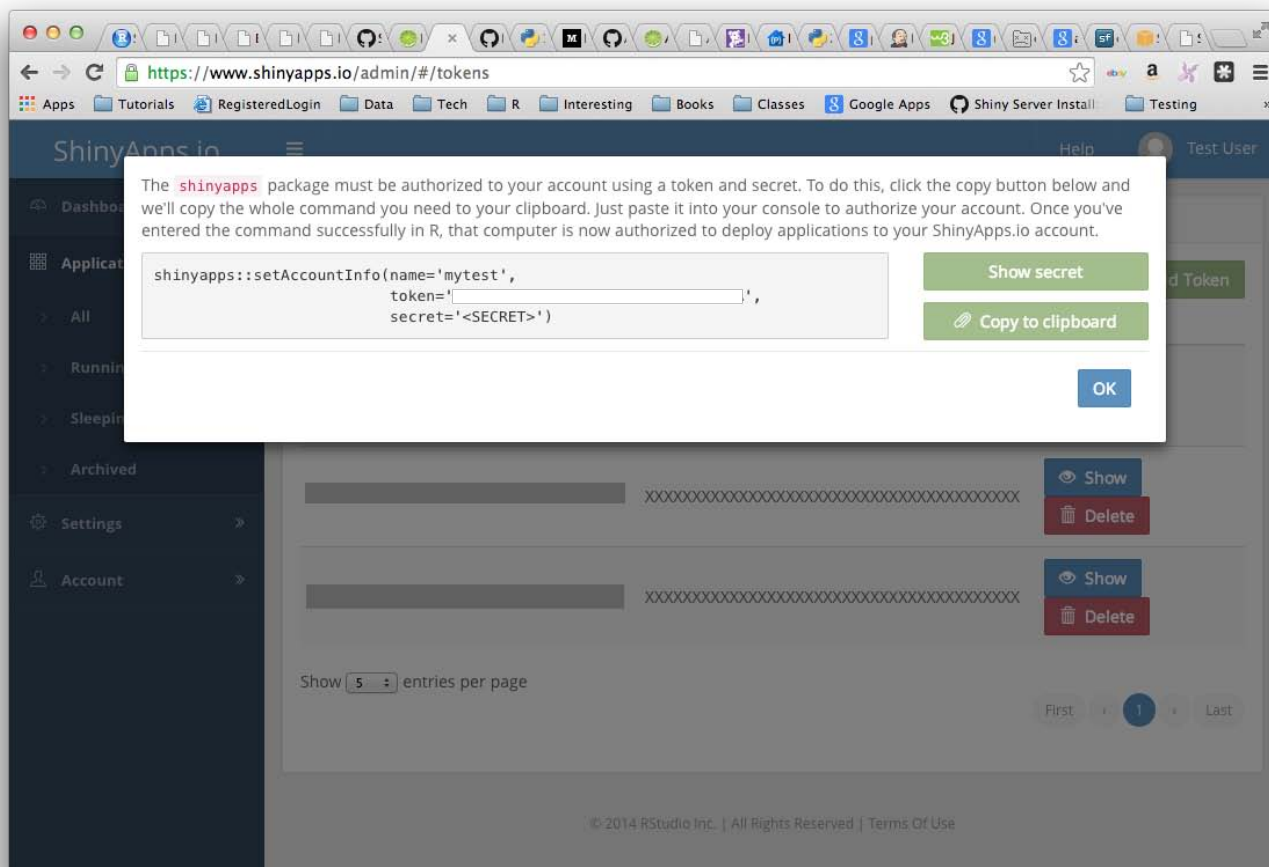
Once you set up your account in shinyapps.io, you can configure the `shinyapps` package to use your account. Shinyapps.io automatically generates a token and secret for you, which the `shinyapps` package can use to access your account. Retrieve your token from the shinyapps.io dashboard. Tokens are listed under `Tokens` in the menu at the top right of the shinyapps dashboard (under your avatar).



You can configure the `shinyapps` package to use your account with two methods:

### Method 1

Click the show button on the token page. A window will pop up that shows the full command to configure your account using the appropriate parameters for the `shinyapps::setAccountInfo` function. Copy this command to your clip board, and then paste it into the command line of RStudio and click enter.



## Method 2

Run the 'setAccountInfo' function from the `shinyapps` package passing in the token and secret from the Profile / Tokens page.

```
shinyapps::setAccountInfo(name="<ACCOUNT>", token="<TOKEN>", secret="<SECRET>")
```

Once you have configured your `shinyapps` installation, you can use it to upload applications to shinyapps.io. In the second part of this guide, we will build a demo application, upload it to shinyapps.io, and create a password for the application.

## A Demo app

For this guide, we created an RStudio project named "demo" that contains a Shiny application to upload to shinyapps.io. Follow these steps to create your own Shiny app.

## Install application dependencies

The demo application we will deploy requires the `ggplot2` package and the `shiny` package. Ensure that any package required by your application is installed locally before you deploy your application:

```
install.packages(c('ggplot2', 'shiny'))
```

## ui.R and server.R

We placed two Shiny source files, `ui.R` and `server.R`, in our demo application. You can cut and paste the code

below to make your own Shiny application:

server.R

```
library(shiny)
library(ggplot2)

function(input, output) {

  dataset <- reactive({
    diamonds[sample(nrow(diamonds), input$sampleSize), ]
  })

  output$plot <- renderPlot({

    p <- ggplot(dataset(), aes_string(x=input$x, y=input$y)) + geom_point()

    if (input$color != 'None')
      p <- p + aes_string(color=input$color)

    facets <- paste(input$facet_row, '~', input$facet_col)
    if (facets != ', ~ .')
      p <- p + facet_grid(facets)

    if (input$jitter)
      p <- p + geom_jitter()
    if (input$smooth)
      p <- p + geom_smooth()

    print(p)

  }, height=700)
}
```

ui.R

```
library(shiny)
library(ggplot2)

dataset <- diamonds

fluidPage(

  titlePanel("Diamonds Explorer"),

  sidebarPanel(

    sliderInput('sampleSize', 'Sample Size', min=1, max=nrow(dataset),
               value=min(1000, nrow(dataset)), step=500, round=0),

    selectInput('x', 'X', names(dataset)),
    selectInput('y', 'Y', names(dataset), names(dataset)[[2]]),
    selectInput('color', 'Color', c('None', names(dataset))),

    checkboxInput('jitter', 'Jitter'),
    checkboxInput('smooth', 'Smooth'),
```

```
selectInput('facet_row', 'Facet Row', c(None='.', names(dataset))),
selectInput('facet_col', 'Facet Column', c(None='.', names(dataset)))
),

mainPanel (
  plotOutput('plot')
)
)
```

## Test your application

Test that your application works by running it locally. Set your [working directory](#) to your app directory, and then run:

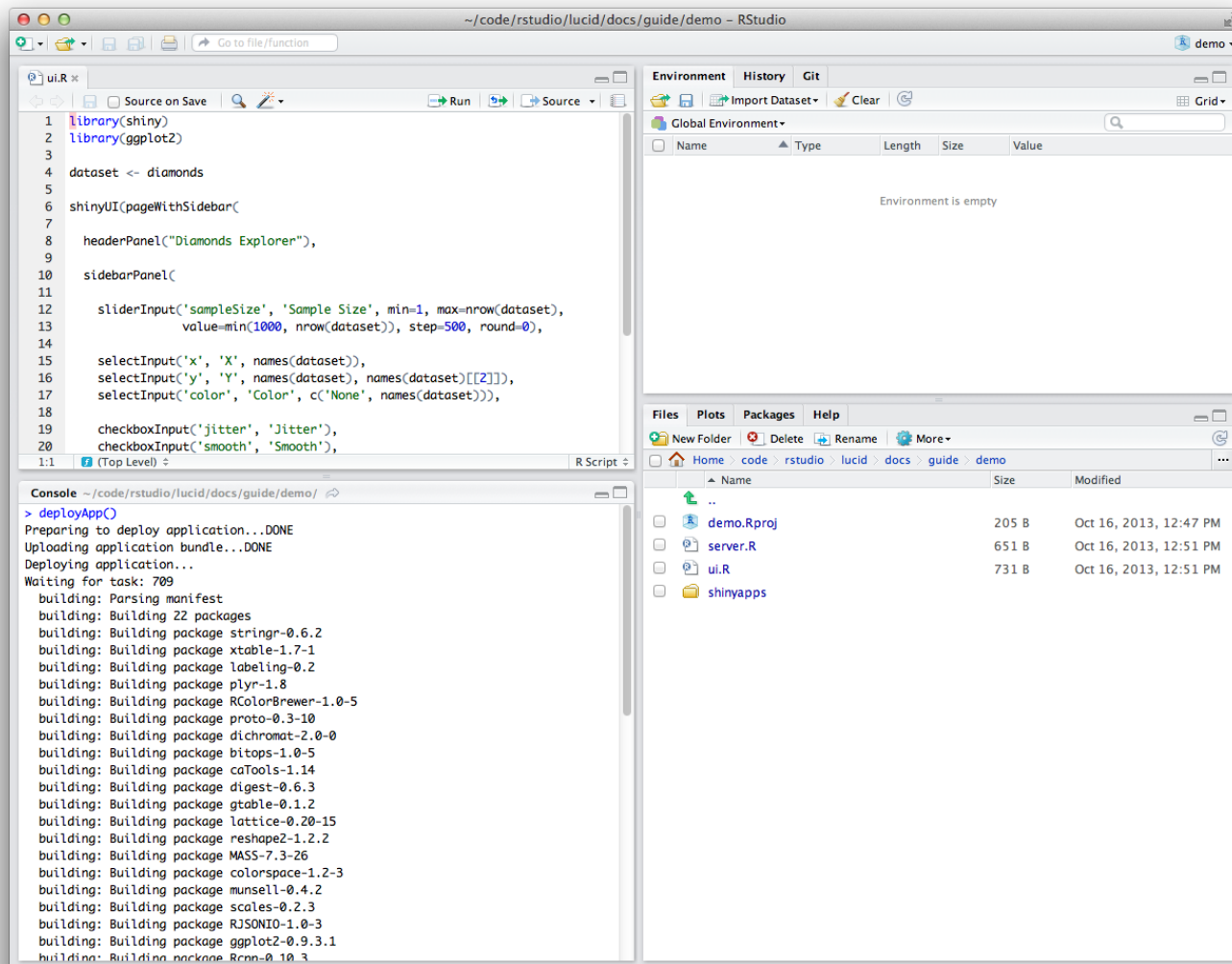
```
library(shiny)
runApp()
```

Now that the application works, let's upload it to shinyapps.io.

## Deploying apps

To deploy your application, use the `deployApp` command from the `shinyapps` packages.

```
library(shinyapps)
deployApp()
```



Once the deployment finishes, your browser should open automatically to your newly deployed application.

Congratulations! You've deployed your first application. :-)

Feel free to make changes to your code and run `deployApp` again. `shinyapps` can deploy an app much more quickly after the first deployment.

## Package dependencies

When you deploy your application, the `shinyapps` package attempts to detect the packages that your application uses. `shinyapps` sends this list of packages and their dependencies along with your application to the shinyapps.io service. Then shinyapps.io builds and installs the packages into the R library for your application. The first time you deploy your application, it may take some time to build these packages (depending on how many packages are used). However, you will not wait for these packages to build during future deployments (unless you upgrade or downgrade a package).

## Package sources

Currently the shinyapps.io service supports deploying packages installed from CRAN, GitHub, and BioConductor. We will look to add support for R-Forge packages in the future.

### Important note on GitHub packages

Only packages installed from GitHub with `devtools::install_github` in version 1.4 (or later) of `devtools` are supported. Packages installed with an earlier version of `devtools` must be reinstalled before you can deploy your

application. If you get an error such as “PackageSourceError” when you attempt to deploy, check that you have installed any package from Github with `devtools` 1.4 or later.

## Application instances

Shinyapps.io hosts each app on its own virtualized server, called an instance. Each instance runs an identical copy of the code and packages that you deployed (called the image).

When you deploy an app, shinyapps.io creates a new image with the updated code and packages, and starts one or more instances with the new image. If the app was previously deployed, shinyapps.io shuts down and destroys the old instances. Consider a few implications of this arrangement:

1) Data written by an application to the local filesystem of an instance will be lost when you re-deploy the app. Additionally, the distributed nature of the shinyapps.io platform means that instances may be shut down and re-created at any time for maintenance or to recover from server failures.

2) It is possible to have more than one instance of an application. This situation means that multiple instances of an application do not share a local filesystem. A file written to one instance will not be available to another instance.

Shinyapps.io limits the amount of system resources an instance can consume. The amount of resources available to an instance will depend on its type. The table below outlines the various instance types and how much memory is allowed. By default, shinyapps.io deploys all applications on ‘medium’ instances, which are allowed to use 512 MB of memory.

Instance Type	Memory
small (default)	256 MB
medium	512 MB
large	1024 MB
xlarge	2048 MB
xxlarge	4096 MB

*Note: Instance types and limits are not finalized; RStudio may change them in the future.*

## Application logging

If you’re having problems with your application, it may be helpful to be able to see the log messages it’s producing. If you deployed your application using `shinyapps` version 0.3.57 or later, you can now use the `shinyapps::showLogs()` function to show the log messages of a deployed application. This log will include both `stdout` (log lines produced via `print` or `cat`) and `stderr` (log lines produced by `message`, `warning`, `stop`). You can even use the `streaming=TRUE` option to specify that you want to continuously monitor the file for changes; this will listen for log messages until you interrupt R (typically by pressing `Escape`). If you deployed your application using an older version of the `shinyapps` package, you will need to redeploy it (`deployApp(upload=FALSE)`) before you can use logging.

## Configuring applications

You can change the instance type used by an application with the `configureApp` function from the `shinyapps` package. To change the instance type of your application (here from medium to small), run:

```
shinyapps::configureApp("APPNAME", size="small")
```

This change will redeploy your application using the small instance type.

You can also change the instance type used by an application from the shinyapps.io dashboard. To do this, log in to shinyapps.io, select the application that you wish to configure, and then open the Settings tab.

To learn more about instances and other details of the shinyapps.io architecture, read [Scaling and Performance](#)

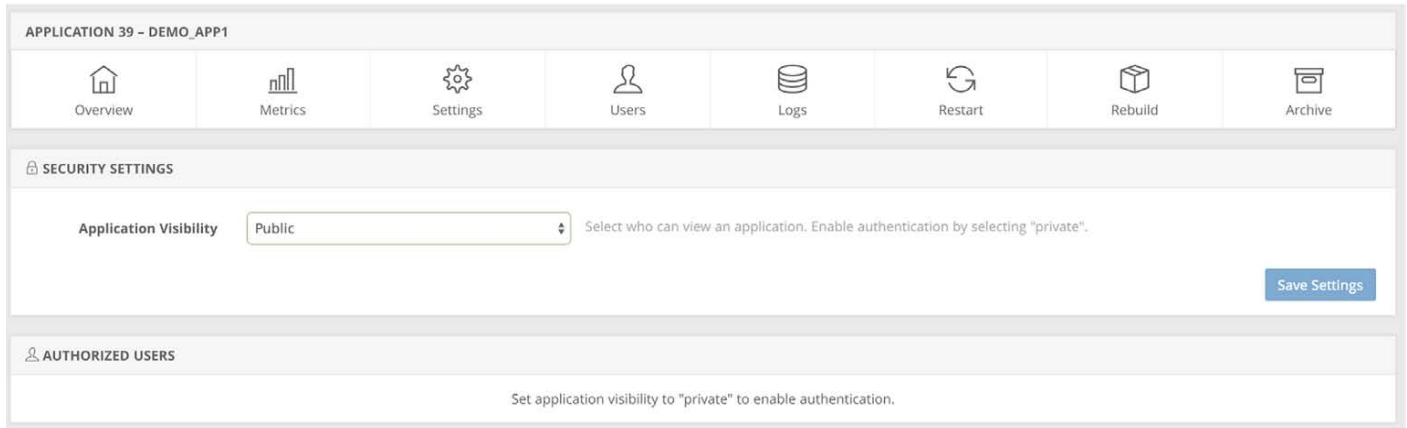
uning. The guide will also show you several advanced options for fine tuning the performance of your apps on shinyapps.io.

## Application authentication

With shinyapps.io, you can limit the access to your application by enabling authentication. Only users who log-in with valid credentials will be able to view or use the app.

To enable authentication in the administrative UI, select the application to modify and click on the Users tab.

Here is a sample application with the default visibility settings:



APPLICATION 39 - DEMO\_APP1

Overview Metrics Settings Users Logs Restart Rebuild Archive

SECURITY SETTINGS

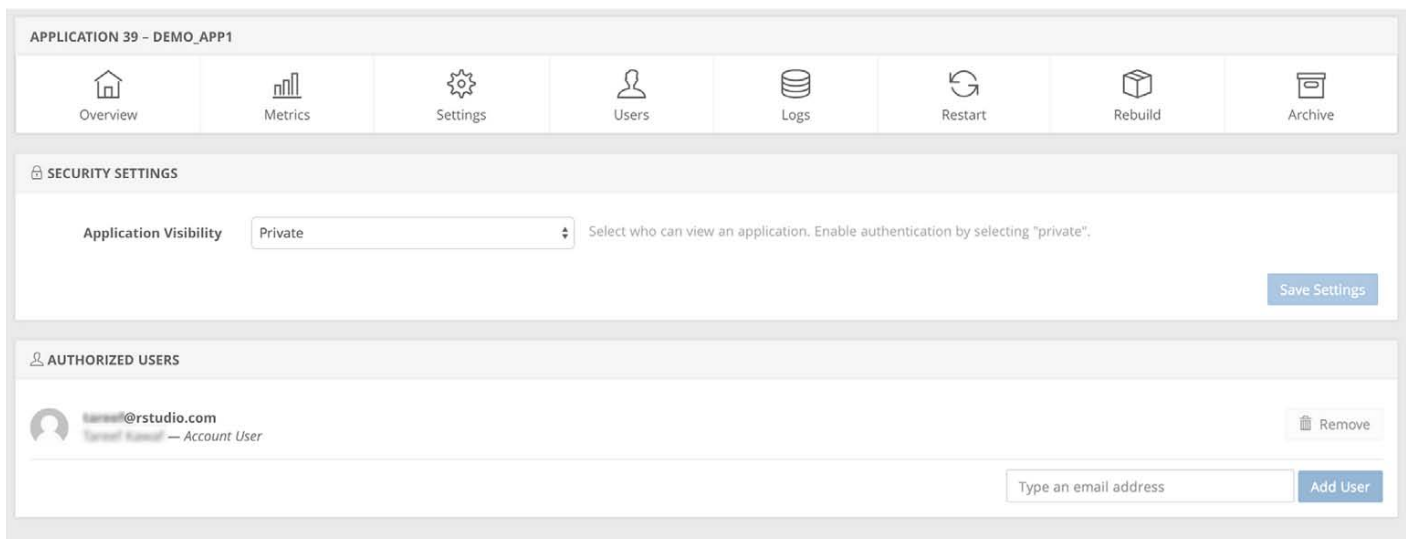
Application Visibility: Public Select who can view an application. Enable authentication by selecting "private".

Save Settings

AUTHORIZED USERS

Set application visibility to "private" to enable authentication.

Change the Application Visibility to Private and click on Save Settings. Changing the visibility of your application *will require a restart of the application*. The Owner of the account and other members of the account will automatically be included in the list of authorized users.



APPLICATION 39 - DEMO\_APP1

Overview Metrics Settings Users Logs Restart Rebuild Archive

SECURITY SETTINGS

Application Visibility: Private Select who can view an application. Enable authentication by selecting "private".

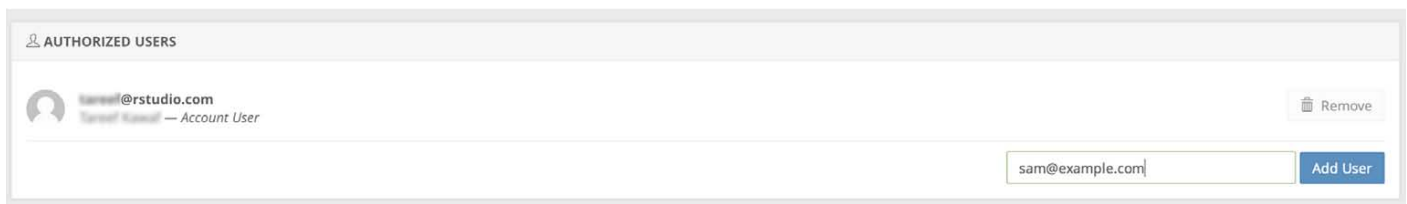
Save Settings

AUTHORIZED USERS

sam@rstudio.com — Account User Remove

Type an email address Add User

After the application is restarted you can add authorized users by entering their email addresses and clicking on Add User.



AUTHORIZED USERS

sam@rstudio.com — Account User Remove

sam@example.com Add User

Each user will receive an email from shinyapps.io with an invite to view your application. If a user does not already have an authenticated account on shinyapps.io, they will be able to create one by authenticating through one of the following three methods:

- Google Authorization
- GitHub authorization
- Shinyapps.io authentication

Shinyapps.io will prompt each visitor to your app for a username and password if they have not been authenticated. Only users who log-in with valid credentials will be able to view or use the app.

If you currently use the pre-beta authentication scheme, please upgrade to the new system by January 28, 2015 as we will be deprecating support for the old authentication system during the beta. For instructions on how to upgrade, please read the guide [here](#).

## Terminate an app

You can remove an app on shinyapps.io from the web with the `terminateApp` command. To use it, run

```
terminateApp("<your app's name>")
```

`terminateApp` requires one argument, the name of the app that you would like to terminate (as a character string). This name should correspond with one of the apps in your shinyapps.io account.

When you run `terminateApp` shinyapps.io will close your app, but the app will remain archived in your shinyapps.io account. This creates efficiencies if you later decide to redeploy your app with `deployApp`.

You can also terminate an app from your shinyapps.io dashboard. To do this, log in to shinyapps.io, select the app that you wish to terminate and then click "Archive."

## Getting help

To seek and share advice about shinyapps.io, please visit the [Shinyapps.io google group](#).

## Recap

Shinyapps.io is an online service for hosting Shiny apps in the cloud. RStudio takes care of all of the details of hosting the app and maintaining the server, which lets you focus on writing great apps!

To use shinyApps.io

- Install the [shinyapps](#) R package from github
- Create an account at [shinyapps.io](#)
- Use the tokens generated by shinyapps.io to configure your `shinyapps` package.
- Deploy apps with `shinyapps::deployApp`
- Terminate apps with `shinyapps::terminateApp`

You can also use shinyapps.io to create secure apps, and manage your authorized users.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)



[Internet Explorer 10+](#)

[Safari](#)

Andrew

---

How do we deploy with helpers.R files and data not hosted on the web?

Dean Attali

---

Another question - is there a way to get the name of the currently logged in authorized user? If someone logs in with username "andy" I want to be able to have something like `getAuthorizedUser()`

Dean Attali

---

As before - found the solution and will post here for anyone else who has this question. You use the variable `session$user` (I don't believe it is documented anywhere, I only found out by debugging the session variable)

Garrett

---

Dean, Thanks for sharing! You're a great help.

comments powered by [Disqus](#)

## 2.18 Setting up custom domains with

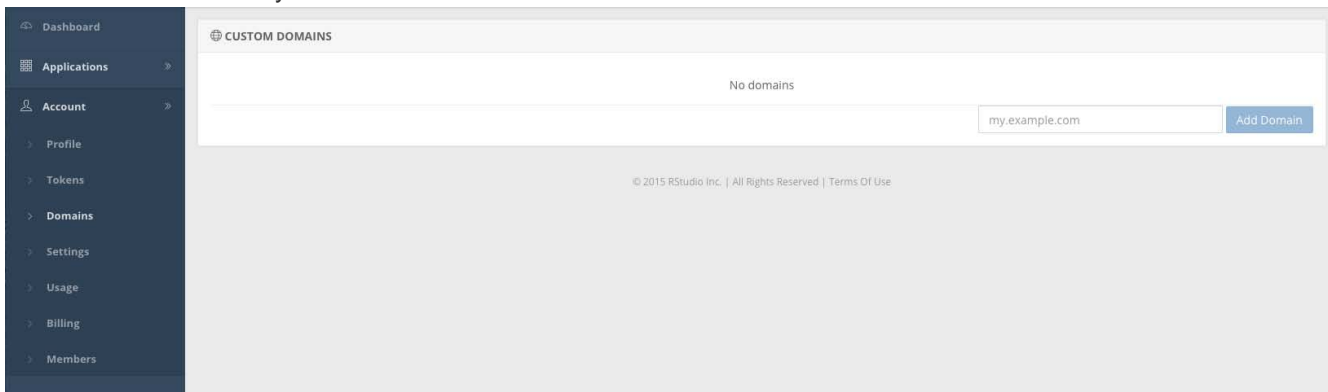
# Setting up custom domains with shinyapps.io

ADDED: 23 JUN 2015

The shinyapps.io Professional plan offers customers the ability to host their Shiny applications using their own domains. This can be useful if you want to control the URL that is viewed by the end-user when you share your application with them.

In order to enable this feature you will want to follow these steps:

1. Decide on domain(s) or subdomain(s) that you would want to host your applications on: (Example: *acmeshinyapps.com* or *apps.acme.com*)
2. Ask your IT administrator to setup a CNAME from that domain/subdomain to your account domain. (Example: If your account name is *acme* and your domain you would like to setup is *apps.acme.com*, then you must create a CNAME from *apps.acme.com* to *acme.shinyapps.io*.) Steps to complete this can vary depending on domain registrar or DNS provider, so we recommend you consult your provider's documentation for exact instructions on completing this step.
3. Once the domain record has been created, log into the shinyapps.io dashboard, and navigate to Account->Domains. From here you can add the domain or subdomain from above and then click Add Domain.



4. Now you are ready to add additional URLs to any of your existing applications. Within the dashboard, find your application in the Applications tab and click on the URLs menu bar choice. You will notice that there is already a single URL which is the one that is created by default. (Note the default URL cannot be removed)
5. Click on Add URL. You can now select from the list of domains you have entered (Example: *apps.acme.com*) and can now specify the path to the application. The URL field below will show you what the final URL would look like.

## Add URL

---

**Domain:**

[Want to add a new domain?](#)

**Path:**

**URL:**

---

---

Note: a single application can be hosted on multiple domains and using different paths, and that all paths are case sensitive.

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

# Scaling and Performance Tuning with shinyapps.io

ADDED: 07 JAN 2015

R is a single threaded application which means that a Shiny application cannot serve two different users at *precisely* the same time. This is not an issue in *most* cases because most computations only take tens or hundreds of milliseconds. As a result, a single R process can usually serve 5 to 30 requests/second. However, as your applications get more complex, requiring more time to service a single request, and as more users interact with the application simultaneously, you may find that the user experience for your applications does not meet your expectations.

## Fine-Tuning Your Shiny apps Performance

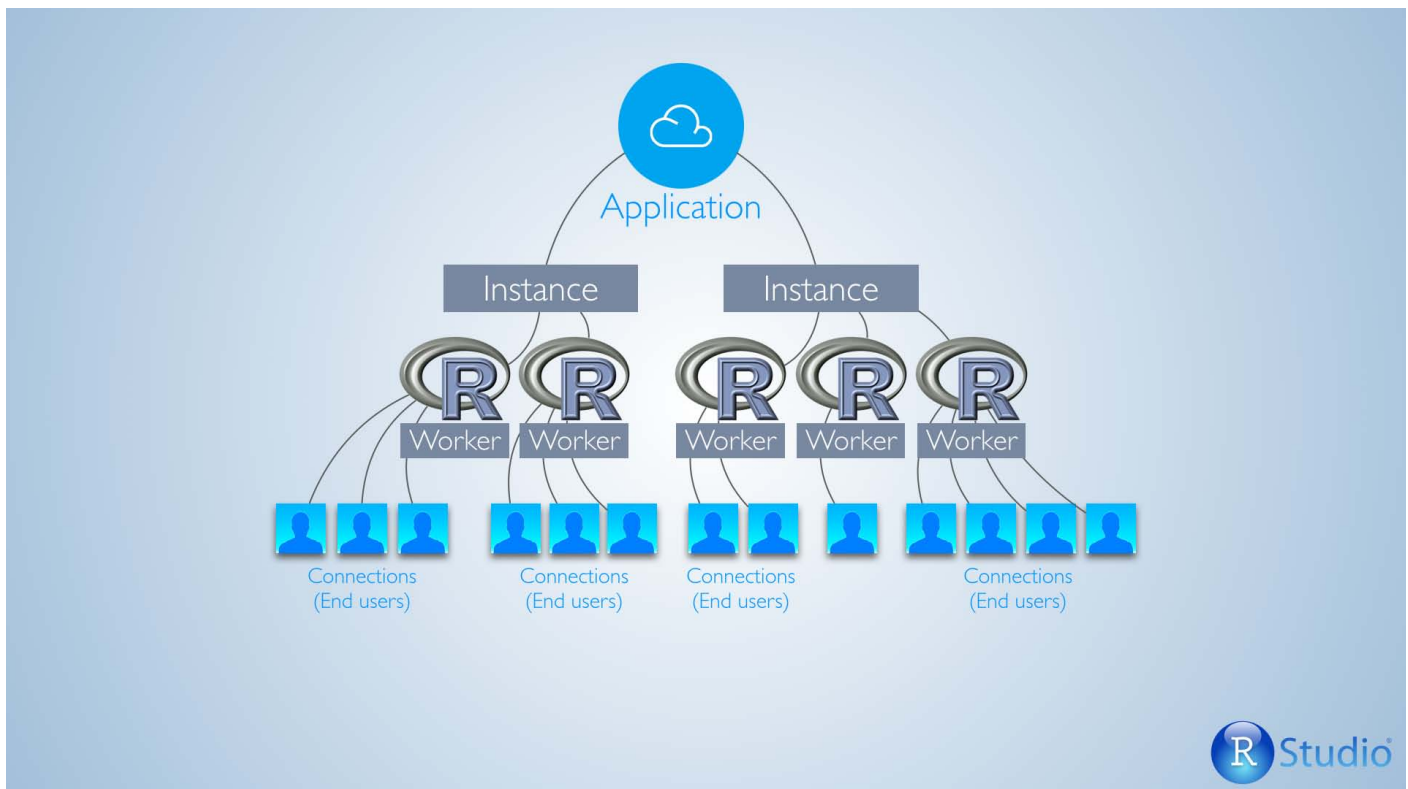
Shinyapps.io lets you optimize the performance of your apps with several tuning parameters. To see your current settings go to the Settings page for any application. The default settings have been chosen to address the needs of most applications.

## Key concepts and terms

There are several ideas that are important when considering the various tuning options that are available. Application

- Application
- Application Instance
- Worker
- Browser Connection

The diagram below shows how these ideas relate to each other.



## Application

An application is a combination of files that you upload to shinyapps.io. These files must include a ui.R file and a server.R file, and can also include data files.

A running application will have at least one Application Instance. You can add additional instances if the application is hosted on a paid tier.

## Application Instance

An Application Instance is a single server that responds to requests from end users. Shinyapps.io will start at least one Application Instance when a user first visits your application, and shinyapps.io will shut down this instance (or these instances) when the application is idle.

Each Application Instance will run one or more R Workers to fulfill user requests.

## Worker

A worker is a special type of R process that an Application Instance runs to service requests to an application. Each Application Instance can run multiple workers. Each worker process is capable of servicing multiple end users depending on the configuration and performance requirements of the application. If there are no processes available to handle a new request, the Application Instance will start a new worker process.

## Browser Connection

A browser connection is a connection between a user's web browser and a worker serving your application.

A user creates a browser connection when they first send a request to your application through their web browser, or when they refresh their browser after it has gone idle. Shinyapps.io assigns each new browser connection to a worker. The worker responds by creating a session for the browser connection to use.

## Tuning parameters

The architecture described above uses two load factors to fine tune the performance of your applications.

- Worker Load Factor - The threshold percentage after which a new browser connection will trigger the addition of a new worker.
- Instance Load Factor - The threshold percentage after which a new connection will trigger the addition of a new Application Instance (limited to the maximum instance limit, free tier is 1)

Each load factor is based on the idea of a threshold percentage, which is the percentage of available connections or processes that are allowed to open before shinyapps.io launches another worker or Application Instance. Both settings are configurable in the Advanced tab within the Settings page for a given application.

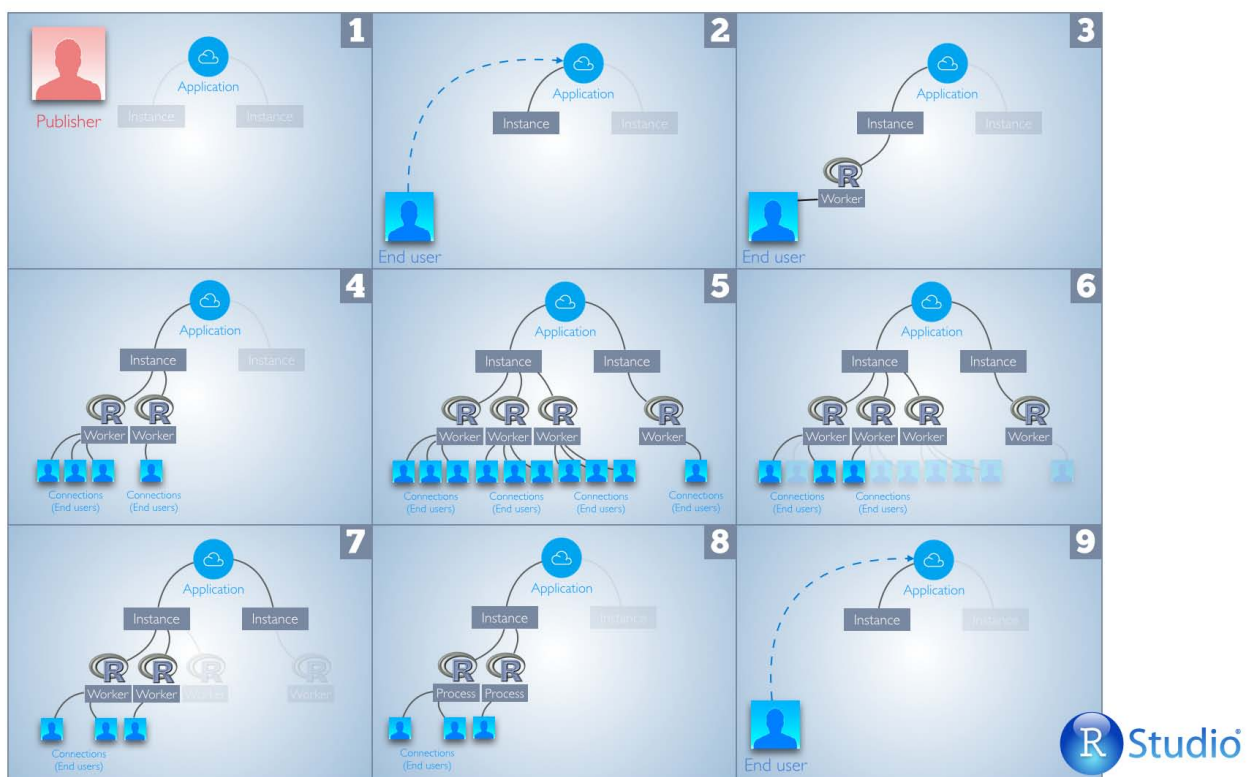
You can also use the Settings page to change:

- the size of your Application Instances
- the maximum number of workers per Application Instance
- the maximum number of connections per worker
- the amount of time at which an instance or connection goes idle.

Each of these changes will further fine tune the performance of your application.

## Lifecycle of an Application

The diagram below shows how shinyapps.io handles user requests throughout the life cycle of an application.



1. Publisher creates a new application and deploys it to shinyapps.io at <https://.shinyapps.io/>
2. A request from an end user triggers the start of an Application Instance
3. Application Instance will start with at least one worker
4. The number of connections to the worker increases as additional end users visit the application. When the Worker Load Factor threshold is exceeded, shinyapps.io adds another worker, so long as the max number of workers per Application Instance has not been reached. New connections are now assigned to the new worker.
5. New workers are added when needed as new users continue to visit the application. When the Instance Load factor is exceeded, shinyapps.io will trigger the addition of another Application Instance, so long as the max number of Application Instances has not been reached (the max number may be one).

6. Shinyapps.io closes connections as end users close their browsers or are idle for longer than the Idle Timeout.
7. Shinyapps.io shuts down each worker once it has no further connections open.
8. Shinyapps.io turns off each Application Instance once it has no running workers, or once its workers are idle for longer than the Instance Idle Timeout. This threshold timeout should be increased if you would like to avoid restarting the application. Note: Increasing the timeout will use up more active hours.
9. A new request from an end user causes shinyapps.io to turn on an Application Instance, and stages 2-9 repeat.

## Examples:

Assuming the following settings:

Instance Load Factor (default is 50%)

Worker Load Factor (default is 5%)

Max worker processes (default is 3)

Max # of concurrent connections supported per worker (default is 50)

Determining when another worker would be started:

Max # of Concurrent connections per worker \* Worker Load Factor

$50 * 5\% = 2.5$  (meaning the 3rd Browser Connection would add another worker up to the Max worker processes)

Determining when another Application Instance would be started:

Max # of connections per worker \* Max worker processes \* Instance Load Factor

$50 * 3 * 50\% = 75$  (meaning the 76th connection would cause an additional instance to be started)

## Troubleshooting

When should you worry about tuning your applications? You should consider tuning your applications if:

1. Your application has several requests that are slow and you have enough concurrent usage that people's expectations for responsiveness aren't being met. For example, If your response time for some key calculations takes one second and you would like to make sure that the average response time for your application is less than two seconds, you will not want more than two concurrent requests per worker.
  - Possible Diagnosis: The application performance might be due to R's single threaded nature. Spreading the load across additional workers should alleviate the issue.
  - Remedy: Consider lowering the maximum number of connections per worker, and possibly increasing the maximum number of workers. Also consider adding additional Application Instances and aggressively scaling them by tweaking the Instance Load Factor to a lower percentage.
2. Sudden large spikes of traffic have poor performance even though you have configured multiple Application Instances. However, additional new users have good performance.
  - Possible Diagnosis: The number of workers within the first container are insufficient for the initial spike of traffic. When the additional containers are started, new users are routed to the new Application Instance.
  - Remedy: Decrease the Instance Load Factor which will aggressively start up additional Application Instances and spread the load.
3. Your application suddenly goes grey and you see in your logs that the application was "killed".
  - Possible Diagnosis: Each Application Instance has a size which corresponds to the amount of RAM (memory) that is allocated to it. If the amount of memory allocated to this application is exceeded, then the Application Instance could be shut down by shinyapps.
  - Remedy: There are two possible solutions:
    - Increase the size of the Application Instance.
    - Decrease the number of workers per Application Instance. Since each worker takes up additional

RAM, you may find that lowering the "Max worker processes" to two or one would help keep each Application Instance's memory usage down.

4. An application isn't fitting in memory even for the largest Application Instance size
  - Possible Diagnosis: If the application loads correctly with one or two users interacting with it, then it is possible that your data set sizes on a per worker basis are too big.
  - Remedy: Decrease the number of workers per Application Instance.
5. Your application stops accepting additional users beyond 150 connections.
  - Possible Diagnosis: It is likely that you have reached the limit on the number of connections that can be served by the default settings in an Application Instance.
  - Remedy: A few things to try would be:
    - Increase the allowed connections per worker by changing Connections setting for the application.
    - Increase the number of workers per Application Instance.
    - Add additional Application Instances.
6. An application that has a significant initialization time (loading lots of data, or talking to 3rd party web services) sometimes doesn't load.
  - Possible diagnosis: Shinyapps.io has an "Instance Startup Timeout" which will stop an application if it is not responsive within that period of time at startup.
  - Remedy: Increase the timeout on the Application Settings page.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

romain

---

Hello,

I am trying to optimize my Shinyapp. As of now it is slow to load and refresh because I am running a large clustering in the background. I need to increase the CPU but I don't know how to do so.

I have tried several methods exposed above but nothing really seems to improve the response time.

If you have any tips, thank you for sharing.

R



## 2.20 Share data across sessions with

# Share data across sessions with shinyapps.io

ADDED: 29 SEP 2014

BY: JEFF ALLEN

ShinyApps.io expects your applications to be portable across servers. It uses this feature to maintain a smooth user experience. For example, shinyapps.io will scale up your application by adding more running instances, or shinyapps.io will keep your application available by starting your application on a server that has sufficient resources to host it. Both of these features will fail if your app is not portable.

One of the consequences of this design is that files written to the file system will be deleted when your application shuts down or moves to another server. These transitions shouldn't happen while a user's session is active, so you can safely write temporary files associated with a particular session to the filesystem and expect to read these files back in during the same session. However, local files created by your application will not persist beyond the session in which they are created. If you want to access files after the session has closed, you will need to design your application so that it saves its temporary files to an external service.

This article will introduce three ways to save—and use—data over multiple sessions. You can:

1. Store arbitrary data of any size in an object storage system—just like you would with a filesystem
2. Store semi-structured data in an easily scalable NoSQL database
3. Store rigidly structured data in a formal relational database like MySQL, SQL Server, or PostgreSQL.

We will not trace out each step required to set up these storage services, but we will provide enough context to help you determine which options would be most appropriate for your application. We will then point you to the appropriate resources that will help you store your data.

Regardless of which option you choose, we recommend using a “hosted” solution to store your data if you don't already have a server capable of the appropriate option. Amazon Web Services products can support the first and last options. AWS provides `S3` object storage and `RDS` for a relational database. R has packages to support NoSQL servers like MongoDB. One hosted provider for MongoDB is [MongoLab](#).

We'll walk through an example application that collects names and comments from users. We want to accrue all comments across user sessions. We'll begin with a functioning app that relies on local file storage, available here

<https://gist.github.com/trestletech/3679b34c5a83f521387b>

Note that this code is not compatible with ShinyApps, since the modified copy of `log.txt` would be overwritten with the original version of `log.txt` every time the application is stopped or moved. The next three sections will present three ways to make the application portable for work on ShinyApps.

## 1. Object Storage (Amazon S3)

Object storage is the simplest of the three alternatives and works the most like a traditional filesystem. You can put an “object” (file) into the store at a particular location, update it as often as you'd like, and then retrieve it from that

This option allows users to continue to use a local file in their code assuming they

1. retrieve the most updated file when the application starts, and
2. write the updates to the file back to their object storage system before the application closes and the changes are lost.

You can do this every time the data changes, or every time a session exits ( `session$onEnded` ).

We'll update our comment example from above using the `RAmazonS3` package which can be installed using the following command:

```
install.packages("RAmazonS3", repos = "http://www.omegahat.org/R")
```

You can view the updated demo [here](#). In short, rather than reading in the log from a file when the application starts, we'll read in the object from S3. And rather than writing out to a file, we'll write to S3. That way we can be sure that, when the process closes, our updated log has been written somewhere where it will persist and can be retrieved later.

One important caveat is that this code is not safe to use in multiple processes simultaneously, so you should be sure to configure your application to have a maximum of 1 process if you're using this approach. As an example, if you started two processes, both would read the up-to-date log file from S3 at startup. They would then write back to S3 any time a comment was posted without ever updating from S3 to capture each other's updates—meaning that they would just overwrite each other as long as they were both running.

This problem is solved in more formal database systems as will be discussed below using the concept of “transactions” which can ensure that only one update occurs to a data set at a time.

## 2. NoSQL Storage (MongoDB, etc.)

NoSQL databases offer a flexible storage system that offers a bit more control than object storage, but without the overhead of a formal relational database. NoSQL is an appropriate option if your data can easily be divided into individual “entries” that can be created, updated, and deleted and there aren't strong relationships between those entities.

There are a couple of R packages that will enable you to communicate with MongoDB, one of which is `rmongodb`, but you can choose the one that is right for your project.

Adapting our example to use MongoDB storage would be fairly simple. Each comment could be treated as a separate record (or “document,” in mongoDB parlance) with two fields: `name` and `comment` (and perhaps the `date-time` if it was posted). To add a comment, we might use `mongo.insert`. To query the list of comments initially we could use `mongo.find`.

Because we're using a proper database, the information can be maintained more granularly which allows us to get around the problems we had previously with concurrency. If multiple processes call `mongo.insert` simultaneously, both records will get inserted into the data store without any data loss. One possible enhancement to our application would be to periodically query the MongoDB database to ensure that we're showing any new comments that have been written into the database by other processes.

## 3. Relational Database (MySQL, PostgreSQL, etc. with Amazon RDS)

The final option for storage would be to use a formal relational database like MySQL or PostgreSQL. Most major databases have associated R packages to ease interaction from R (see `RPostgreSQL` or `RMySQL`).

Relational databases excel when data is highly structured and there are well-defined relationships between entities.

However, relational databases are often the most cumbersome of the above options and require the most know-how to setup and manage. If you do not have any prior experience with relational databases or know someone who does, you may be better served starting with one of the other options.

The architecture of our sample application using a relational database would be very similar to the solution described for a NoSQL server—run an `INSERT` when a new comment is posted and a `SELECT` to retrieve the existing comments from the database. Most relational databases include a feature called “transactions” which allow you to guarantee data accuracy when performing complex operations on your data (like querying a bank account balance, then updating it). If you find that you’re encountering such problems with NoSQL or an object store, it may be worth the learning curve to get acquainted with a relational database like PostgreSQL.

## Recap

To summarize, shinyapps.io expects your apps to be portable, which means that your app cannot pass data from session to session with local files. Instead, your app should save data to an external source that future sessions can access. You can do this by saving different types of data to different storage services.

1. Save arbitrary data of any size in an object storage system like Amazon S3
2. Save semi-structured data in an easily scalable NoSQL database like MongoDB
3. Save rigidly structured data in a formal relational database like MySQL, SQL Server, or PostgreSQL.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Jen Underwood

---

I'd love to see a demo with a database, ggplot and possibly integrate/pass variable parameters like start date and end date from a web page. Basically putting all the elements together with web app integration. To take it one step further, passing authentication or credentials to the cloud app for integration scenarios. Thanks.

pcavatore

---

It would be great to have a tutorial documenting option 3 in more details considering the comments from <https://groups.google.com/foru...> using Amazon RDS

Dean Attali

---

There is a new article that's a follow-up to this one that documents option 3 with a few different databases, but not Amazon RDS specifically. Hopefully it's still useful.

<http://shiny.rstudio.com/artic...>

pcavatore  
comments powered by Disqus

---

## 2.21 Migrating shinyapps.io authentication

# Migrating shinyapps.io authentication

ADDED: 07 JAN 2015

The general release of shinyapps.io introduces a new mechanism for authentication and authorization. This system replaces the existing rscript based approach and provides a more flexible and manageable flow.

The new authentication system provides several advantages. With it you can:

- \* Add or remove authorized users without restarting the application thereby preserving the sessions of logged in users.
- \* Manage application access through the admin interface (new)
- \* Leverage Google or Github authentication to improve security for your users.
- \* Save your users the burden of managing and maintaining their own user authentication information.

To migrate your application from the old authentication system to the new one you will need to follow these steps:

1. Set the Application Visibility setting to Private in the Users tab for that application and click Save Settings. This will restart the application and apply the new setting. Note, once you do this, none of the existing users will be able to authenticate.
2. On your local system, rename the `passwords.txt` file in `/shinyapps` to `old_passwords.txt`.
3. Re-deploy your application using `shinyapps::deployApp()`
4. In the Users tab, add the email addresses for the individuals that were in your `old_passwords.txt` file. If you were not using email addresses before, you will need to do so at this time. Don't worry if your users don't have Google or GitHub accounts, they can always use local authentication through shinyapps.io.
5. Your users should now be able to authenticate and see your application.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

## 2.22 Introduction to Shiny Server

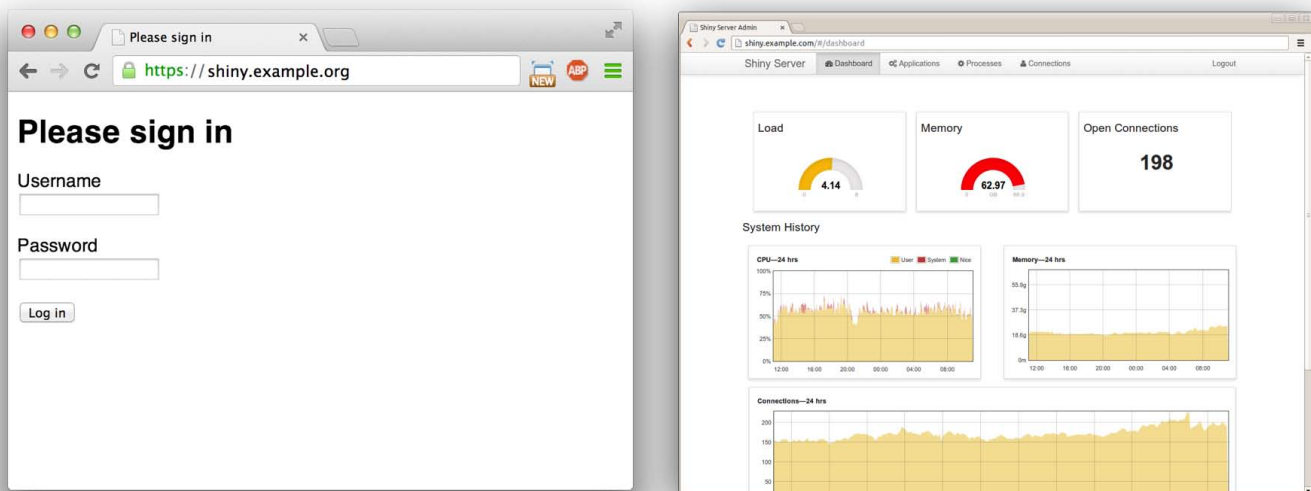
# Introduction to Shiny Server

ADDED: 25 FEB 2014

BY: JEFF ALLEN

Shiny Server is a back end program that makes a big difference. It builds a web server specifically designed to host Shiny apps. With Shiny Server you can host your apps in a controlled environment, like inside your organization, so your Shiny app (and whatever data it needs) will never leave your control. You can also use Shiny Server to make your apps available across the Internet when you choose. Shiny Server will host each app at its own web address and automatically start the app when a user visits the address. When the user leaves, Shiny Server will automatically stop the app.

The professional version of Shiny Server offers even more features. With Shiny Server Pro, you can password protect your apps and use an administrative dashboard to learn who is using your apps and how (below).



This article will demonstrate some of the features of Shiny Server and introduce you to the deep literature that is waiting to help you download, install, and configure your own Shiny Server.

## Shiny Server, Shiny Server Pro, and shinyapps.io

You can use the free Community Edition of Shiny Server to begin hosting your Shiny applications, or you can leverage Shiny Server Professional to scale your applications to a broader audience, restrict access to particular applications, or control the resources consumed by your Shiny applications. You can see a full breakdown of the differences between the two editions [here](#); we'll discuss the features of both editions of Shiny Server in this article. We hope to highlight some features of Shiny Server here, but for a full discussion on how to manage and configure your server, please see the official [Admin Guide](#).

Shiny Server runs on a variety of Linux distributions. If you're not comfortable with Linux or would prefer to have someone else manage the server on which your Shiny applications are hosted, check out [ShinyApps.io](#) to learn about

hosting your applications in an environment that is managed and maintained by RStudio for you. ShinyApps.io lets you use a pay-as-you-go model to tap into some of the features only available in Shiny Server Professional, an arrangement that is more approachable for some organizations.

Shiny Server Professional offers a variety of nice features that build on top of the open source Shiny Server including:

- A dashboard to help understand the activity on your server (as shown above)
- The ability to secure your Shiny applications using SSL (HTTPS)
- The ability to control which users are allowed to access which applications
- Priority support from RStudio
- Controls to fine-tune resource consumption per Shiny application

Below we'll demonstrate a few examples of using Shiny Server to make your Shiny applications available in different ways.

## Host a Directory of Applications

(See [this page](#) for a complete step-by-step walkthrough of this example.)

Shiny Server allows you to host a directory full of Shiny applications and other web assets (HTML files, CSS files, etc.) using the `site_dir` configuration. By default, Shiny Server will use a `site_dir` to make any applications and assets stored in `/srv/shiny-server/` available. You can begin placing Shiny applications inside this directory then referencing them on your server. For instance, a Shiny application stored in `/srv/shiny-server/myApp` would be available at `http://myserver.org:3838/myApp`, (where `myserver.org` is the name of your server) by default. You could also place HTML files in this directory to make them available on your server, as well.

## Let Users Manage Their Own Applications

(See [this page](#) for a complete step-by-step walkthrough of this example.)

In some cases, it may be desirable to allow users on a system to manage and update their own Shiny applications stored in their home directories; the `user_apps` configuration allows you to do just that. Shiny applications hosted inside users' `ShinyApps` directory will be available online. For instance, a user who stored a Shiny application in `/home/kim/ShinyApps/myApp` would be able to access it at `http://myserver.org:3838/kim/myApp` on a server configured to use `user_apps`.

## Require User Authentication On An Application

(See [this page](#) for a complete step-by-step walkthrough of this example.)

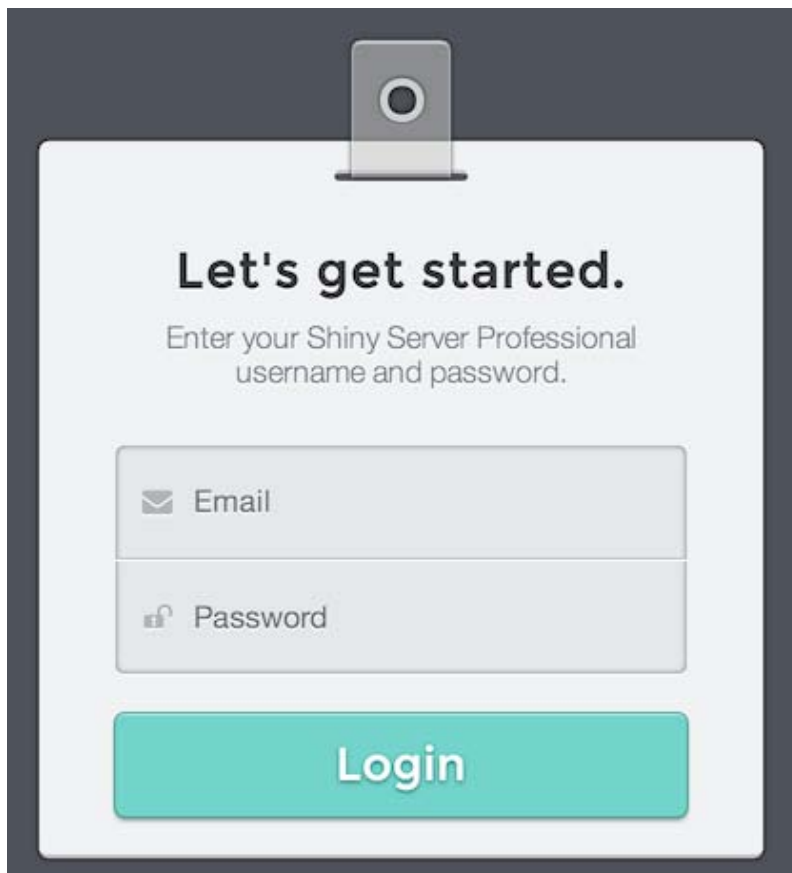
Shiny Server Professional supports various forms of user authentication which can be used to require your users to login before being able to access particular Shiny applications on your server.

As of Shiny Server v1.1.0, you can even change the appearance of the login page using the `template_dir` configuration to make your login page look something like this:

A login form titled "Login to Shiny Server" set against a blue background. It features two input fields: "Username or Email" and "Password". A "Login" button is positioned at the bottom right of the form.

(Inspired by Thibaut Courouble and Orman Clark.)

Or this:

A login form titled "Let's get started." presented as a clipboard with a silver clip at the top. The text "Enter your Shiny Server Professional username and password." is centered below the title. There are two input fields: "Email" with an envelope icon and "Password" with a lock icon. A large teal "Login" button is at the bottom.

(Inspired by Ionut Zamfir.)

## Summary

To recap, Shiny Server is a useful way to share your Shiny applications in a controlled environment. It lets you

- Automatically start and stop your applications as needed on a Linux server
- Provide a unique URL for each application
- Restrict access to particular applications, if using Shiny Server Professional

If you have any questions or would like more information, please visit <http://www.rstudio.com/shiny/server/> or email us at [sales@rstudio.com](mailto:sales@rstudio.com).

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Yerviz

---

This is more their way of making you pay if you want a viable production server :P

Thomas

---

I don't consider SSL to be a Professional feature, but rather a basic requirement for most any realized web services.

comments powered by Disqus



## 2.23 Save your app as a function

# Save your app as a function

ADDED: 11 JUL 2014

BY: GARRETT GROLEMUND

Shiny apps are a great way to build interactive data analysis tools, but they require a bit of set up to use. Or do they? This article will show you how to write R functions that launch Shiny apps.

The result is a quick tool that you can reuse on any data set (or with any set of arguments that you like). Moreover, you can use this method to share your Shiny apps as straightforward R functions.

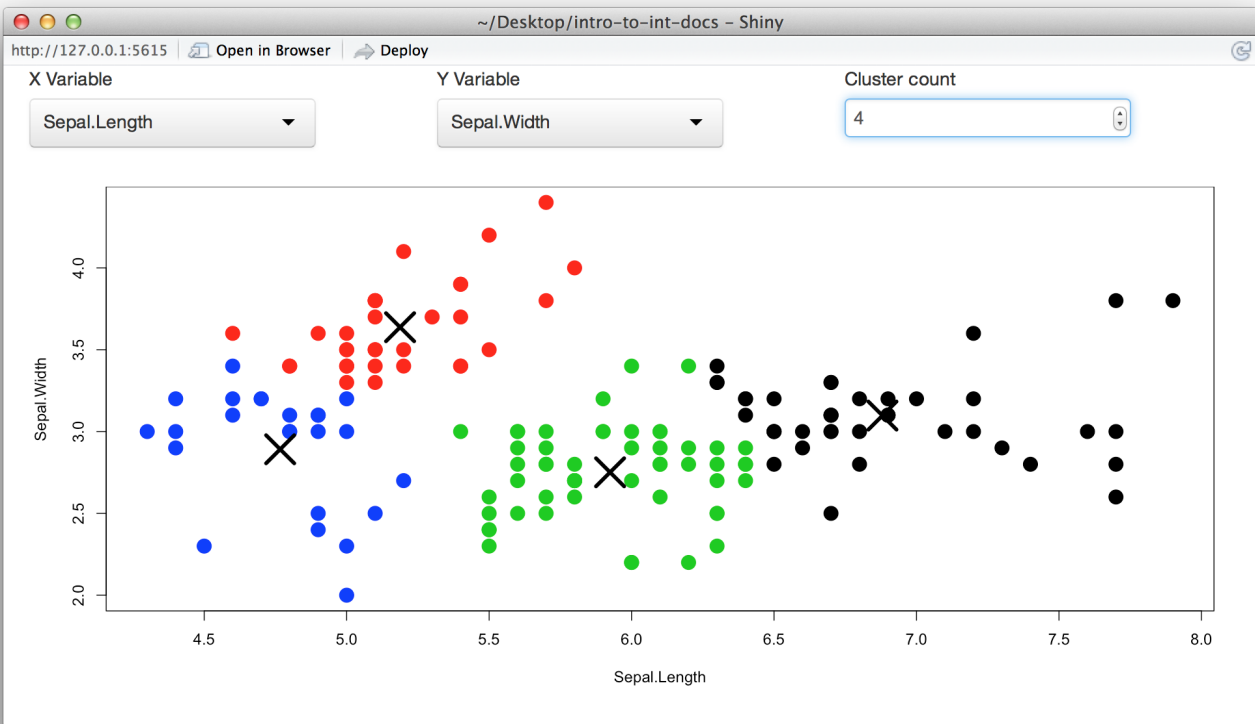
## An example

`rmexamples::kmeans_cluster` is a function that launches a Shiny app. It takes one argument, a data frame, and launches a cluster analysis app that explores the data frame.

You can install the `rmexamples` library from github. To do so, run

```
devtools::install_github("rmexamples", "rstudio")
```

```
library(rmexamples)  
kmeans_cluster(iris)
```



The following sections will show you how to

1. Define a Shiny app in a single script with `shinyApp`
2. Save your app as a parameterized function
3. Launch your app from the command line or inside an interactive document

## shinyApp

`shinyApp` is a function that builds Shiny apps. You can use `shinyApp` to define a complete app in a single R script, or even at the command line.

`shinyApp` builds an app from two arguments that parallel the structure of a standard Shiny app. The `ui` argument takes code that builds the user interface for your app, and the `server` argument takes code that sets up the server for your Shiny app.

When you build a standard Shiny app, you save two files in your working directory and then call `runApp()`. One file, named `ui.R`, contains a call to `shinyUI`. The second file, named `server.R`, contains a call to `shinyServer`.

```
# ui.R

shinyUI(fluidPage(
  sidebarLayout(
    sidebarPanel(sliderInput("n",
      "Bins", 5, 100, 20)),
    mainPanel(plotOutput("hist"))
  )
))
```

```
# server.R

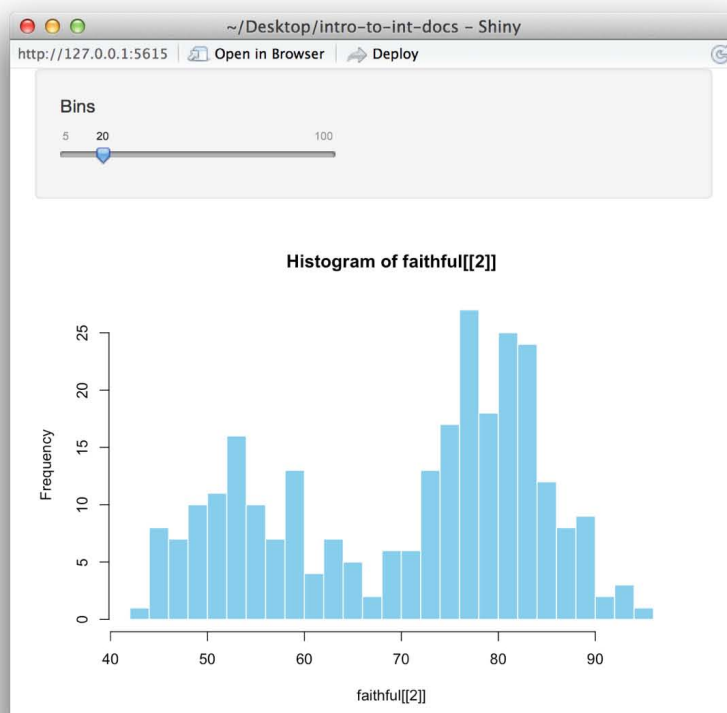
shinyServer(function(input, output) {
  output$hist <- renderPlot(
    hist(faithful[[2]], breaks = input$n,
      col = "skyblue", border = "white")
  )
})
```

To build an app with `shinyApp`, give `ui` the code that you would normally pass to `shinyUI` in a `ui.R` file. Then give `server` the code that you would normally pass to `shinyServer` in a `server.R` file.

```
shinyApp(
  ui = fluidPage(
    sidebarLayout(
      sidebarPanel(sliderInput("n", "Bins", 5, 100, 20)),
      mainPanel(plotOutput("hist"))
    )
  ),
  server = function(input, output) {
    output$hist <- renderPlot(
      hist(faithful[[2]], breaks = input$n,
        col = "skyblue", border = "white")
    )
  }
)
```

R will build and launch your app when you run the complete `shinyApp` call at the command line.

For example, the code above will build a minimal Shiny app. When you run the code, your app will look like this.



`shinyApp` also uses an `option` argument, which takes a list of named options. You can use the `option` argument to set any options that you would normally set in a `runApp` call.

In addition, you can use the `option` argument to provide a hint to the browser environment about the ideal height and width of your Shiny app. This becomes very useful when you [embed Shiny apps in R Markdown documents](#) (described below). The code below will launch an app that has a suggested size of 600 by 500 pixels.

```
shinyApp(
  ui = fluidPage(
    sidebarLayout(
      sidebarPanel(sliderInput("n", "Bins", 5, 100, 20)),
      mainPanel(plotOutput("hist"))
    )
  ),
  server = function(input, output) {
    output$hist <- renderPlot(
      hist(faithful[[2]], breaks = input$n,
          col = "skyblue", border = "white")
    )
  },
  options = list(height = 600, width = 500)
)
```

## Save your app as a function

To turn your app into a function, write a function that calls `shinyApp`. Parameterize your app by passing function arguments to `shinyApp`.

The code below saves the histogram app as a function named `binner` that takes a vector named `var`. The function passes its `var` argument to `shinyApp`, which then launches an app that visualizes `var`.

```

binner <- function(var) {
  require(shiny)
  shinyApp(
    ui = fluidPage(
      sidebarLayout(
        sidebarPanel(sliderInput("n", "Bins", 5, 100, 20)),
        mainPanel(plotOutput("hist"))
      )
    ),
    server = function(input, output) {
      output$hist <- renderPlot(
        hist(var, breaks = input$n,
             col = "skyblue", border = "white")
      )
    }
  )
}

```

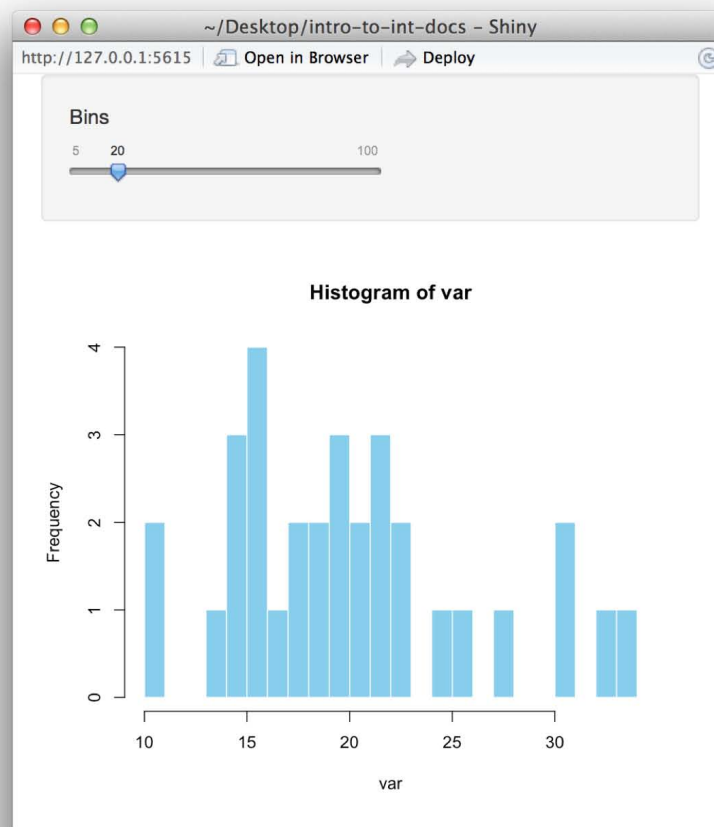
Since `var` is a function argument, you can supply it at runtime. This means you can reuse the app on many different data frames.

## Call your app as a function

You can launch your app from the command line once you define your function. To do this, call the function and supply a value for each function argument.

For example, you can now use `binner` to explore various vectors.

```
binner(faithful$eruptions)
```



## Embed your app in an interactive document

You can also embed your app in [R Markdown](#) reports. Define the function that launches your app in an [R code chunk](#) (by including the definition, or loading a package that has the function). Then call the function.

The R Markdown script below places `binner` in an interactive document.

```
---
runtime: shiny
output: html_document
---

```{r echo = FALSE}
binner <- function(var) {
  require(shiny)
  shinyApp(
    ui = fluidPage(
      sidebarLayout(
        sidebarPanel(sliderInput("n", "Bins", 5, 100, 20)),
        mainPanel(plotOutput("hist"))
      )
    ),
    server = function(input, output) {
      output$hist <- renderPlot(
        hist(var, breaks = input$n,
            col = "skyblue", border = "white")
      )
    }
  )
}
```

```
    }  
  )  
}  
````br/>## Old Faithful
```

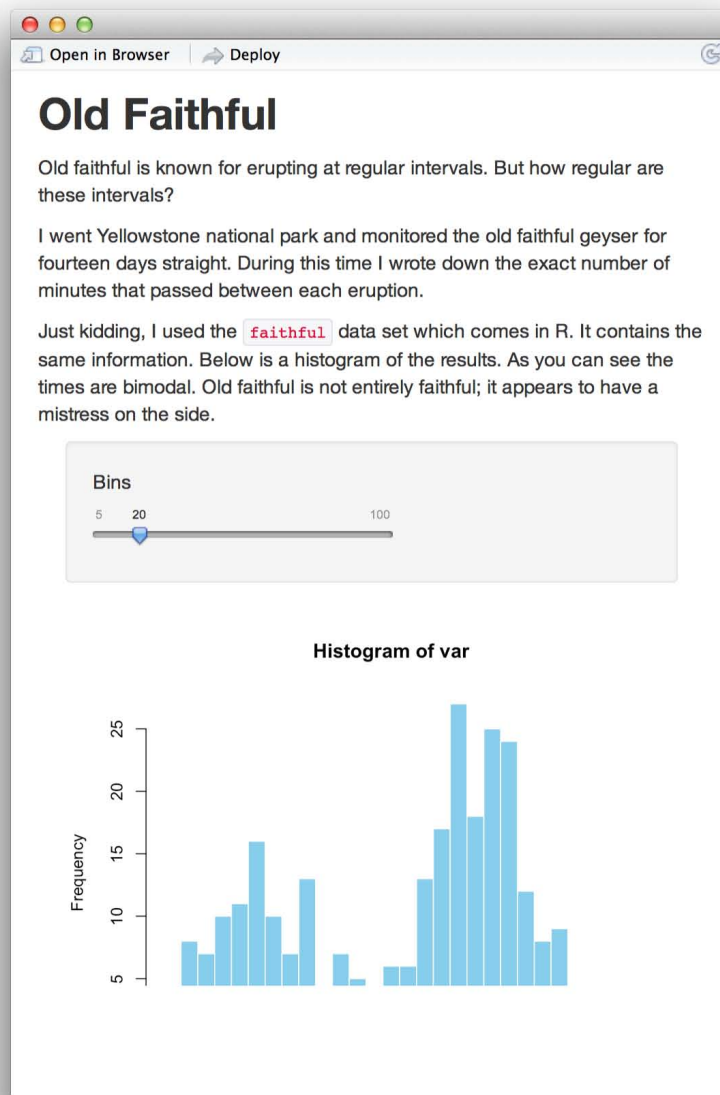
Old faithful is known for erupting at regular intervals. But how regular are these intervals?

I went Yellowstone national park and monitored the old faithful geyser for fourteen days straight. During this time I wrote down the exact number of minutes that passed between each eruption.

Just kidding. I used the `faithful` data set which comes in R. It contains the same information. Below is a histogram of the results. As you can see the times are bimodal. Old faithful is not entirely faithful; it appears to have a mistress on the side.

```
````{r echo = FALSE}  
binner(faithful$waiting)  
````
```

The document looks like this when you render the report. The binner app is interactive in the final document.



## Recap

You can package your Shiny apps as parameterized R functions. To do this, define a function that builds your app with `shinyApp`.

Why would you save your Shiny apps this way? Saving your app as a function opens several opportunities

- Easy to share - you can share your apps with other R users just as you would share functions. For example, you can put your apps in an R package very easily.
- Reusable - it is very easy to reuse your apps on new data sets, or with new conditions, by parameterizing your apps.
- A Chance to be a Hero - you can create and share apps that any R user can use. Your users do not need to know Shiny; they only need to call a function. You can use this method to make packages that contain interactive data exploration tools, teaching examples and quizzes, gui versions of R programs, and much more.



you'd like to get specific help, we recommend the [Shiny Discussion Forum](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Thomas Rosendal

---

Is it possible to deploy the app in a function with shiny server? We have an app that we currently deploy using shiny server that I would like to place inside an R package. I would like to avoid having parallel code ie. having both the server.R and ui.R as well as the functions in the package that do the same thing.

jwarrick

---

This was a great suggestion. However, it doesn't seem to always work. When I call my shiny app function from the commandline, it works, but when I source a file that calls the shiny app function, it doesn't. Suggestions?

ImAndy

---

how do I add, say, `library(dplyr)` in the above code, if the app requires additional manipulation?

spencer

---

comments powered by [Disqus](#)

## 2.24 Sharing apps to run locally

# Sharing apps to run locally

ADDED: 06 JAN 2014

Once you've written your Shiny app, you can distribute it for others to run on their own computers—they can download and run Shiny apps with a single R command. This requires that they have R and Shiny installed on their computers.

If you want your Shiny app to be accessible over the web, so that users only need a web browser, see

- [Introduction to Shiny Server](#) (to host your own apps), or
- [Getting started with shinyapps.io](#) (to use RStudio's hosting service)

Here are some ways to deliver Shiny apps to run locally:

## Gist

One easy way is to put your code on [gist.github.com](https://gist.github.com), a code pasteboard service from [GitHub](#). Both `server.R` and `ui.R` must be included in the same gist, and you must use their proper filenames. See <https://gist.github.com/jcheng5/3239667> for an example.

Your recipient must have R and the Shiny package installed, and then running the app is as easy as entering the following command:

```
shiny : runGist (' 3239667' )
```

In place of `' 3239667'` you will use your gist's ID; or, you can use the entire URL of the gist (e.g. `' https://gist.github.com/jcheng5/3239667'` ).

## Pros

- Source code is easily visible by recipient (if desired)
- Easy to run (for R users)
- Easy to post and update

## Cons

- Code is published to a third-party server

## GitHub repository

If your project is stored in a git repository on GitHub, then others can download and run your app directly. An example repository is at [https://github.com/rstudio/shiny\\_example](https://github.com/rstudio/shiny_example). The following command will download and run the application:

```
shiny : runGitHub (' shiny_example', ' rstudio' )
```

In this example, the GitHub account is 'rstudio' and the repository is 'shiny\_example'; you will need to replace them with your account and repository name.

## Pros

- Source code is easily visible by recipient (if desired)
- Easy to run (for R users)
- Very easy to update if you already use GitHub for your project
- Git-savvy users can clone and fork your repository

## Cons

- Developer must know how to use git and GitHub
- Code is hosted by a third-party server

## Zip File, delivered over the web

If you store a zip or tar file of your project on a web or FTP server, users can download and run it with a command like this:

```
runUrl('https://github.com/rstudio/shiny_example/archive/master.zip')
```

The URL in this case is a zip file that happens to be stored on GitHub; replace it with the URL to your zip file.

## Pros

- Only requires a web server for delivery

## Cons

- To view the source, recipient must first download and unzip it

## Zip File, copied to recipient's computer

Another way is to simply zip up your project directory and send it to your recipient(s), where they can unzip the file and run it the same way you do (shiny::runApp).

## Pros

- Share apps using e-mail, USB flash drive, or any other way you can transfer a file

## Cons

- Updates to app must be sent manually

## Package

If your Shiny app is useful to a broader audience, it might be worth the effort to turn it into an R package. Put your Shiny application directory under the package's `inst` directory, then create and export a function that contains something like this:

```
shiny::runApp(system.file('appdir', package='packagename'))
```

where `appdir` is the name of your app's subdirectory in `inst`, and `packagename` is the name of your package.

## Pros

- Publishable on CRAN
- Easy to run (for R users)

## Cons

- More work to set up
- Source code is visible by recipient (if not desired)

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

cmd

---

You can also just type runApp() in the code folder

Guest

---

hello,  
haw to host my to install shiny server on Windows 7 ?

Thanks in advance

matt

---

Introduction to Shiny Server (to host your own apps), or  
link broken

Garrett

---

Thanks. Fixed now.

comments powered by [Disqus](#)

## 2.25 Introduction to R Markdown

# Introduction to R Markdown

ADDED: 16 JUL 2014

BY: GARRETT GROLEMUND

Interactive documents are a new way to build Shiny apps. An interactive document is an [R Markdown](#) file that contains Shiny widgets and outputs. You write the report in [markdown](#), and then launch it as an app with the click of a button.

This article will show you how to write an R Markdown report.

The companion article, [Introduction to interactive documents](#), will show you how to turn an R Markdown report into an interactive document with Shiny components.

## R Markdown

R Markdown is a file format for making dynamic documents with R. An R Markdown document is written in [markdown](#) (an easy-to-write plain text format) and contains chunks of embedded R code, like the document below.

```
---  
output: html_document  
---
```

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
```${r}  
summary(cars)  
```
```

You can also embed plots, for example:

```
```${r, echo=FALSE}  
plot(cars)  
```
```

Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.

R Markdown files are designed to be used with the `rmarkdown` package. `rmarkdown` comes installed with the RStudio IDE, but you can acquire your own copy of `rmarkdown` from [github](#) with the command

```
devtools::install_github("rmarkdown", "rstudio")
```

R Markdown files are the source code for rich, reproducible documents. You can transform an R Markdown file in two ways.

1. *knit* - You can *knit* the file. The `rmarkdown` package will call the `knitr` package. `knitr` will run each chunk of R code in the document and append the results of the code to the document next to the code chunk. This workflow saves time and facilitates reproducible reports.

Consider how authors typically include graphs (or tables, or numbers) in a report. The author makes the graph, saves it as a file, and then copy and pastes it into the final report. This process relies on manual labor. If the data changes, the author must repeat the entire process to update the graph.

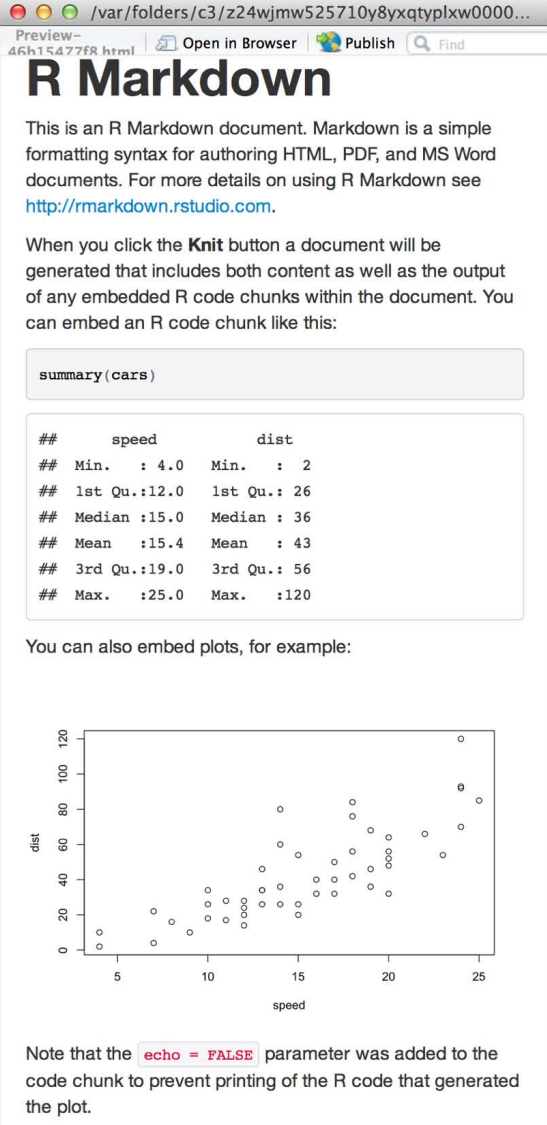
In the R Markdown paradigm, each report contains the code it needs to make its own graphs, tables, numbers, etc. The author can automatically update the report by re-knitting.

2. *convert* - You can *convert* the file. The `rmarkdown` package will use the `pandoc` program to transform the file into a new format. For example, you can convert your `.Rmd` file into an HTML, PDF, or Microsoft Word file. You can even turn the file into an HTML5 or PDF slideshow. `rmarkdown` will preserve the text, code results, and formatting contained in your original `.Rmd` file.

Conversion lets you do your original work in markdown, which is very easy to use. You can include R code to knit, and you can share your document in a variety of formats.

In practice, authors almost always knit and convert their documents at the same time. In this article, I will use the term *render* to refer to the two step process of knitting and converting an R Markdown file.

You can manually render an R Markdown file with `rmarkdown::render()`. This is what the above document looks like when rendered as a HTML file.



Preview-  
46h15472fR.html Open in Browser Publish Find

# R Markdown

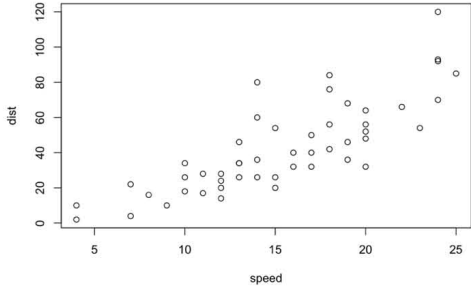
This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

```
##      speed      dist
## Min.   : 4.0   Min.   :  2
## 1st Qu.:12.0   1st Qu.: 26
## Median :15.0   Median : 36
## Mean   :15.4   Mean   : 43
## 3rd Qu.:19.0   3rd Qu.: 56
## Max.   :25.0   Max.   :120
```

You can also embed plots, for example:



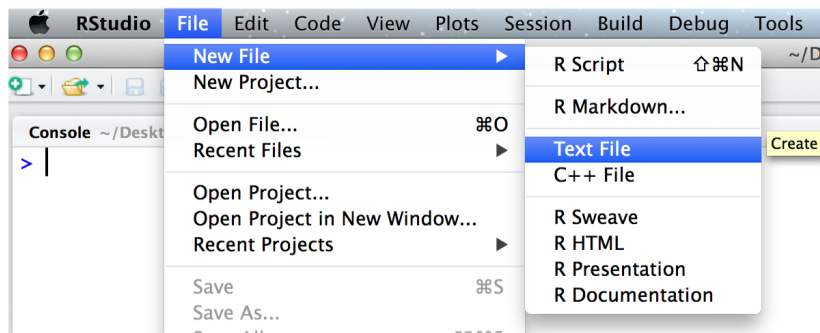
```
##      echo = FALSE
```

Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.

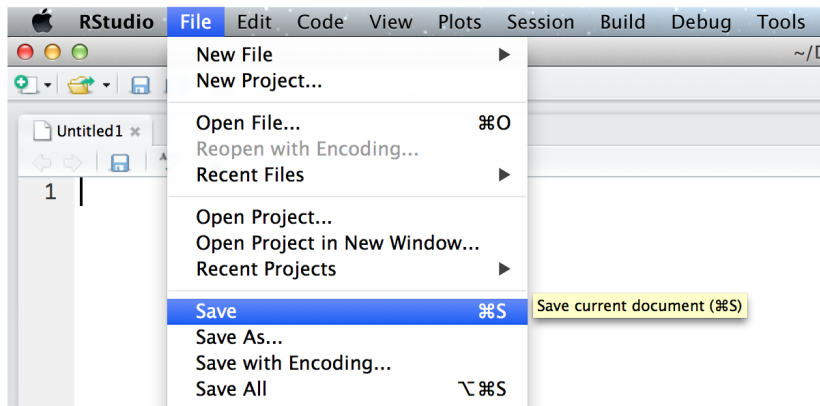
In practice, you do not need to call `rmarkdown::render()`. You can use a button in the RStudio IDE to render your report. R Markdown is heavily integrated into the RStudio IDE.

## Getting started

To create an R Markdown report, open a plain text file and save it with the extension `.Rmd`. You can open a plain text file in your scripts editor by clicking `File > New File > Text File` in the RStudio toolbar.



Be sure to save the file with the extension `.Rmd`. The RStudio IDE enables several helpful buttons when you save the file with the `.Rmd` extension. You can save your file by clicking `File > Save` in the RStudio toolbar.



R Markdown reports rely on three frameworks

1. `markdown` for formatted text
2. `kni tr` for embedded R code
3. `YAML` for render parameters

The sections below describe each framework.

## Markdown for formatted text

`.Rmd` files are meant to contain text written in [markdown](#). Markdown is a set of conventions for formatting plain text. You can use markdown to indicate

- bold and italic text
- lists
- headers (e.g., section titles)
- hyperlinks
- and much more

The conventions of markdown are very unobtrusive, which make Markdown files easy to read. The file below uses several of the most useful markdown conventions.

```
# Say Hello to markdown
```



Markdown is an **easy to use** format for writing reports. It resembles what you naturally write every time you compose an email. In fact, you may have already used markdown **without realizing it**. These websites all rely on markdown formatting

- \* [\[Github\]](http://www.github.com) ([www.github.com](http://www.github.com))
- \* [\[StackOverflow\]](http://www.stackoverflow.com) ([www.stackoverflow.com](http://www.stackoverflow.com))
- \* [\[Reddit\]](http://www.reddit.com) ([www.reddit.com](http://www.reddit.com))

The file demonstrates how to use markdown to indicate:

1. headers - Place one or more hashtags at the start of a line that will be a header (or sub-header). For example, `# Say Hello to markdown`. A single hashtag creates a first level header. Two hashtags, `##`, creates a second level header, and so on.
2. italicized and bold text - Surround italicized text with asterisks, like this `*without realizing it*`. Surround bold text with two asterisks, like this `**easy to use**`.
3. lists - Group lines into bullet points that begin with asterisks. Leave a blank line before the first bullet, like this

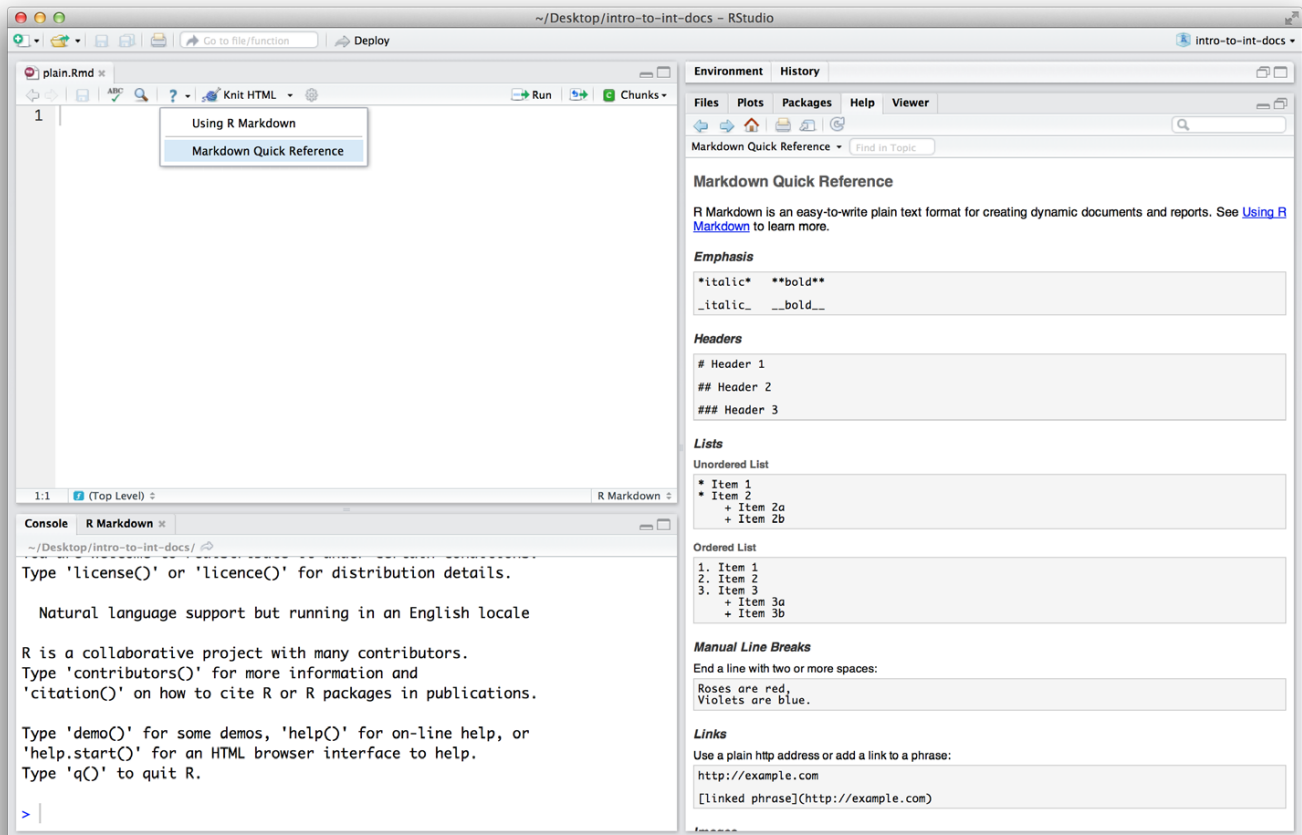
`This is a list`

- `* item 1`
- `* item 2`
- `* item 3`

4. hyperlinks - Surround links with brackets, and then provide the link target in parentheses, like this `[Github]` ([www.github.com](http://www.github.com)) .

You can learn about more of markdown's conventions in the *Markdown Quick Reference* guide, which comes with the RStudio IDE.

To access the guide, open a `.md` or `.Rmd` file in RStudio. Then click the question mark that appears at the top of the scripts pane. Next, select "Markdown Quick Reference". RStudio will open the *Markdown Quick Reference* guide in the Help pane.



## Rendering

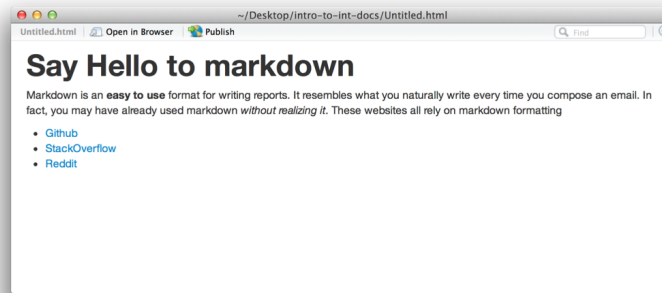
To transform your markdown file into an HTML, PDF, or Word document, click the “Knit” icon that appears above your file in the scripts editor. A drop down menu will let you select the type of output that you want.



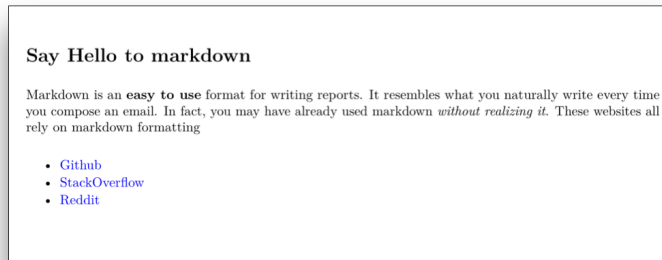
When you click the button, `rmarkdown` will duplicate your text in the new file format. `rmarkdown` will use the formatting instructions that you provided with markdown syntax.

Once the file is rendered, RStudio will show you a preview of the new output and save the output file in your working directory.

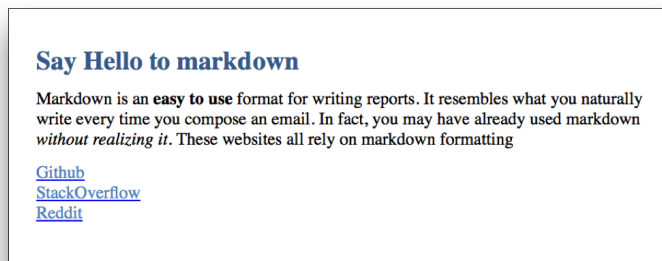
Here is how the markdown script above would look in each output format.



HTML



PDF



MS Word

*Note: RStudio does not build PDF and Word documents from scratch. You will need to have a distribution of Latex installed on your computer to make PDFs and Microsoft Word (or a similar program) installed to make Word files.*

## knitr for embedded R code

The `knitr` package extends the basic markdown syntax to include chunks of executable R code.

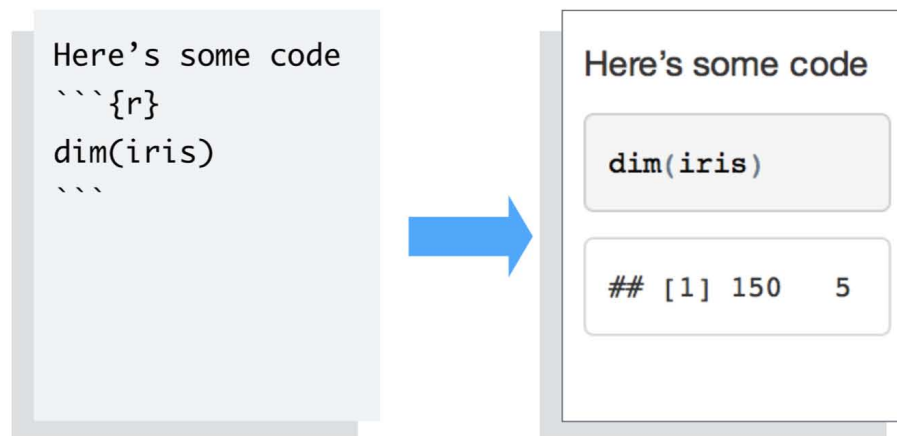
When you render the report, `knitr` will run the code and add the results to the output file. You can have the output display just the code, just the results, or both.

To embed a chunk of R code into your report, surround the code with two lines that each contain three backticks. After the first set of backticks, include `{r}`, which alerts `knitr` that you have included a chunk of R code. The result will look like this

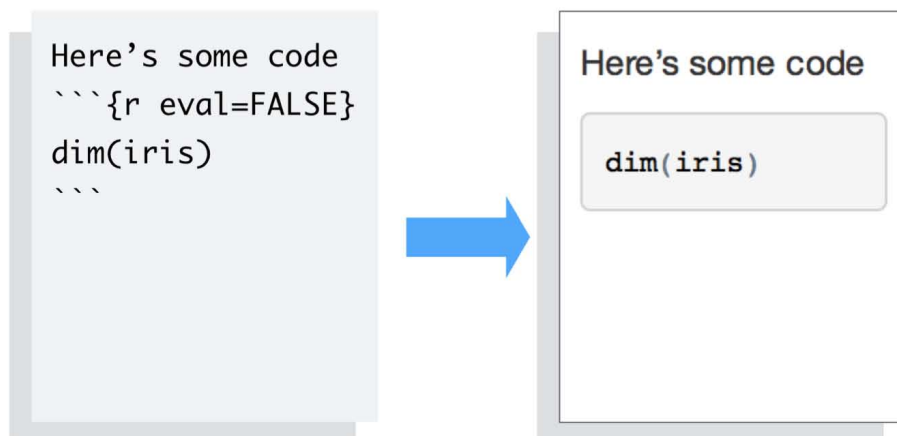
```
Here's some code
```{r}
dim(iris)
```
```

When you render your document, `knitr` will run the code and append the results to the code chunk. `knitr` will provide formatting and syntax highlighting to both the code and its results (where appropriate).

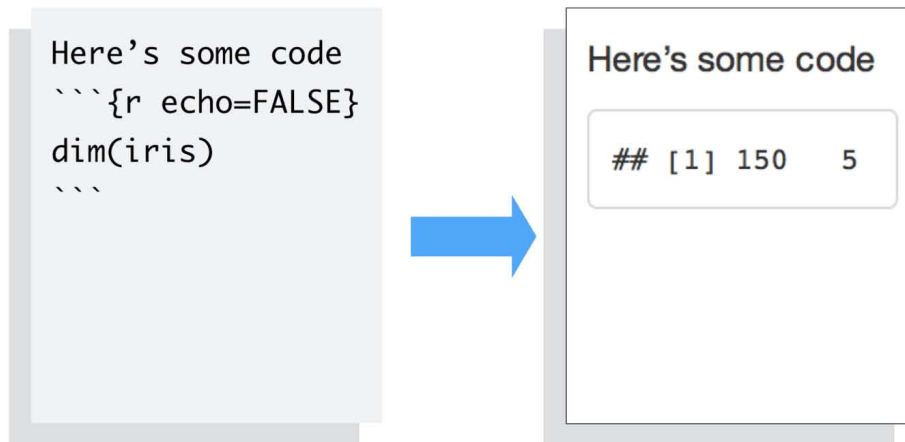
As a result, the markdown snippet above will look like this when rendered (to HTML).



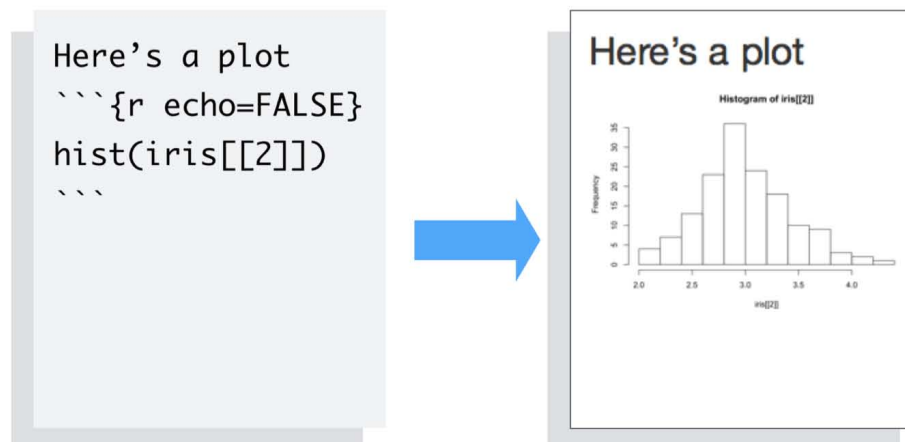
To omit the *results* from your final report (and not run the code) add the argument `eval = FALSE` inside the brackets and after `r`. This will place a copy of your code into the report.



To omit the *code* from the final report (while including the results) add the argument `echo = FALSE`. This will place a copy of the results into your report.



`echo = FALSE` is very handy for adding plots to a report, since you usually do not want to see the code that generates the plot.



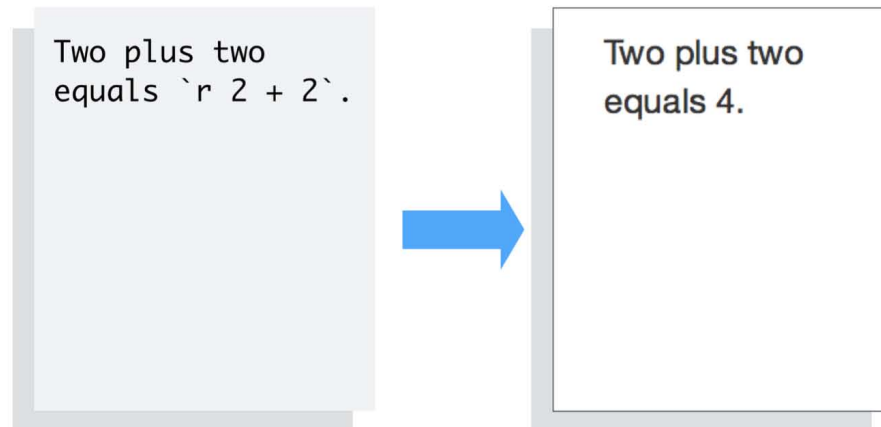
`echo` and `eval` are not the only arguments that you can use to customize code chunks. You can learn more about formatting the output of code chunks at the [markdown](#) and [knitr](#) websites.

## Inline code

To embed R code in a line of text, surround the code with a pair of backticks and the letter `r`, like this.

Two plus two equals ``r 2 + 2``.

`knitr` will replace the inline code with its result in your final document (inline code is *always* replaced by its result). The result will appear as if it were part of the original text. For example, the snippet above will appear like this:



## YAML for render parameters

You can use a YAML header to control how `rmarkdown` renders your `.Rmd` file. A YAML header is a section of `key: value` pairs surrounded by `---` marks, like below

```
---
title: "Untitled"
author: "Garrett"
date: "July 10, 2014"
output: html_document
---
```

Some inline R code, ``r 2 + 2``.

The `output:` value determines what type of output to convert the file into when you call `rmarkdown::render()`.  
*Note: you do not need to specify `output:` if you render your file with the RStudio IDE knit button.*

`output:` recognizes the following values:

- `html_document`, which will create HTML output (default)
- `pdf_document`, which will create PDF output
- `word_document`, which will create Word output

If you use the RStudio IDE knit button to render your file, the selection you make in the gui will override the `output:` setting.

## Slideshows

You can also use the `output:` value to render your document as a slideshow.

- `output: ioslides_presentation` will create an ioslides (HTML5) slideshow
- `output: beamer_presentation` will create a beamer (PDF) slideshow

*Note: The knit button in the RStudio IDE will update to show slideshow options when you include one of the above output values and save your `.Rmd` file.*

`rmarkdown` will convert your document into a slideshow by starting a new slide at each header or horizontal rule (e.g., `***`).

Visit [rmarkdown.rstudio.com](http://rmarkdown.rstudio.com) to learn about more YAML options that control the render process.

## Recap

R Markdown documents provide quick, reproducible reporting from R. You write your document in markdown and embed executable R code chunks with the `kni tr` syntax.

You can update your document at any time by re-knitting the code chunks.

You can then convert your document into several common formats.

R Markdown documents implement Donald's Knuth's idea of literate programming and take the manual labor out of writing and maintaining reports. Moreover, they are quick to learn. You already know enough about markdown, knitr, and YAML to begin writing your own R Markdown reports.

In the next article, [Introduction to interactive documents](#), you will learn how to add interactive Shiny components to an R Markdown report. This creates a quick workflow for writing light-weight Shiny apps.

To learn more about R Markdown and interactive documents, please visit [rmarkdown.rstudio.com](http://rmarkdown.rstudio.com).

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Lee Zamparo

---

Great tutorial, but you're missing a ':' in the installation code block:

```
...
```

```
devtools:install_github("rmarkdown", "rstudio")
```

```
...
```

should be

```
...
```

```
devtools::install_github("rmarkdown", "rstudio")
```

```
...
```

## 2.26 Introduction to interactive documents

# Introduction to interactive documents

ADDED: 09 JUL 2014

BY: GARRETT GROLEMUND

Interactive documents are a new way to build Shiny apps. An interactive document is an [R Markdown](#) file that contains Shiny widgets and outputs. You write the report in [markdown](#), and then launch it as an app with the click of a button.

## R Markdown

The previous article, [Introduction to R Markdown](#), described how to write R Markdown files. R Markdown files are useful because

- They are quick and easy to write.
- You can embed executable R code into your file, which saves manual labor and creates a reproducible report.
- You can convert R Markdown files into HTML, PDF, and Word documents with the click of a button.
- You can convert R Markdown files into ioslides and beamer slideshows with the click of a button.

In fact, R Markdown files are the ultimate R reporting tool.

This article will show you one more thing that R Markdown files can do: you can embed Shiny components in an R Markdown file to create an interactive report or slideshow.

Your report will be a complete Shiny app. In fact, R Markdown provides the easiest way to build light-weight Shiny apps. I will refer to apps that combine Shiny with R Markdown as *interactive documents*.

## Interactive documents

You can make an R Markdown document interactive in two steps:

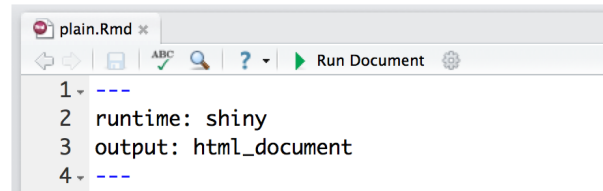
1. add `runtime: shiny` to the document's YAML header.
2. add Shiny widgets and Shiny render functions to the file's R code chunks

The `rmarkdown` package will compile your document into a reactive Shiny app. The document will look just as it would otherwise, but it will include reactive components.

### runtime: shiny

Notify `rmarkdown` that your file contains Shiny components by adding `runtime: shiny` to the file's YAML header. RStudio will change its "Knit" icon to a "Run Document" icon when you save this change.





“Run Document” is a cue that `rmarkdown` will no longer compile your document into a static file. Instead it will “run” the document as a live Shiny app.

Since the document is a Shiny app, you must render it into an HTML format. Do this by selecting either `html_document` or `ioslides_presentation` for your final output.

## Widgets

To add a widget to your document, call a Shiny widget function in an R code chunk. R Markdown will add the widget to the code chunk’s output.

For example, the file below creates an HTML document with two widgets.

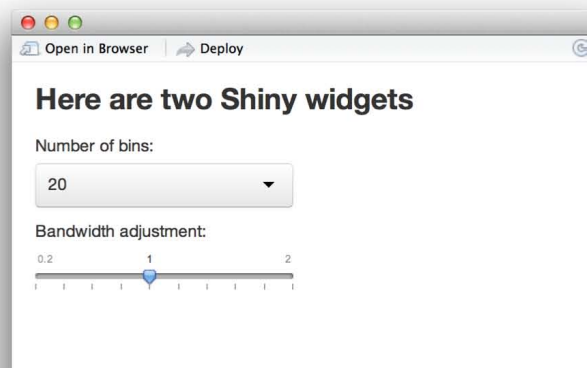
```
---
runtime: shiny
output: html_document
---

### Here are two Shiny widgets

```{r echo = FALSE}
selectInput("n_breaks", label = "Number of bins:",
            choices = c(10, 20, 35, 50), selected = 20)

sliderInput("bw_adjust", label = "Bandwidth adjustment:",
            min = 0.2, max = 2, value = 1, step = 0.2)
```
```

The document looks like this when rendered. (*This is a static image of the output, the actual widgets are “live”; you can manipulate them*).



## Rendered output

To add reactive output to your document, call one of the `render*` functions below in an R code chunk.

render function creates

|                          |                                                 |
|--------------------------|-------------------------------------------------|
| <code>renderImage</code> | images (saved as a link to a source file)       |
| <code>renderPlot</code>  | plots                                           |
| <code>renderPrint</code> | any printed output                              |
| <code>renderTable</code> | data frame, matrix, other table like structures |
| <code>renderText</code>  | character strings                               |
| <code>renderUI</code>    | a Shiny tag object or HTML                      |

R Markdown will include the rendered output in the result of the code chunk.

This output will behave like rendered output in a standard Shiny app. The output will automatically update whenever you change a widget value or a reactive expression that it depends on.

The file below uses `renderPlot` to insert a histogram that reacts to the two widgets.

```
---
runtime: shiny
output: html_document
---

### Here are two Shiny widgets

```{r echo = FALSE}
selectInput("n_breaks", label = "Number of bins:",
            choices = c(10, 20, 35, 50), selected = 20)

sliderInput("bw_adjust", label = "Bandwidth adjustment:",
            min = 0.2, max = 2, value = 1, step = 0.2)
...

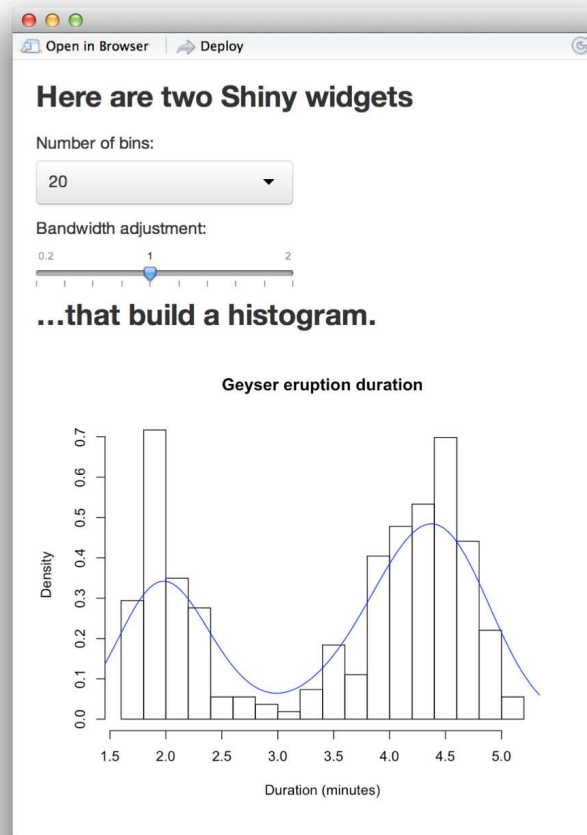
### ...that build a histogram.

```{r echo = FALSE}
renderPlot({
  hist(faithful$eruptions, probability = TRUE, breaks = as.numeric(input$n_breaks),
       xlab = "Duration (minutes)", main = "Geyser eruption duration")

  dens <- density(faithful$eruptions, adjust = input$bw_adjust)
  lines(dens, col = "blue")
})
...

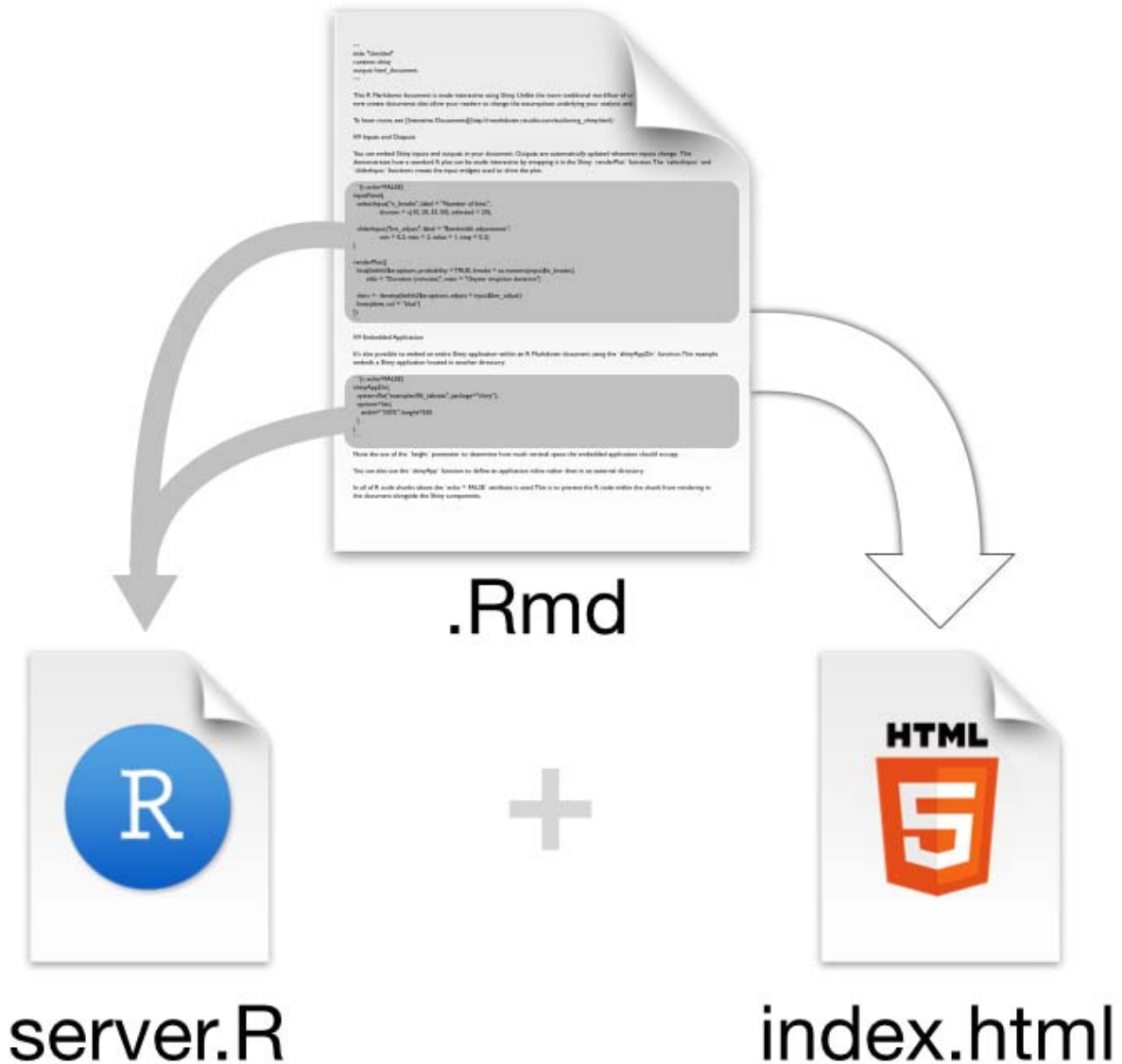
```

The document creates the app below when you click “Run Document.”



## The structure of an interactive document

When you run an interactive document, `rmarkdown` extracts the code in your code chunks and places them into a pseudo server.R file. R Markdown uses the html output of the markdown file as an `index.html` file to place the reactive elements into.



As a result, outputs in one code chunk can use widgets and reactive expressions that occur in other code chunks. Since the R Markdown document provides a layout for the app, you do not need to write a `ui.R` file.

## Sharing interactive documents

Interactive documents are a type of Shiny app, which means that you can share them in the same way that you share other Shiny apps. You can

1. Email a `.Rmd` file to a colleague. He or she can run the file locally by opening the file and clicking “Run Document”
2. Host the document with [Shiny Server](#) or [Shiny Server Pro](#)
3. Host the document at [ShinyApps.io](#)

*Note: If you are familiar with R Markdown, you might expect RStudio to save an HTML version of an interactive document in your working directory. However, this only works with static HTML documents. Each interactive document must be served by a computer that manages the document. As a result, interactive documents cannot be shared as a standalone HTML file.*

## Conclusion

Interactive documents provide a new and easy way to make Shiny apps.

Interactive documents will not replace standard Shiny apps since they cannot provide the design options that come with a `ui.R` or `index.html` file. However, interactive documents do create some easy wins:

- The R Markdown workflow makes it easy to build light-weight apps. You do not need to worry about laying out your app or building an HTML user interface for the app.
- You can use R Markdown to create interactive slideshows, something that is difficult to do with Shiny alone. To create a slideshow, change `output: html_document` to `output: ioslides_presentation` in the YAML front matter of your `.Rmd` file. R Markdown will divide your document into slides when you click “Run Document.” A new slide will begin whenever a header or horizontal rule ( `***` ) appears.
- Interactive documents enhance the existing R Markdown workflow. R Markdown makes it easy to write literate programs and reproducible reports. You can make these reports even more effective by adding Shiny to the mix.

To learn more about R Markdown and interactive documents, please visit [rmarkdown.rstudio.com](http://rmarkdown.rstudio.com).

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

gjthompson1

---

Is it a pipe dream to have an interactive document with shiny widgets embedded?

Behrooz Mirmolavi

---

It would be truly valuable to have an interactive document with shiny widgets embedded. Else R Markdown is just a different way of deploying a Shiny App? And really it looks like Shiny Apps are more customisable.

Sudipta Banerjee

---

I have installed R shiny server on my Ubuntu OS. How can I use markdown to write my first program

Yahia

---

I tried it. It is really nice. But for some reason caching doesn't work when I choose `runtime: shiny`. Any fix for that problem?

comments powered by [Disqus](#)

## 2.27 R Markdown integration in the

# R Markdown integration in the RStudio IDE

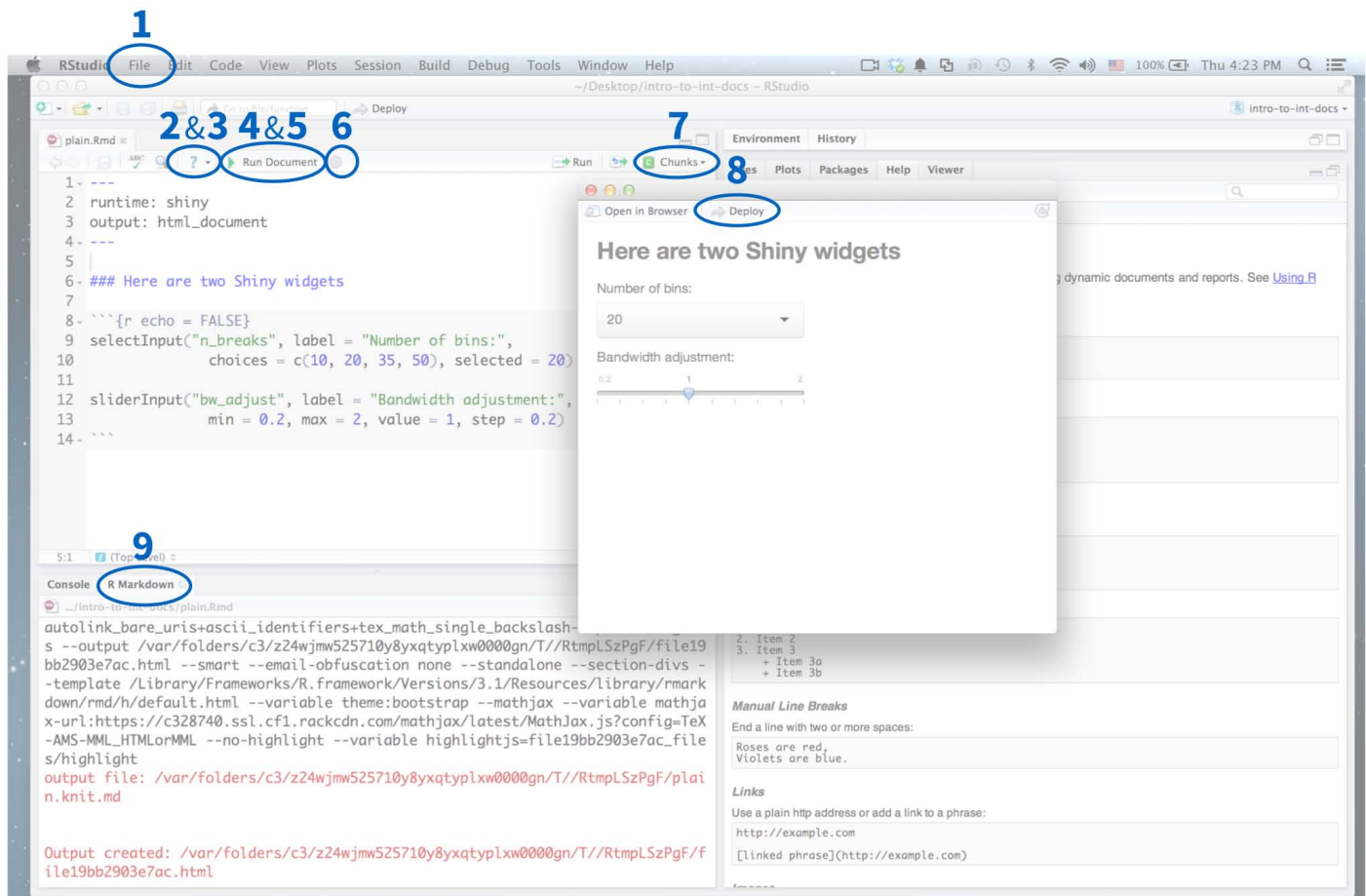
ADDED: 10 JUL 2014

BY: GARRETT GROLEMUND

[Introduction to interactive documents](#) describes how to use R Markdown to build light-weight Shiny apps that are easy to assemble.

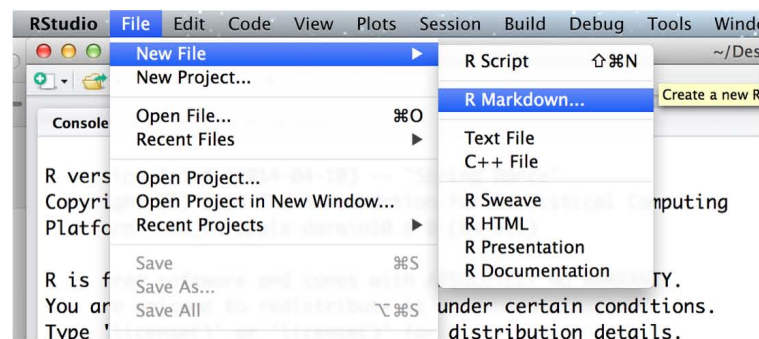
The RStudio IDE contains many features that make it easy to write and run interactive documents. This article will highlight some of the most useful:

1. File templates
2. *Using R Markdown*
3. *Markdown Quick Reference*
4. The Run Document button
5. The Viewer Pane
6. Document options
7. Insert Chunk
8. Deploy to shinyapps.io
9. The R Markdown console

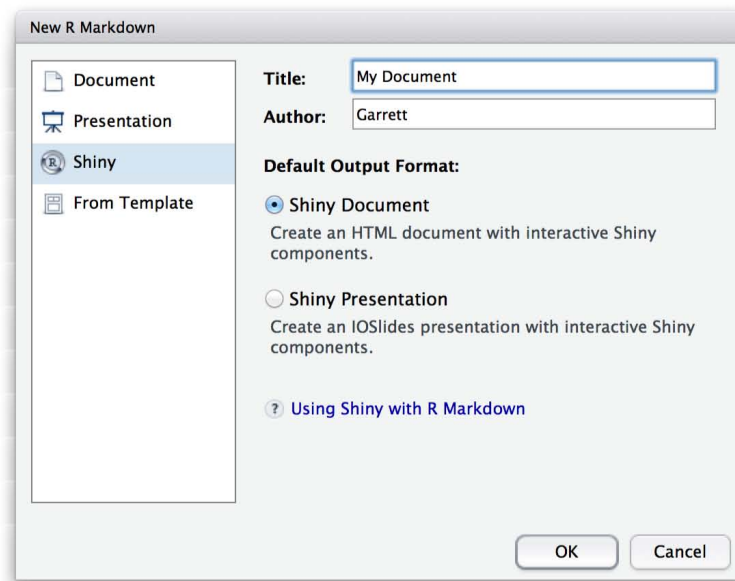


# 1. File templates

The RStudio IDE provides a template document when you open a new .Rmd file. To open a new file, click File > New File > R Markdown in the RStudio menu bar.



A window will pop up that helps you build the YAML frontmatter for the .Rmd file.



From the window's sidebar, select the category of output that you plan to convert your .Rmd file into. You can select

- Document - a static document
- Presentation - an ioslides or beamer slideshow
- Shiny - an interactive document
- From Template - a format that you have pre-saved as a template (if you have one)

Use the radio buttons to select the specific type of output that you wish to build. Your options will depend on the category you selected in the sidebar.

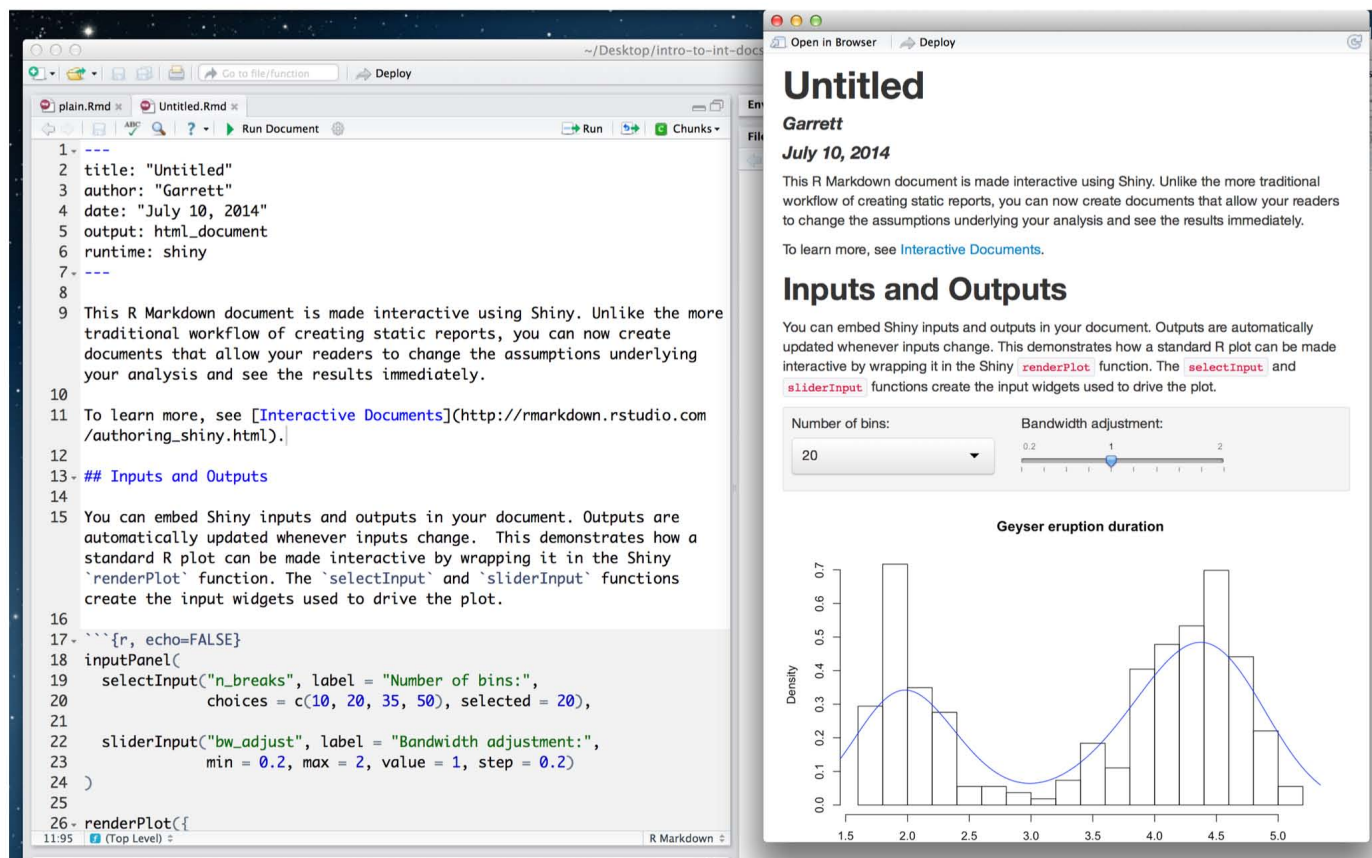
You can also use the window to give your file a title and author field.

To make an interactive document, select Shiny from the sidebar and Shiny Document from the radio buttons. Then click OK.

RStudio will open a new .Rmd file for you to use. The file will contain a YAML header that includes all of the parameters that your file will need to correctly render with `rmarkdown::render()`. You can manually change these parameters afterwards if you like.

RStudio will fill the rest of the file with a template that demonstrates the basic features of .Rmd files. The templates work right out of the box, which means that you can immediately knit or run one. The image below shows the template for interactive documents.

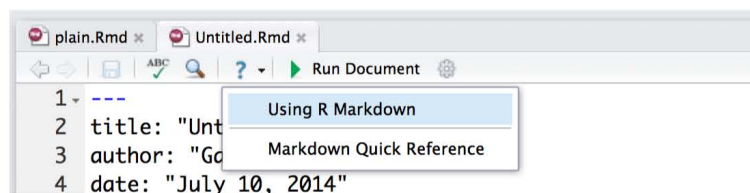




Study the template as a refresher on R Markdown, or erase it and begin writing your own document.

## 2. Using R Markdown

The IDE places a question mark icon in the scripts pane whenever you open a .Rmd file. The question mark opens a drop down menu with two helpful resources.



The first option, “Using R Markdown,” opens the development website for the `rmarkdown` package, [rmarkdown.rstudio.com](http://rmarkdown.rstudio.com). Here you can look up the many useful features of R Markdown.

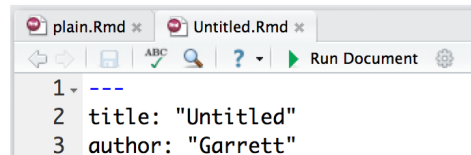
## 3. Markdown Quick Reference

The second link, “Markdown Quick Reference,” opens a reference guide to the markdown syntax. This guide will appear in the help pane of the RStudio IDE.

The guide uses examples to explain the different formatting options of markdown. It is like a markdown cheatsheet that is built right in to the RStudio IDE.

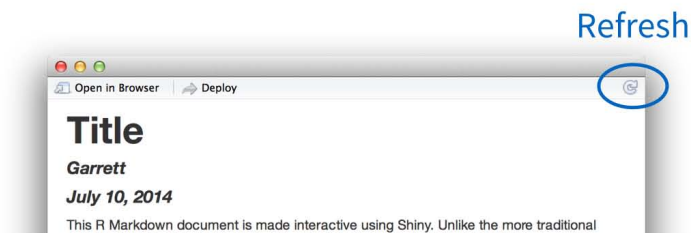
## 4. The Run Document button

If your `.Rmd` file contains `runtime: shiny` in its YAML header, the RStudio IDE will display a “Run Document” button at the top of the scripts pane.



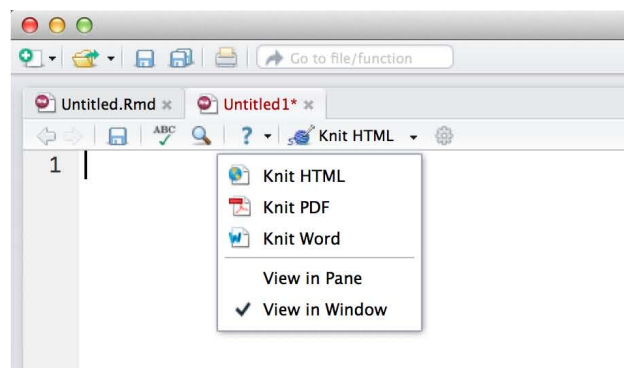
The “Run Document” button is a shortcut for the `rmarkdown::render` command. It lets you quickly render your `.Rmd` file into an interactive document hosted locally on your computer. The RStudio IDE will display your document in a preview window.

You can edit the `.Rmd` file while the preview is running. To see your changes, save the `.Rmd` file. Then click the refresh icon in the top left corner of the preview window.



If your `.Rmd` file does not contain `runtime: shiny`, the RStudio IDE will display a “Knit HTML” button in place of the “Run Document” button. The “Knit HTML” button works in the same way. It renders your `.Rmd` file and launches a preview of your output document.

The Knit HTML button contains a dropdown menu that lets you choose which type of output to knit your file into (this will override the output type specified in your file’s YAML header).

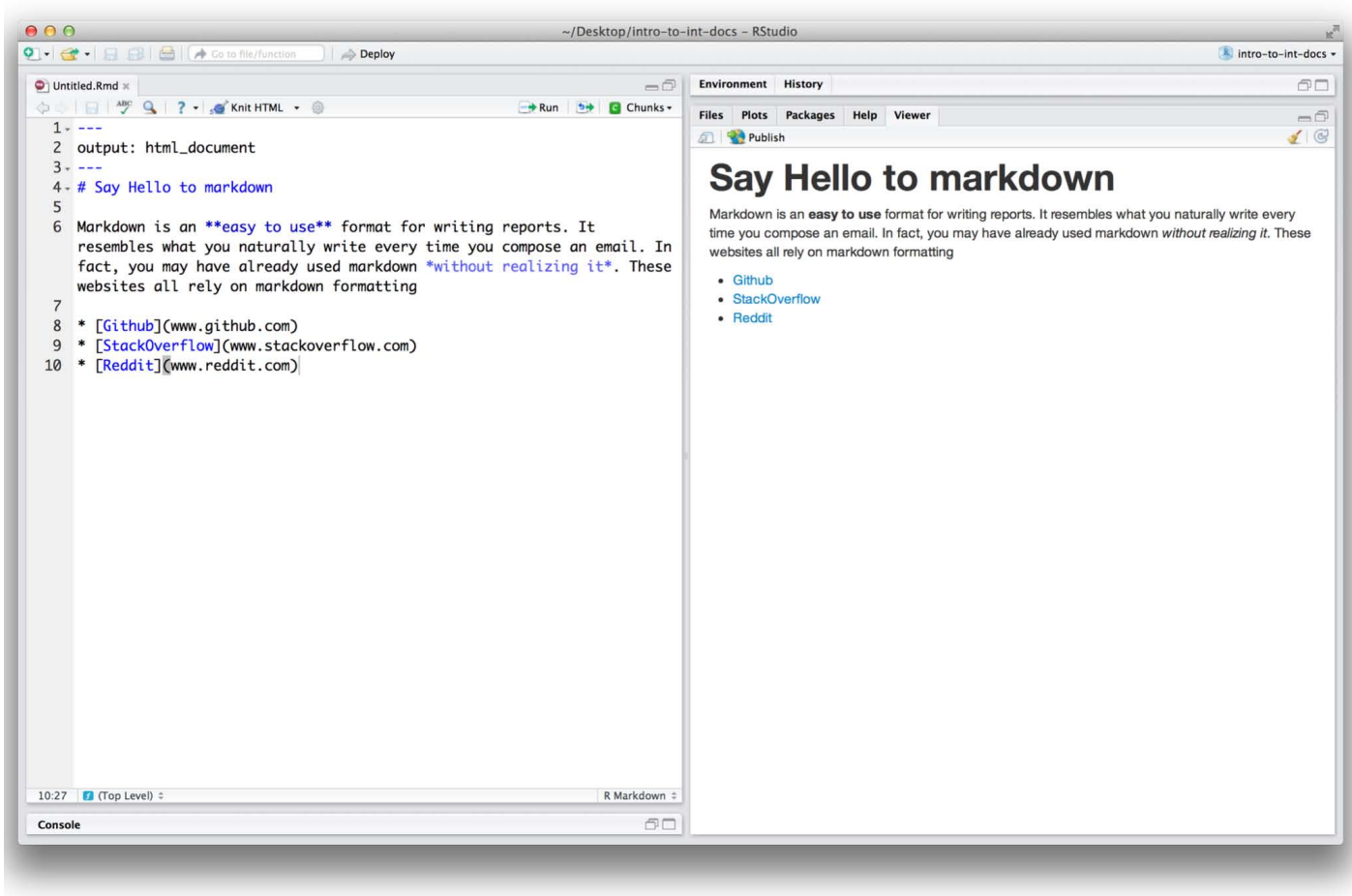


## 5. Viewer Pane

By default, the RStudio IDE opens a preview window to display the output of your `.Rmd` file. However, you can choose to display the output in a dedicated viewer pane.

To do this, select “View in Pane” from the drop down menu that appears when you click on the “Run Document” button (or “Knit HTML” button).

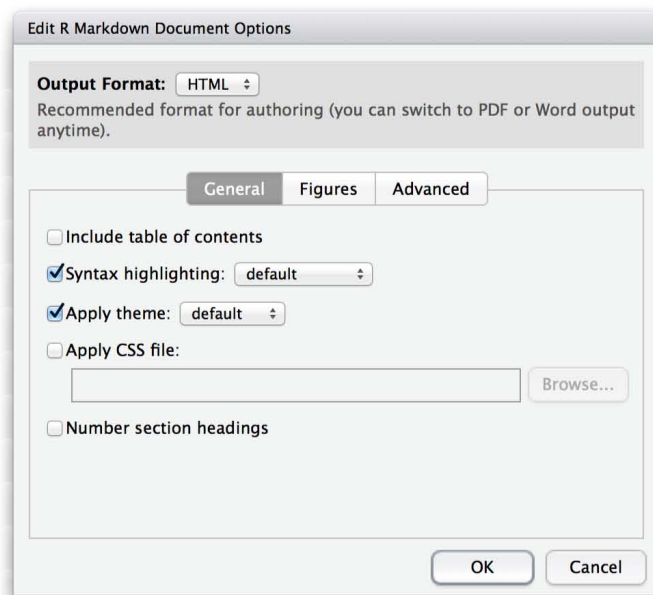
The viewer pane provides a side-by-side view that resembles some text and Latex editors.



## 6. Document options

The gear icon beside “Run Document” opens a wizard that lets you customize your interactive document. You can use this wizard to

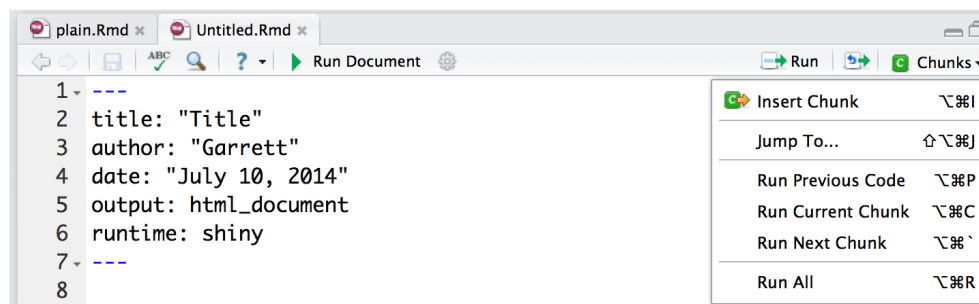
- Include a table of contents
- Apply syntax highlighting to code chunks
- Apply one of eight built in bootstrap CSS themes to your document
- Link to your own custom CSS file to style your document
- Number section headings
- Size figures and add captions, and
- Tweak the render process



Set the features you like, and the RStudio IDE will apply them when you click “Run Document”.

## 7. Insert Chunk

The Chunks button in the top left corner of the Scripts pane opens a dropdown menu that you can use to manage code chunks in your .Rmd file.

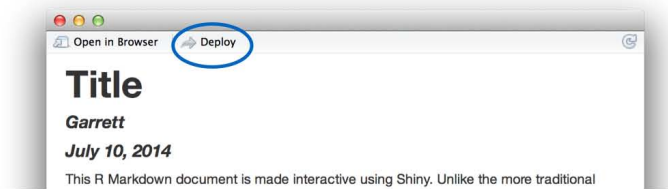


The first option in the menu is the most useful. “Insert Chunk” will insert a blank code chunk into your .Rmd file at the location of your cursor. You can then fill this chunk with code.

You can use basic RStudio tab completion to write arguments inside the `{r}` braces at the top of each code chunk.

## 8. Deploy to shinyapps.io

If you’ve set up the `shinyapps` package as described in [Getting started with shinyapps.io](#), the RStudio IDE will place a deploy button at the top of your interactive document’s preview window.



You can click this button to deploy your document directly to your shinyapps.io account. Shinyapps.io will host the document at its own web URL for people to visit.

## 9. The R Markdown console

When you render a .Rmd file, the RStudio IDE opens a second console pane that displays R Markdown output. This pane shows the status of the render process and displays any errors or warnings that occur while rendering your document. If your document is an interactive document, the pane will also display errors that occur while you navigate the app.

This extra pane keeps your original R console clean and uncluttered.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

glenn.vdl

---

Step 8 indicates there should be a deploy button. For me this isn't the case (even after adding 'runtime: shiny' to the header. What could be causing this?

# 2.28 The R Markdown Cheat sheet

# The R Markdown Cheat sheet


ADDED: 01 AUG 2014  
BY: GARRETT GROLEMUND

The R Markdown cheat sheet is a quick reference guide for writing reports with R Markdown.

Download  Notify me when you make new cheat sheets


## R Markdown Cheat Sheet

learn more at [rmarkdown.rstudio.com](http://rmarkdown.rstudio.com)  
rmarkdown 0.2.50 Updated: 8/14




**1. Workflow** R Markdown is a format for writing reproducible, dynamic reports with R. Use it to embed R code and results into slideshows, pdfs, html documents, Word files and more. To make a report:


**i. Open** - Open a file that uses the .Rmd extension



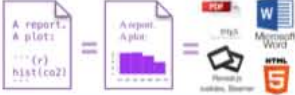
**ii. Write** - Write content with the easy to use R Markdown syntax



**iii. Embed** - Embed R code that creates output to include in the report




**iv. Render** - Replace R code with its output and transform the report into a slideshow, pdf, html or ms Word file.



**2. Open File** Start by saving a text file with the extension .Rmd, or open an RStudio Rmd template

- In the menu bar, click **File > New File > R Markdown...**
- A window will open. Select the class of output you would like to make with your .Rmd file
- Select the specific type of output to make with the radio buttons (you can change this later)
- Click OK



**3. Markdown** Next, write your report in plain text. Use markdown syntax to describe how to format text in the final report.

**syntax**

Plain text  
End a line with two spaces to start a new paragraph.

*\*italics\** and \_\_italics\_\_  
**\*\*bold\*\*** and \_\_bold\_\_  
<sup>superscript^2^</sup>  
~~--strikethrough--~~  
[link](www.rstudio.com)

# Header 1  
## Header 2  
### Header 3  
#### Header 4  
##### Header 5  
##### Header 6

endash: --  
emdash: ---  
ellipsis: ...  
inline equation:  $\$A = \pi r^{\{2\}}$   
image: 

horizontal rule (or slide break):  
\*\*\*

> block quote

\* unordered list  
\* item 2  
+ sub-item 1  
+ sub-item 2

1. ordered list  
2. item 2  
+ sub-item 1  
+ sub-item 2


Table Header	
Table Cell	Cell 2
Cell 3	Cell 4

**becomes**

Plain text  
End a line with two spaces to start a new paragraph.

*italics* and italics  
**bold** and bold  
<sup>superscript</sup>  
~~strikethrough~~  
[link](#)

**Header 1**  
**Header 2**  
**Header 3**  
**Header 4**  
**Header 5**  
**Header 6**

endash: --  
emdash: ---  
ellipsis: ...  
inline equation:  $A = \pi r^2$   
image: 

horizontal rule (or slide break):  
-----

block quote

- unordered list
- item 2
  - + sub-item 1
  - + sub-item 2

1. ordered list  
2. item 2  
+ sub-item 1  
+ sub-item 2

Table Header	Second Header
Table Cell	Cell 2
Cell 3	Cell 4

**4. Choose Output** Write a YAML header that explains what type of document to build from your R Markdown file.

**YAML**


A YAML header is a set of key: value pairs at the start of your file. Begin and end the header with a line of three dashes (- - -)


```
title: "Untitled"
author: "Anonymous"
output: html_document
---
```


This is the start of my report. The above is metadata saved in a YAML header.


The RStudio template writes the YAML header for you


The output.value determines which type of file R will build from your .Rmd file (in Step 6)

**output: html\_document** ..... html file (web page) 

**output: pdf\_document** ..... pdf document 

**output: word\_document** ..... Microsoft Word .docx 

**output: beamer\_presentation** ..... beamer slideshow (pdf) 

**output: ioslides\_presentation** ..... ioslides slideshow (html) 

**5. Embed Code** Use knitr syntax to embed R code into your report. R will run the code and include the results when you render your report.

**inline code**

Surround code with back ticks and r.  
R replaces inline code with its results.


**code chunks**

Start a chunk with `{r}`.  
End a chunk with ```

**6. Render** Use your .Rmd file as a blueprint to build a finished report.

Render your report in one of two ways

1. Run `rmarkdown::render("<file path>")`



RStudio® is a trademark of RStudio, Inc. • All rights reserved • [info@rstudio.com](mailto:info@rstudio.com) • 844-448-1212 • [rstudio.com](http://rstudio.com)

Two plus two equals `"r 2 + 2"`. → Two plus two equals 4.

Here's some code `{r}`  
`dim(iris)` → `dim(iris)`

Here's some code `{r}`  
`dim(iris)` → `## [1] 150 5`

### display options

Use knitr options to style the output of a chunk. Place options in brackets above the chunk.

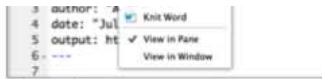
Here's some code `{r eval=FALSE}`  
`dim(iris)` → `dim(iris)`

Here's some code `{r echo=FALSE}`  
`dim(iris)` → `## [1] 150 5`

option	default	effect
<code>eval</code>	TRUE	Whether to evaluate the code and include its results
<code>echo</code>	TRUE	Whether to display code along with its results
<code>warning</code>	TRUE	Whether to display warnings
<code>error</code>	FALSE	Whether to display errors
<code>message</code>	TRUE	Whether to display messages
<code>tidy</code>	FALSE	Whether to reformat code in a tidy way when displaying it
<code>results</code>	"markup"	"markup", "asis", "hold", or "hide"
<code>cache</code>	FALSE	Whether to cache results for future renders
<code>comment</code>	"##"	Comment character to preface results with
<code>fig.width</code>	7	Width in inches for plots created in chunk
<code>fig.height</code>	7	Height in inches for plots created in chunk

For more details visit [yihui.name/knitr/](http://yihui.name/knitr/)

2. Click the **knit HTML** button at the top of the RStudio scripts pane



When you render, R will

- execute each embedded code chunk and insert the results into your report
- build a new version of your report in the output file type
- open a preview of the output file in the viewer pane
- save the output file in your working directory

### 7. Interactive Docs

Turn your report into an interactive Shiny document in 3 steps

1. Add runtime: shiny to the YAML header


```
title: "Line graph"
output: html_document
runtime: shiny
```

2. In the code chunks, add Shiny input functions to embed widgets. Add Shiny render functions to embed reactive output

```
Choose a time series:
r echo = FALSE
selectInput("data", "", c("a", "b"))

See a plot:
r echo = FALSE
renderPlot({
  d <- get(input$data)
  plot(d)
})
```

3. Render with `markdown::run` or click Run Document in RStudio



*\* Note: your report will be a Shiny app, which means you must choose an html output format, like `html_document` (for an interactive report) or `ioslides_presentation` (for an interactive slideshow).*

### 8. Publish

Share your report where users can visit it online

**Rpubs.com**

Share non-interactive documents on RStudio's free R Markdown publishing site


[www.rpubs.com](http://www.rpubs.com)

**ShinyApps.io**

Host an interactive document on RStudio's server. Free and paid options

[www.shinyapps.io](http://www.shinyapps.io)

Click the "Publish" button in the RStudio preview window to publish to [rpubs.com](http://rpubs.com) with one click.




### 9. Learn More

Documentation and examples - [rmarkdown.rstudio.com](http://rmarkdown.rstudio.com)

Further Articles - [shiny.rstudio.com/articles](http://shiny.rstudio.com/articles)

🌐 - [blog.rstudio.com](http://blog.rstudio.com)

🐦 - @rstudio

 RStudio® and Shiny® are trademarks of RStudio, Inc. All rights reserved. [info@rstudio.com](mailto:info@rstudio.com) 844-458-1111 [rstudio.com](http://rstudio.com)

Check out all of our cheat sheets:

- The Shiny cheat sheet
- The R Markdown cheat sheet

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

- [Firefox](#)
- [Chrome](#)
- [Internet Explorer 10+](#)
- [Safari](#)

Pete

-----

What did you use to make the sheet ? It's nice.

Garrett

-----

Thanks, Pete. I used keynote.

Arun Soni

-----

Could you add a section on how to customise fonts and other css items when publishing to a pdf document?

Karthik

-----

Thanks a ton mate! Incredibly useful!

oraclemaster

## 2.29 Using Action Buttons

# Using Action Buttons

ADDED: 26 MAR 2015

This article describes five patterns to use with Shiny's [action buttons](#) and [action links](#). Action buttons and action links are different from other Shiny widgets because they are intended to be used exclusively with `observeEvent()` or `eventReactive()`.

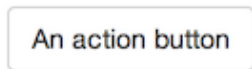
## How action buttons work

Create an action button with `actionButton()` and an action link with `actionLink()`. Each of these functions takes two arguments:

- `inputId` - the ID of the button or link
- `label` - the label to display in the button or link

```
actionButton("button", "An action button")
actionLink("button", "An action link")
```

An action button appears as a button in your app.



An action link appears as a hyperlink, but behaves in the same way as an action button.

[An action link](#)

Like all widgets, action buttons have a value. The value is an integer that changes each time a user clicks the button. You can access this value from within your app as `input$<inputId>` where `<inputId>` is the ID that you assigned to your action button.

Action buttons are different from other widgets because the value of an action button is almost never meaningful by itself. The value is designed to be observed by one of `observeEvent()` or `eventReactive()`. These functions monitor the value, and when it changes they run a block of code.

The patterns below explain this arrangement and illustrate the most popular ways to use an action button or an action link.

---

## Pattern 1 - Command

Use `observeEvent()` to trigger a command with an action button.

### Example



In the code above, `session$setCustomMessage()` generates a popup message.

`tags$head(tags$script(src = "message-handler.js"))` supplies the JavaScript that makes this possible. See [this example](#) to learn more about `sendCustomMessage()`.

## Why the pattern works

Action buttons do not automatically generate actions in Shiny. Like other widgets, action buttons maintain a state (a value). The state changes when a user clicks the button.

`observeEvent()` observes a reactive value, which is set in the first argument of `observeEvent()`. Whenever the value changes, `observeEvent()` will run its second argument, which should be a block of code surrounded in braces.

This pattern uses `observeEvent()` to connect the change in an action button's value to the code that the action button should trigger.

## Tips

- `observeEvent()` isolates the block of code in its second argument with `isolate()`.
- `observeEvent()` only notices *changes* in the value of the action button. It does not matter what the actual value of the button is. If your code depends on the value of the action button, it may be mis-written.

---

## Pattern 2 - Delay reactions

Use `eventReactive()` to delay reactions until a user clicks the action button.

## Example

## Why the pattern works

`eventReactive()` creates a reactive expression that monitors a reactive value, which is set in the first argument of `eventReactive()`. The expression will be invalidated whenever the value changes, but it will ignore changes in other reactive values.

Complete this pattern by using the reactive expression created by `eventReactive()` in rendered output. Output that depends on the expression will not update until the expression is invalidated, i.e. until the action button is clicked.

## Tips

- Like `observeEvent()`, `eventReactive()` isolates the block of code in its second argument with `isolate()`.
- `eventReactive()` returns `NULL` until the action button is clicked. As a result, the graph does not appear until the user asks for it by clicking “Go”.

---

## Pattern 3 - Dueling buttons

To build several action buttons that control the same object, combine `observeEvent()` calls with `reactiveValues()`.

## Example

## Why the pattern works

`reactiveValues()` creates a reactive values object, a list of reactive values that you can update and call programmatically. These values are like the values stored in Shiny's `input` object with one difference: you can update the values of a reactive values object, but you cannot normally update the values of the `input` object (those values are reserved for the user to update interactively).

To complete the pattern, monitor each button with its own `observeEvent()` call. Arrange for the calls to update the object created by `reactiveValues()`. Reactive values obey reference class semantics, which means that you can update them from within the scope of an `observeEvent()` function.

---

## Pattern 4 - Reset buttons

To create a reset button, use the above pattern to assign `NULL` to a reactive values object.

### Example

this, arrange for a button to assign `NULL` to the reactive values object with the help of `observeEvent()` .

---

## Pattern 5 - Reset on tab change

Observe the value of a `tabsetPanel()` , `navListPanel()` , or `navbarPage()` with `observeEvent()` to reset the value of an object each time your user switches tabs.

### Example

## Why this pattern works

This pattern extends the previous reset pattern. You use `observeEvent()` to reset an element of a reactive values object. However, instead of observing the value of an action button, you observe the value of a tab function.

`tabsetPanel()`, `navlistPanel()`, and `navbarPage()` each combine multiple tabs (created with `tabPanel()`) into a single ui object. These functions maintain a reactive value that contains the title of the current tab. When your user navigates to a new tab, this value changes. `observeEvent()` resets the reactive value to `NULL` when it does.

As with the patterns above, this pattern requires you to store and manipulate a value created with `reactiveValues()`.

## Tips

- Although the example uses `tabsetPanel()`, you can achieve the same effect with `navlistPanel()` and `navbarPage()`.

---

## Recap

Action buttons and action links are meant to be used with one of `observeEvent()` or `eventReactive()`. You can extend the effects of an action button with `reactiveValues()`.

- Use `observeEvent()` to trigger a block of code with an action button.
- Use `eventReactive()` to update rendered output with an action button.
- Use `reactiveValues()` to maintain an object for multiple action buttons to interact with.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

akshay madiwale

---

Hi, I have one screen where there are two `tabPanel`. I am selecting some columns from these two `tabPanel`. In the end there is `submitButton`, if that is pressed then it will display the columns choosed. It should make the screen blank and give the columns choosed as output. Can we delete the `tabPanel`?

Thomas Rampley

---

I copied the first action button example verbatim into RStudio and nothing happens when I click the action button. This is consistent with other attempts to insert one in my own apps. Anyone else have this issue? Running 3.2 on the latest version of RStudio, Windows 7.

Garrett

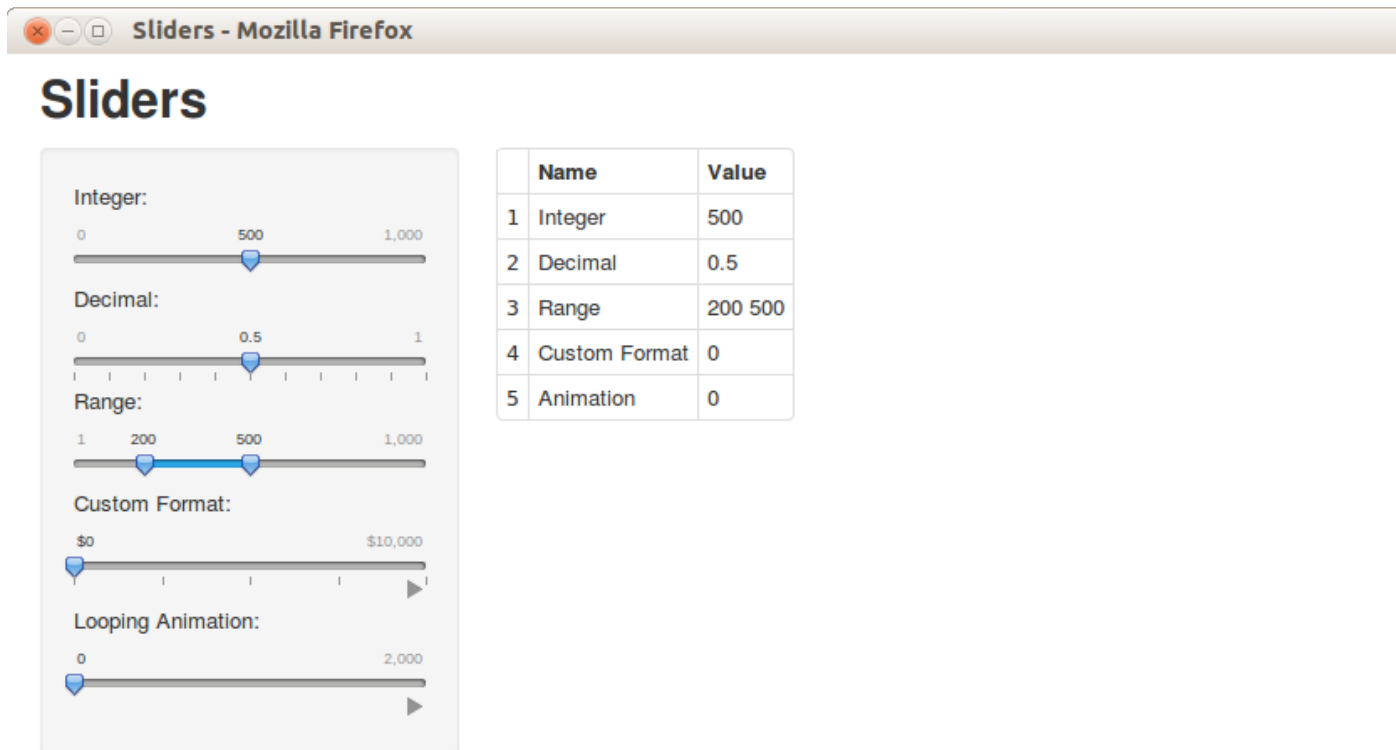
---

Thomas, You would need to do more than copy and paste the code to get Example 1 to work. You'd also need to write the script `message-handler.js` and save it in a folder named `www` alongside the app. The example is a live app in this page---try clicking the button. I only meant for you to use the app in this page and see the [comments powered by Disqus](#)

## 2.30 Using sliders

# Using sliders

ADDED: 06 JAN 2014



	Name	Value
1	Integer	500
2	Decimal	0.5
3	Range	200 500
4	Custom Format	0
5	Animation	0

The Sliders application demonstrates the many capabilities of slider controls, including the ability to run an animation sequence. To run the example type:

```
> library(shiny)
> runExample("05_sliders")
```

## Customizing Sliders

Shiny slider controls are extremely capable and customizable. Features supported include:

- The ability to input both single values and ranges
- Custom formats for value display (e.g for currency)
- The ability to animate the slider across a range of values

Slider controls are created by calling the `sliderInput` function. The `ui.R` file demonstrates using sliders with a variety of options:

`ui.R`

```

library(shiny)

# Define UI for slider demo application
shinyUI(pageWithSidebar(

  # Application title
  headerPanel("Sliders"),

  # Sidebar with sliders that demonstrate various available options
  sidebarPanel(
    # Simple integer interval
    sliderInput("integer", "Integer:",
               min=0, max=1000, value=500),

    # Decimal interval with step value
    sliderInput("decimal", "Decimal:",
               min = 0, max = 1, value = 0.5, step= 0.1),

    # Specification of range within an interval
    sliderInput("range", "Range:",
               min = 1, max = 1000, value = c(200, 500)),

    # Provide a custom currency format for value display, with basic animation
    sliderInput("format", "Custom Format:",
               min = 0, max = 10000, value = 0, step = 2500,
               format="$#,##0", locale="us", animate=TRUE),

    # Animation with custom interval (in ms) to control speed, plus looping
    sliderInput("animation", "Looping Animation:", 1, 2000, 1, step = 10,
               animate=animationOptions(interval=300, loop=T)
  ),

  # Show a table summarizing the values entered
  mainPanel(
    tableOutput("values")
  )
))

```

## Server Script

The server side of the Slider application is very straightforward: it creates a data frame containing all of the input values and then renders it as an HTML table:

server.R

```

library(shiny)

# Define server logic for slider examples
shinyServer(function(input, output) {

  # Reactive expression to compose a data frame containing all of the values
  sliderValues <- reactive({

    # Compose data frame
    data.frame(
      Name = c("Integer",

```



```

    "Decimal",
    "Range",
    "Custom Format",
    "Animation"),
  Value = as.character(c(input$integer,
                        input$decimal,
                        paste(input$range, collapse=' '),
                        input$format,
                        input$animation)),
  stringsAsFactors=FALSE)
})

# Show the values using an HTML table
output$values <- renderTable({
  sliderValues()
})
})

```

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

JosÃ© BayoÃ­n Santiago CalderÃ³n

---

Any way to pass it the min\_interval argument from ion.rangeSlider?

mwalkup55

---

Is it possible to rotate the slider, so that it is arranged vertically?

danbro

---

I am using slider inputs in my shiny application as weights for a mathematical model. I want to create an action button to set the sliders to their initial start positions. Any ideas on how to do this?

Emil Kirkegaard

---

Use updateSliderInput(). Live example: <http://shiny.rstudio.com/galle...>

rhleu

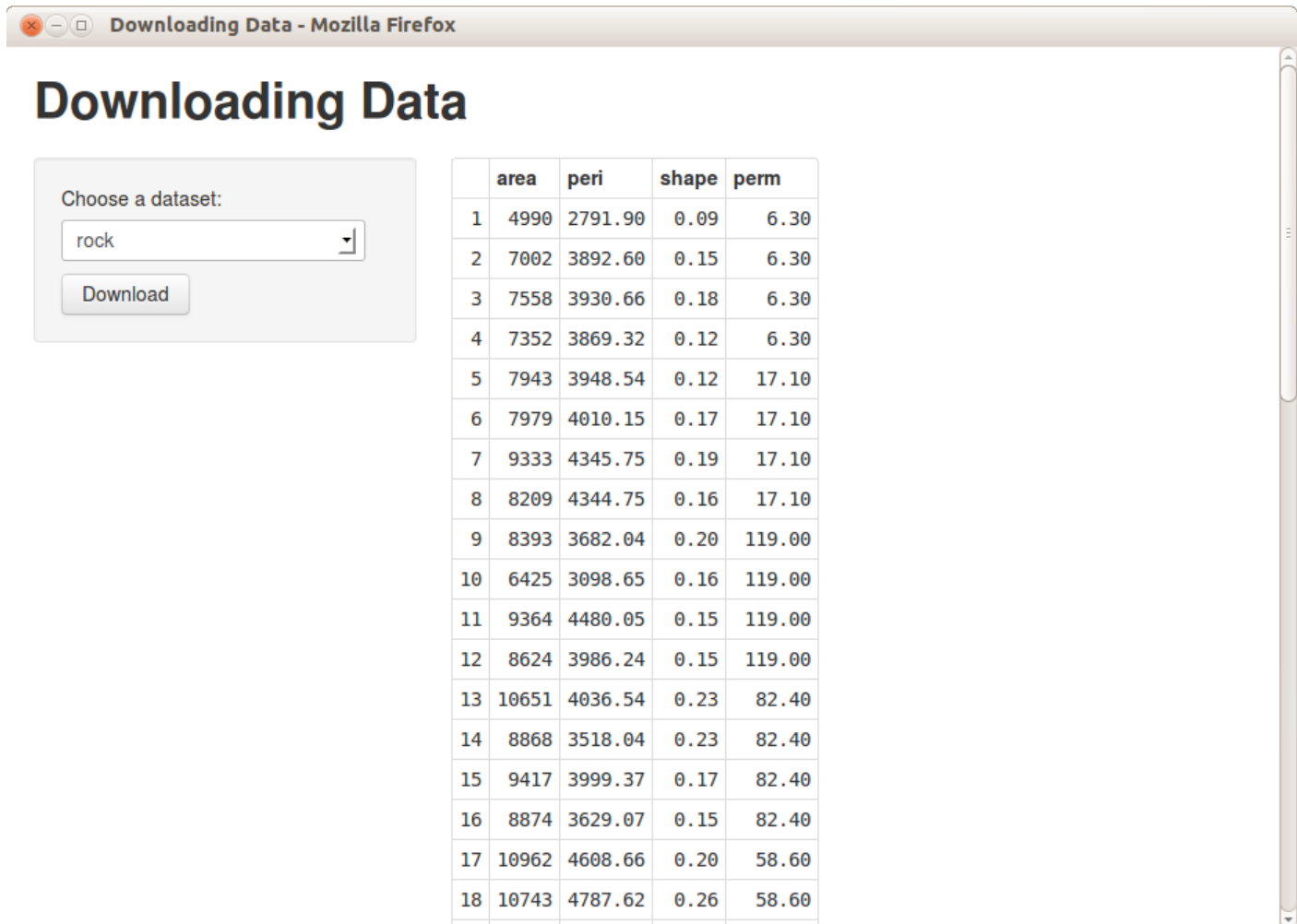
---

comments powered by [Disqus](#)

## 2.31 Help users download data from your

# Help users download data from your app

ADDED: 06 JAN 2014



	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
11	9364	4480.05	0.15	119.00
12	8624	3986.24	0.15	119.00
13	10651	4036.54	0.23	82.40
14	8868	3518.04	0.23	82.40
15	9417	3999.37	0.17	82.40
16	8874	3629.07	0.15	82.40
17	10962	4608.66	0.20	58.60
18	10743	4787.62	0.26	58.60

Shiny has the ability to offer file downloads that are created on the fly, which makes it easy to build data exporting features. See [here](#) for an example app with file downloads.

To run the example below, type:

```
> library(shiny)
> runExample("10_download")
```

You define a download using the `downloadHandler` function on the server side, and either `downloadButton` or `downloadLink` in the UI:

## ui.R

```
shinyUI(pageWithSidebar(
  headerPanel('Download Example'),
  sidebarPanel(
    selectInput("dataset", "Choose a dataset:",
               choices = c("rock", "pressure", "cars")),
    downloadButton('downloadData', 'Download')
  ),
  mainPanel(
    tableOutput('table')
  )
))
```

## server.R

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

  output$table <- renderTable({
    datasetInput()
  })

  output$downloadData <- downloadHandler(
    filename = function() { paste(input$dataset, '.csv', sep='') },
    content = function(file) {
      write.csv(datasetInput(), file)
    }
  )
})
```

As you can see, `downloadHandler` takes a `filename` argument, which tells the web browser what filename to default to when saving. This argument can either be a simple string, or it can be a function that returns a string (as is the case here).

The `content` argument must be a function that takes a single argument, the file name of a non-existent temp file. The `content` function is responsible for writing the contents of the file download into that temp file.

Both the `filename` and `content` arguments can use reactive values and expressions (although in the case of `filename`, if you are using a reactive value, be sure your argument is an actual function;

`filename = paste(input$dataset, '.csv')` will not work the way you want it to, since it is evaluated only once, when the download handler is being defined).

Generally, those are the only two arguments you'll need. There is an optional `contentType` argument; if it is `NA` or `NULL`, Shiny will attempt to guess the appropriate value based on the filename. Provide your own content type string (e.g. `"text/plain"`) if you want to override this behavior.

you'd like to get specific help, we recommend the [Shiny Discussion Forum](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

mdimi

---

I would like to implement a download link for files that already exists on the file system. In other words, I want to skip writing a new file to the disk. How do I go about that?

Artem Klevtsov

---

The download button does not work in the RStudio viewer. Details:

<http://stackoverflow.com/q/259...>

dI7631

---

What if I want to give the user a chance to download more than one file? Do I have to create a separate button for each file or is it possible to create a button with a pull-down so that the user can download any of several files?

Gordon Arsenoff

---

comments powered by [Disqus](#)

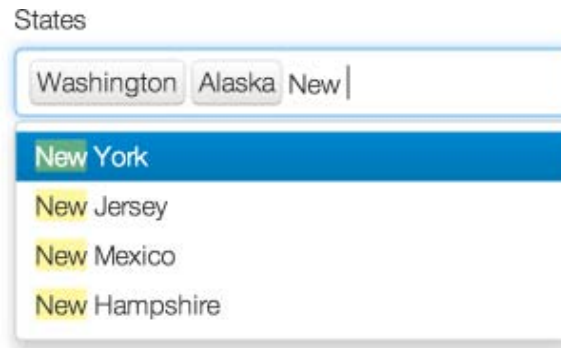
## 2.32 Using selectize input

# Using selectize input

ADDED: 04 JUN 2014

BY: YIHUI XIE

The JavaScript library [selectize.js](#) provides a much more flexible interface compared to the basic select input. It allows you to type and search in the options, use placeholders, control the number of options/items to show/select, and so on. See [here](#) for an example app.



To create a selectize input, you can use the function `selectizeInput()`, and the usage is very similar to `selectInput()`:

```
selectizeInput(inputId, label, choices, selected = NULL, multiple = FALSE,  
              options = NULL)
```

A major difference between the usage of `selectizeInput()` and `selectInput()` is the `options` argument, which is a list of parameters to initialize the selectize input. Please check out the [usage documentation](#) of `selectize.js` for all the possible parameters. [This example](#) shows a side by side comparison between selectize and select input.

When we type in the input box, selectize will start searching for the options that partially match the string we typed. The searching can be done on the client side (default behavior), when all the possible options have been written on the HTML page. It can also be done on the server side, using R to match the string and return results. This is particularly useful when the number of choices is very large. For example, when there are 100,000 choices for the selectize input, it will be slow to write all of them at once into the page, but we can start from an empty selectize input, and only fetch the choices that we may need, which can be much faster. We will introduce both types of the selectize input below.

## Client-side selectize

The selectize input returns the item(s) that you selected, but keep in mind that it may also return an empty string when all the selected items are deleted using the key `Backspace` or `Del` etc.

We can make use of the `options` argument to specify a list of initialization options. Here are some quick examples:

```
# allow creation of new items in the drop-down list
```

```

selectizeInput (
  'foo', label = NULL, choices = state.name,
  options = list(create = TRUE)
)

# show at most 5 options in the list
selectizeInput(..., options = list(maxOptions = 5))

# allow at most 2 items to be selected
selectizeInput(..., options = list(maxItems = 2))

# add a placeholder in the text box
selectizeInput(..., options = list(placeholder = 'select a state name'))

```

Of course, you can combine multiple options, e.g.

```
selectizeInput(..., options = list(maxItems = 3, placeholder = 'hi there'))
```

## Server-side selectize

The client-side selectize input relies solely on JavaScript to process searching on typing. The server-side selectize input uses R to process searching, and R will return the filtered data to selectize. To use the server version, you need to create a selectize instance in the UI, and update it to the server version:

```

# in ui.R
selectizeInput('foo', choices = NULL, ...)

# in server.R
shinyServer(function(input, output, session) {
  updateSelectizeInput(session, 'foo', choices = data, server = TRUE)
})

```

You may use `choices = NULL` to create an empty selectize instance, so that it will load quickly initially, then use `updateSelectizeInput(server = TRUE)` to pass the `choices` data to R. Here `data` can be an arbitrary R data object, such as a (named) character vector, or a data frame. Note the client-side selectize can only accept a character vector for the `choices` argument.

What happens when we type in the text box is:

1. the character string in the text box is sent to R, and split into multiple keywords using white spaces;
2. R matches each keyword in the variable(s) specified in the `searchField` option of selectize initialization options;
3. depending on the `searchConjunction` option ('and' or 'or'), the results from each keyword are combined using AND or OR;
4. the first `maxOptions` records of the `data` is returned (as JSON);

When we use the server version of selectize, we may want to define the `render` method for selectize, although normally the default rendering method should just work. A custom rendering method allows us to create richer content in the drop-down list, instead of just some plain text options. [This example](#) shows how we can render images in the options.

```

updateSelectizeInput(..., options = list(render = I(
  '{
    options: function(item, escape) {
      // your own code to generate HTML here for each option item
    }
  }

```

```

    }
  }'
)))

```

The `options` element of the `render` object is a JavaScript function that has two arguments, `item` and `escape`. Please read the `selectize.js` documentation to understand what they mean. Basically you can treat `item` as a record in the `data` that we passed in as `choices`. For example, if `choices = state.name`, an `item` might be

```

{
  label: "California",
  value: "California"
}

```

You can define the rendering method for `options` as

```

function(item, escape) {
  return "<div>" + escape(item.value) + "</div>";
}

```

This means we create a `div` for each of the items, and the `div` contains their values. This is a very simple example, and we can use more complicated data objects, and write rendering methods accordingly. Here is a quick example:

```

updateSelectizeInput(
  ...,
  choices = cbind(name = rownames(mtcars), mtcars),
  options = list(render = I(
    '{
      options: function(item, escape) {
        return "<div><strong>" + escape(item.name) + "</strong> (" +
          "MPG: " + item.mpg +
          ", Transmission: " + item.am == 1 ? "automatic" : "manual" + ")")
      }
    }'))
)

```

Then in the drop-down list, we will see the name of the car in bold text, and the variables `mpg` and `am` in the parentheses (e.g. **Mazda RX4** (MPG: 21.0, Transmission: manual)).

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Mahender

---

Currently i am using selectize.js in one of our project. I am facing one issue with the duplication of tags. For Ex: if i type test in the input field it is taking the value as test and if i type the Test in the input field, this is also taking. Here duplication in my tags list like test and Test.

Please help me how can i avoid duplication with in the tags if i type uppercase or lower case.

Regis A. James

---

Another tip (something I learned the hairpullingly-hard way):

When using the server-side management of selectable options, it should be noted that in order for already selected entries to remain selected options are updated, you **MUST** set the value of `updateSelectizeInput`'s "selected" argument to `input$[nameOfSelectizeInputBeingUpdated]`, or else your selections will be automatically deselected, driving you CRAZY! Also, to prevent the inevitable infinite loop that occurs when the browser's Javascript sends the Shiny server an update faster than the server can send an update to the browser (resulting in a constant

[comments powered by Disqus](#)



## 2.33 Render images in a Shiny app

# Render images in a Shiny app

ADDED: 06 JAN 2014

## Sending Images

When you want to have R generate a plot and send it to the client browser, the `renderPlot()` function will in most cases do the job. But when you need finer control over the process, you might need to use the `renderImage()` function instead. This is demonstrated in the [image output](#) demo application.

## About `renderPlot()`

`renderPlot()` is useful for any time where R generates an image using its normal graphical device system. In other words, any plot-generating code that would normally go between `png()` and `dev.off()` can be used in `renderPlot()`. If the following code works from the console, then it should work in `renderPlot()`:

```
png()
# Your plotting code here
dev.off()

# This would go in shinyServer()
output$myPlot <- renderPlot({
  # Your plotting code here
})
```

`renderPlot()` takes care of a number of details automatically: it will resize the image to fit the output window, and it will even increase the resolution of the output image when displaying on high-resolution (“Retina”) screens.

The limitation to `renderPlot()` is that it won’t send just any image file to the browser – the image must be generated by code that uses R’s graphical output device system. Other methods of creating images can’t be sent by `renderPlot()`. For example, the following won’t work:

- Image files generated by the `writePNG()` function from the `png` package.
- Image files generated by the `rgl.snapshot()` function, which creates images from 3D plots made with the `rgl` package.
- Images generated by an external program.
- Pre-rendered images.

The solution in these cases is the `renderImage()` function.

## Using `renderImage()`

Image files can be sent using `renderImage()`. The expression that you pass to `renderImage()` must return a list containing an element named `src`, which is the path to the file. Here is a very basic example of a Shiny app with an output that generates a plot and sends it with `renderImage()`:

## server.R

```
shinyServer(function(input, output, session) {
  output$myImage <- renderImage({
    # A temp file to save the output.
    # This file will be removed later by renderImage
    outfile <- tempfile(fileext='.png')

    # Generate the PNG
    png(outfile, width=400, height=300)
    hist(rnorm(input$obs), main="Generated in renderImage()")
    dev.off()

    # Return a list containing the filename
    list(src = outfile,
         contentType = 'image/png',
         width = 400,
         height = 300,
         alt = "This is alternate text")
  }, deleteFile = TRUE)
})
```

## ui.r

```
shinyUI(pageWithSidebar(
  headerPanel("renderImage example"),
  sidebarPanel(
    sliderInput("obs", "Number of observations:",
               min = 0, max = 1000, value = 500)
  ),
  mainPanel(
    # Use imageOutput to place the image on the page
    imageOutput("myImage")
  )
))
```

Each time this output object is re-executed, it creates a new PNG file, saves a plot to it, then returns a list containing the filename along with some other values.

Because the `deleteFile` argument is `TRUE`, Shiny will delete the file (specified by the `src` element) after it sends the data. This is appropriate for a case like this, where the image is created on-the-fly, but it wouldn't be appropriate when, for example, your app sends pre-rendered images.

In this particular case, the image file is created with the `png()` function. But it just as well could have been created with `writePNG()` from the `png` package, or by any other method. If you have the filename of the image, you can send it with `renderImage()`.

## Structure of the returned list

The list returned in the example above contains the following:

- `src`: The output file path.
- `contentType`: The MIME type of the file. If this is missing, Shiny will try to autodetect the MIME type, from the file extension.
- `width` and `height`: The desired output size, in pixels.
- `alt`: Alternate text for the image.

Except for `src` and `contentType`, all values are passed through directly to the `<img>` DOM element on the web page. The effect is similar to having an image tag with the following:

```

```

Note that the `src="..."` is shorthand for a longer URL. For browsers that support the [data URI scheme](#), the `src` and `contentType` from the returned list are put together to create a special URL that embeds the data, so the result would be similar to something like this:

```

```

For browsers that don't support the data URI scheme, Shiny sends a URL that points to the file.

## Sending pre-rendered images with `renderImage()`

If your Shiny app has pre-rendered images saved in a subdirectory, you can send them using `renderImage()`. Suppose the images are in the subdirectory `images/`, and are named `image1.jpeg`, `image2.jpeg`, and so on. The following code would send the appropriate image, depending on the value of `input$n`:

server.R

```
shinyServer(function(input, output, session) {
  # Send a pre-rendered image, and don't delete the image after sending it
  output$preImage <- renderImage({
    # When input$n is 3, filename is ./images/image3.jpeg
    filename <- normalizePath(file.path('./images',
                                         paste('image', input$n, '.jpeg', sep='')))

    # Return a list containing the filename and alt text
    list(src = filename,
         alt = paste("Image number", input$n))
  }, deleteFile = FALSE)
})
```

In this example, `deleteFile` is `FALSE` because the images aren't ephemeral; we don't want Shiny to delete an image after sending it.

Note that this might be less efficient than putting images in `www/images` and emitting HTML that points to the images, because in the latter case the image will be cached by the browser.

## Using `clientData` values

In the first example above, the plot size was fixed at 400 by 300 pixels. For dynamic resizing, it's possible to use values from `session$clientData` to detect the output size.

In the example below, the output object is `output$myImage`, and the width and height on the client browser are sent via `session$clientData$output_myImage_width` and `session$clientData$output_myImage_height`. This example also uses `session$clientData$pixelRatio` to multiply the resolution of the image, so that it appears sharp on high-resolution (Retina) displays:

server.R

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

```

# A dynamically-sized plot
output$myImage <- renderImage({
  # Read myImage's width and height. These are reactive values, so this
  # expression will re-run whenever they change.
  width <- session$clientData$output_myImage_width
  height <- session$clientData$output_myImage_height

  # For high-res displays, this will be greater than 1
  pixel_ratio <- session$clientData$pixel_ratio

  # A temp file to save the output.
  outfile <- tempfile(fileext='.png')

  # Generate the image file
  png(outfile, width=width*pixel_ratio, height=height*pixel_ratio,
       res=72*pixel_ratio)
  hist(rnorm(input$obs))
  dev.off()

  # Return a list containing the filename
  list(src = outfile,
       width = width,
       height = height,
       alt = "This is alternate text")
}, deleteFile = TRUE)

# This code reimplements many of the features of `renderPlot()`.
# The effect of this code is very similar to:
# renderPlot({
#   hist(rnorm(input$obs))
# })
})

```

The `width` and `height` values passed to `png()` specify the pixel dimensions of the saved image. These can differ from the `width` and `height` values in the returned list: those values are the pixel dimensions to used display the image. For high-res displays (where `pixel_ratio` is 2), a “virtual” pixel in the browser might correspond to 2 x 2 physical pixels, and a double-resolution image will make use of each of the physical pixels.

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

I am trying to show selective image as a result of outcome from a predictive model. The images are predefined and stored in www folder. While using `renderImage` I had to specifically mention the path - it is not taking the image from the default www folder. This solution is working fine with the local machine - but when I deploy to [shinyapps.io](http://shinyapps.io), it not able to find the image (showing alternate text)... what is the way out ?

Juan Pedro Luengas Garc a

---

How can I tell shiny where do I want an image? I'm trying to place a logo in the right side of the title panel but I haven't found the instruction. Any help? Thanks!!!

Abhijit Sahay

---

Thank you for creating this wonderful tool.

I am using `renderPlot` to show different plots in response to changes in a `selectInput`, but for one of the choices, I would like to show a pre-computed image (using [comments powered by Disqus](#))

## 2.34 How to use DataTables in a Shiny

# How to use DataTables in a Shiny App

ADDED: 06 JAN 2014

BY: YIHUI XIE

diamonds

mtcars

iris

10
▼ records per page

Search:

carat	cut	color	clarity	depth	table	price	x	y	z
0.24	Very Good	J	VVS2	62.8	57	336	3.94	3.96	2.48
0.27	Very Good	J	VVS2	60.8	57	443	4.16	4.20	2.54
0.30	Very Good	J	SI1	62.7	59	351	4.21	4.27	2.66
0.30	Very Good	J	VS2	62.2	57	357	4.28	4.30	2.67
0.30	Very Good	J	VS1	59.5	60	427	4.39	4.45	2.63
0.30	Very Good	J	VS1	61.1	61	427	4.33	4.38	2.66
0.30	Very Good	J	VS2	61.6	61	368	4.28	4.33	2.65
0.30	Very Good	J	SI1	63.4	54	450	4.29	4.23	2.70
0.30	Very Good	J	VS2	63.4	57	473	4.26	4.22	2.69
0.30	Very Good	J	VS1	63.5	58	506	4.27	4.24	2.70

carat

cut

J

clarity

depth

table

price

x

y

z

Showing 1 to 10 of 985 entries (filtered from 53,940 total entries)

← Previous

1

2

3

4

5

Next →

## Basic Usage

The `DataTables` application demonstrates HTML tables using the jQuery library `DataTables`.

The basic usage is to create an output element in the UI using `dataTableOutput(id = 'foo')`, and render a table on the server side using `output$foo <- renderDataTable({ data })`. Normally `renderDataTable()` takes an expression that returns a rectangular data object with column names, such as a data frame or a matrix. Below is a minimal example:

```
library(shiny)

runApp(list(
  ui = basicPage(
    h2('The mtcars data'),
    dataTableOutput('mytable')
  ),
  server = function(input, output) {
    output$mytable = renderDataTable({
      mtcars
    })
  }
))
```

By default, the data is paginated, showing 25 rows per page. The number of rows to display can be changed through the drop down menu in the top-left. We can sort the columns by clicking on the column headers, and sort multiple columns by holding the `Shift` key while clicking (the sorting direction loops through `ascending`, `descending`, and `none` if we keep on clicking). We can search globally in the table using the text input box in the top-right, or search individual columns using the text boxes at the bottom. Currently the searching terms are treated as regular expressions in R. Since searching can be time-consuming in large datasets, there is a delay of 0.5 seconds (customizable) before searching is really processed; that means if we type fast enough in the search box, searching may be processed only once on the server side even if we have typed more than one character.

## Customizing DataTables

There are a large number of options in DataTables that are customizable (see its website for details). In [this example](#), we show a few possibilities. First, we create the UI to display three datasets `diamonds`, `mtcars`, and `iris`, with each dataset in its own tab:

ui.R

```
library(shiny)
library(ggplot2) # for the diamonds dataset

shinyUI(pageWithSidebar(
  headerPanel('Examples of DataTables'),
  sidebarPanel(
    checkboxGroupInput('show_vars', 'Columns in diamonds to show:', names(diamonds),
      selected = names(diamonds)),
    helpText('For the diamonds data, we can select variables to show in the table;
      for the mtcars example, we use orderClasses = TRUE so that sorted
      columns are colored since they have special CSS classes attached;
      for the iris data, we customize the length menu so we can display 5
      rows per page.')
  ),
  mainPanel(
    tabsetPanel(
      tabPanel('diamonds',
        dataTableOutput("mytable1")),
      tabPanel('mtcars',
        dataTableOutput("mytable2")),
      tabPanel('iris',
        dataTableOutput("mytable3"))
    )
  )
))
```

We also added a checkbox group to select the columns to show in the `diamonds` data.

## Server Script

The `options` argument in `renderDataTable()` can take a list (literally an R list) of options, and pass them to DataTables when the table is initialized. For example, for the `mtcars` data, we pass `orderClasses = TRUE` to DataTables so that the sorted columns will have CSS classes attached on them (this is disabled by default); in this example, we can see the sorted columns are highlighted by darker colors. For the `iris` data, we pass the options `lengthMenu` and `pageLength` to customize the drop down menu, which has items `[10, 25, 50, 100]` by default; now the menu has three items `[5, 30, 50]`, and `5` is selected as the default value.

### server.R

```
library(shiny)

shinyServer(function(input, output) {

  # a large table, relative to input$show_vars
  output$mytable1 = renderDataTable({
    library(ggplot2)
    diamonds[, input$show_vars, drop = FALSE]
  })

  # sorted columns are colored now because CSS are attached to them
  output$mytable2 = renderDataTable({
    mtcars
  }, options = list(orderClasses = TRUE))

  # customize the length drop-down menu; display 5 rows per page by default
  output$mytable3 = renderDataTable({
    iris
  }, options = list(lengthMenu = c(5, 30, 50), pageLength = 5))

})
```

For more DataTable options, please refer to its full reference on its website.

## Upgrading from DataTables v1.9 to v1.10

Shiny ( $\geq$  v1.10.2) currently uses DataTables v1.10. If you have used DataTables in Shiny before (specifically, before Shiny v0.10.2), you may need to change some parameter names for your DataTables, because Shiny ( $\leq$  v0.10.1) was using DataTables v1.9, and DataTables v1.10 has changed the parameter names.

A guide for upgrading parameter names from DataTables 1.9 to 1.10 is here: <https://datatables.net/upgrade/1.10-convert>. Shiny will try to automatically correct some of the old parameter names, but this automatic correction certainly will not work for all use cases, especially if you have deeply customized your DataTables using complicated JavaScript options. You can see [this GIT commit](#) for examples of converting DataTables 1.9 names to 1.10 names.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.



Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

pami

---

Hello,

First thanks for the wonderful work being done here.

My question is : When an object is created through a `renderDataTable`, say "MyTable" , can we use it later elsewhere or do we have to re-create the table through old-fashioned ways?

To be more clear, is there an "input\$MyTable" or a `MyTable()` sort of thing to be used later on in the programme?

Thanks in advance

Ravi Chetan

---

Hi, just curious to know if you found any solution to your query above?

Xiushi Le

---

Would it be possible to allow mouse event on Output objects such as the table  
[comments powered by Disqus](#)

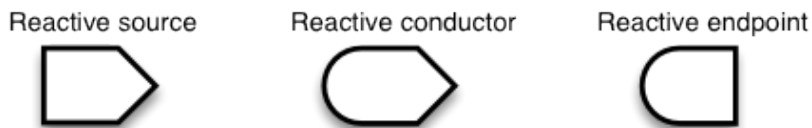
## 2.35 Reactivity: An overview

# Reactivity: An overview

ADDED: 06 JAN 2014

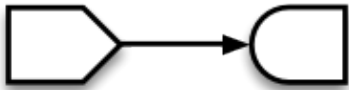
It's easy to build interactive applications with Shiny, but to get the most out of it, you'll need to understand the reactive programming model used by Shiny.

In Shiny, there are three kinds of objects in reactive programming: reactive sources, reactive conductors, and reactive endpoints, which are represented with these symbols:



## Reactive sources and endpoints

The simplest structure of a reactive program involves just a source and an endpoint:

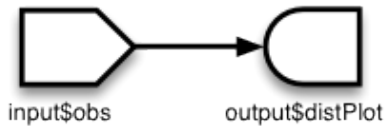


In a Shiny application, the source typically is user input through a browser interface. For example, when the user selects an item, types input, or clicks on a button, these actions will set values that are reactive sources. A reactive endpoint is usually something that appears in the user's browser window, such as a plot or a table of values.

In a simple Shiny application, reactive sources are accessible through the `input` object, and reactive endpoints are accessible through the `output` object. (Actually, there are other possible kinds of sources and endpoints, which we'll talk about later, but for now we'll just talk about `input` and `output`.)

This simple structure, with one source and one endpoint, is used by the `01_hello` example. The `server.R` code for that example looks something like this:

```
shinyServer(function(input, output) {  
  output$distPlot <- renderPlot({  
    hist(rnorm(input$obs))  
  })  
})
```



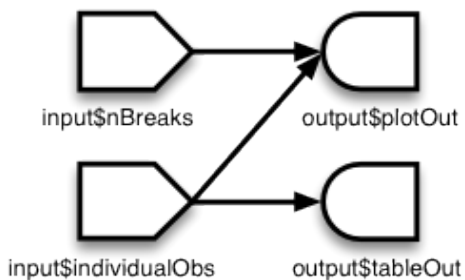
You can see it in action at [http://glimmer.rstudio.com/shiny/01\\_hello/](http://glimmer.rstudio.com/shiny/01_hello/).

The `output$distPlot` object is a reactive endpoint, and it uses the reactive source `input$obs`. Whenever `input$obs` changes, `output$distPlot` is notified that it needs to re-execute. In traditional program with an interactive user interface, this might involve setting up event handlers and writing code to read values and transfer data. Shiny does all these things for you behind the scenes, so that you can simply write code that looks like regular R code.

A reactive source can be connected to multiple endpoints, and vice versa. Here is a slightly more complex Shiny application:

```
shinyServer(function(input, output) {
  output$plotOut <- renderPlot({
    hist(faithful$eruptions, breaks = as.numeric(input$nBreaks))
    if (input$individualObs)
      rug(faithful$eruptions)
  })

  output$tableOut <- renderTable({
    if (input$individualObs)
      faithful
    else
      NULL
  })
})
```



In a Shiny application, there's no need to explicitly describe each of these relationships and tell R what to do when each input component changes; Shiny automatically handles these details for you.

In an app with the structure above, whenever the value of the `input$nBreaks` changes, the expression that generates the plot will automatically re-execute. Whenever the value of the `input$individualObs` changes, the plot and table functions will automatically re-execute. (In a Shiny application, most endpoint functions have their results automatically wrapped up and sent to the web browser.)

## Reactive conductors

So far we've seen reactive sources and reactive endpoints, and most simple examples use just these two components, wiring up sources directly to endpoints. It's also possible to put reactive components in between the sources and endpoints. These components are called *reactive conductors*.

A conductor can both be a dependent and have dependents. In other words, it can be both a parent and child in a

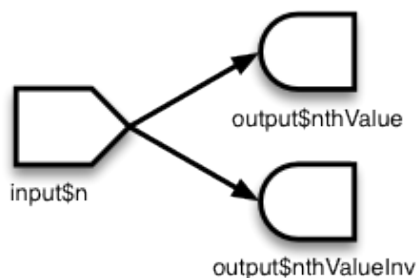
graph of the reactive structure. Sources can only be parents (they can have dependents), and endpoints can only be children (they can be dependents) in the reactive graph.

Reactive conductors can be useful for encapsulating slow or computationally expensive operations. For example, imagine that you have this application that takes a value `input$n` and prints the `_n`\_th value in the Fibonacci sequence, as well as the inverse of `_n`\_th value in the sequence plus one (note the code in these examples is condensed to illustrate reactive concepts, and doesn't necessarily represent coding best practices):

```
# Calculate nth number in Fibonacci sequence
fib <- function(n) ifelse(n<3, 1, fib(n-1)+fib(n-2))

shinyServer(function(input, output) {
  output$nthValue <- renderText({ fib(as.numeric(input$n)) })
  output$nthValueInv <- renderText({ 1 / fib(as.numeric(input$n)) })
})
```

The graph structure of this app is:



The `fib()` algorithm is very inefficient, so we don't want to run it more times than is absolutely necessary. But in this app, we're running it twice! On a reasonably fast modern machine, setting `input$n` to 30 takes about 15 seconds to calculate the answer, largely because `fib()` is run twice.

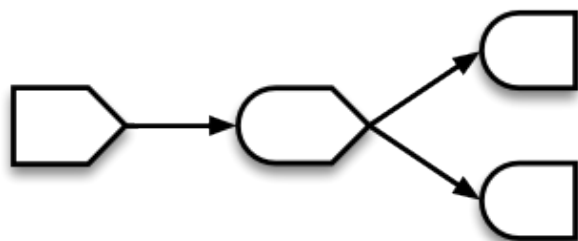
The amount of computation can be reduced by adding a reactive conductor in between the source and endpoints:

```
fib <- function(n) ifelse(n<3, 1, fib(n-1)+fib(n-2))

shinyServer(function(input, output) {
  currentFib <- reactive({ fib(as.numeric(input$n)) })

  output$nthValue <- renderText({ currentFib() })
  output$nthValueInv <- renderText({ 1 / currentFib() })
})
```

Here is the new graph structure:



Keep in mind that if your application tries to access reactive values or expressions from outside a reactive context — that is, outside of a reactive expression or observer — then it will result in an error. You can think of there being a

reactive “world” which can see and change the non-reactive world, but the non-reactive world can’t do the same to the reactive world. Code like this will not work, because the call to `fib()` is not in the reactive world (it’s not in a `reactive()` or `renderXX()` call) but it tries to access something that is, the reactive value `input$n`:

```
shinyServer(function(input, output) {
  # Will give error
  currentFib <- fib(as.numeric(input$n))
  output$nthValue <- renderText({ currentFib })
})
```

On the other hand, if `currentFib` is a function that accesses a reactive value, and that function is called within the reactive world, then it will work:

```
shinyServer(function(input, output) {
  # OK, as long as this is called from the reactive world:
  currentFib <- function() {
    fib(as.numeric(input$n))
  }

  output$nthValue <- renderText({ currentFib() })
})
```

## Summary

In this section, we’ve learned about:

- Reactive sources can signal objects downstream that they need to re-execute.
- Reactive conductors are placed somewhere in between sources and endpoints on the reactive graph. They are typically used for encapsulating slow operations.
- Reactive endpoints can be told to re-execute by the reactive environment, and can request *upstream* objects to execute.
- Invalidation arrows diagram the flow of invalidation events. It can also be said that the child node is a dependent of or takes a dependency on the parent node.

## Implementations of sources, conductors, and endpoints: values, expressions, and observers

We’ve discussed reactive sources, conductors, and endpoints. These are general terms for parts that play a particular role in a reactive program. Presently, Shiny has one class of objects that act as reactive sources, one class of objects that act as reactive conductors, and one class of objects that act as reactive endpoints, but in principle there could be other classes that implement these roles.

- Reactive values are an implementation of Reactive sources; that is, they are an implementation of that role.
- Reactive expressions are an implementation of Reactive conductors. They can access reactive values or other reactive expressions, and they return a value.
- Observers are an implementation of Reactive endpoints. They can access reactive sources and reactive expressions, and they don’t return a value; they are used for their side effects.

**Reactive value**  
(implementation of reactive source)



**Reactive expression**  
(implementation of reactive conductor)



**Observer**  
(implementation of reactive endpoint)



All of the examples use these three implementations, as there are presently no other implementations of the source, conductor, and endpoint roles.

## Reactive values

Reactive values contain values (not surprisingly), which can be read by other reactive objects. The `input` object is a `ReactiveValues` object, which looks something like a list, and it contains many individual reactive values. The values in `input` are set by input from the web browser.

## Reactive expressions

We've seen reactive expressions in action, with the Fibonacci example above. They cache their return values, to make the app run more efficiently. Note that, abstractly speaking, *reactive conductors* do not necessarily cache return values, but in this implementation, *reactive expressions*, they do.

A reactive expressions can be useful for caching the results of any procedure that happens in response to user input, including:

- accessing a database
- reading data from a file
- downloading data over the network
- performing an expensive computation

## Observers

Observers are similar to reactive expressions, but with a few important differences. Like reactive expressions, they can access reactive values and reactive expressions. However, they do not return any values, and therefore do not cache their return values. Instead of returning values, they have side effects – typically, this involves sending data to the web browser.

The `output` object looks something like a list, and it can contain many individual observers.

If you look at the code for `renderText()` and friends, you'll see that they each return a function which returns a value. They're typically used like this:

```
output$number <- renderText({ as.numeric(input$n) + 1 })
```

This might lead you to think that the observers *do* return values. However, this isn't the whole story. The function returned by `renderText()` is actually not an observer/endpoint. When it is assigned to `output$x`, the function returned by `renderText()` gets automatically wrapped into another function, which is an observer. The wrapper function is used because it needs to do special things to send the data to the browser.

## Differences between reactive expressions and observers

Reactive expressions and observers are similar in that they store expressions that can be executed, but they have some fundamental differences.

- Observers (and endpoints in general) respond to reactive *flush* events, but reactive expressions (and conductors in general) do not. We'll learn more about flush events in the next section. If you want a reactive expression to execute, it must have an observer as a descendant on the reactive dependency graph.
- Reactive expressions return values, but observers don't.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

GruntledEmployee

---

How does one represent (graphically) a conditional input (e.g., input widgets within a `conditionalPanel`)?

Jaroslaw Piskorski

---

Hi!

One remark - in the Summary section you talk about "Invalidation arrows". I don't think these are mentioned in the text before.

regards

Jarek

Bill Jackson

---

Re: "Weâ€™ll learn more about flush events in the next section." - I got to this article from a google search, can you add a link to where the next section materials are?

comments powered by [Disqus](#)

## 2.36 Stop reactions with isolate()

# Stop reactions with isolate()

ADDED: 06 JAN 2014

## Isolation: avoiding dependency

Sometimes it's useful for an observer/endpoint to access a reactive value or expression, but not to take a dependency on it. For example, if the observer performs a long calculation or downloads large data set, you might want it to execute only when a button is clicked.

For this, we'll use `actionButton`. We'll define a `ui.R` that is a slight modification of the one from 01\_hello – the only difference is that it has an `actionButton` labeled "Go!". You can see it in [action here](#).

The `actionButton` includes some JavaScript code that sends numbers to the server. When the web browser first connects, it sends a value of 0, and on each click, it sends an incremented value: 1, 2, 3, and so on.

```
shinyUI(pageWithSidebar(
  headerPanel("Click the button"),
  sidebarPanel(
    sliderInput("obs", "Number of observations:",
               min = 0, max = 1000, value = 500),
    actionButton("goButton", "Go!")
  ),
  mainPanel(
    plotOutput("distPlot")
  )
))
```

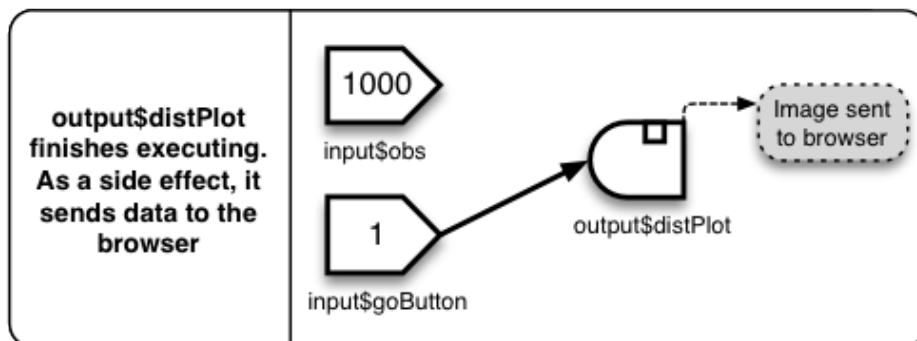
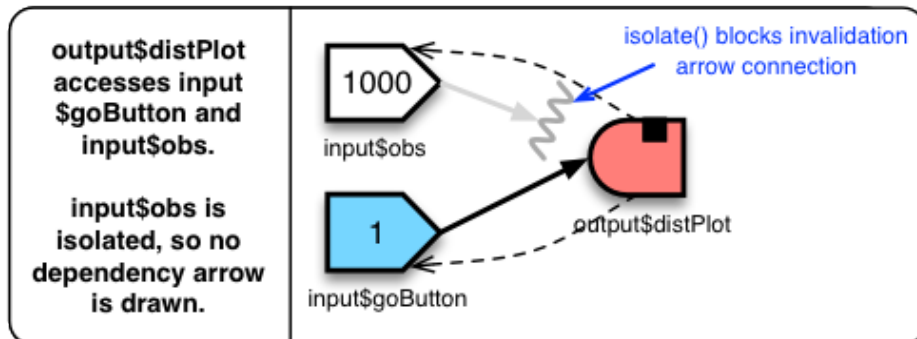
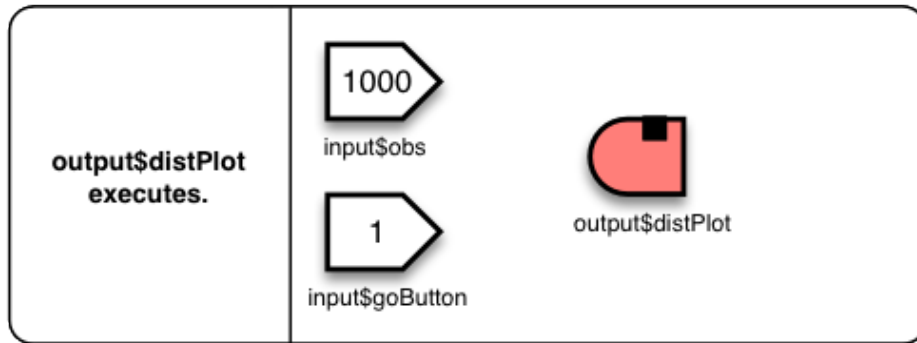
In our `server.R`, there are two changes to note. First, `output$distPlot` will take a dependency on `input$goButton`, simply by accessing it. When the button is clicked, the value of `input$goButton` increases, and so `output$distPlot` re-executes.

The second change is that the access to `input$obs` is wrapped with `isolate()`. This function takes an R expression, and it tells Shiny that the calling observer or reactive expression should not take a dependency on any reactive objects inside the expression.

```
shinyServer(function(input, output) {
  output$distPlot <- renderPlot({
    # Take a dependency on input$goButton
    input$goButton

    # Use isolate() to avoid dependency on input$obs
    dist <- isolate(rnorm(input$obs))
    hist(dist)
  })
})
```





In the `actionButton` example, you might want to prevent it from returning a plot the first time, before the button has been clicked. Since the starting value of an `actionButton` is zero, this can be accomplished with the following:

```
output$distPlot <- renderPlot({
  if (input$goButton == 0)
    return()

  # plot-making code here
})
```

Reactive values are not the only things that can be isolated; reactive expressions can also be put inside an `isolate()`. Building off the Fibonacci example from above, this would calculate the `_n_th` value only when the button is clicked:

```
output$nthValue <- renderText({
  if (input$goButton == 0)
    return()

  isolate({ fib(as.numeric(input$n)) })
})
```

It's also possible to put multiple lines of code in `isolate()`. For example here are some blocks of code that have equivalent effect:

```
# Separate calls to isolate -----
x <- isolate({ input$xSlider }) + 100
y <- isolate({ input$ySlider }) * 2
z <- x/y

# Single call to isolate -----
isolate({
  x <- input$xSlider + 100
  y <- input$ySlider * 2
  z <- x/y
})

# Single call to isolate, use return value -----
z <- isolate({
  x <- input$xSlider + 100
  y <- input$ySlider * 2
  x/y
})
```

In all of these cases, the calling function won't take a reactive dependency on either of the `input` variables.

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Yuji Zhang

Hi Garrett, great post! Thanks for sharing. I tested a little bit and understand that, without `isolate()`, any change in the reactive variable will immediately automatically trigger render to re-execute. With `isolate()`, the render only go check if the reactive variable is out of date when we tell it to re-execute.

This maybe quite subject, but I though if you add in the graph something like `(plot 500) / (plot 1000)`, it would be a lot easier to read...

Martin Johnson

Hello, I've been trying to use an action button to trigger the plotting of a line on an otherwise blank plot and has taken me ages to realise that the logic is slightly wrong in the example above (for the 2 examples under the schematic).

To get it to work for me I needed to use

*Stop reactions with isolate()*  
if(input\$goButton[1]==0)

not  
comments powered by Disqus

Shiny is an RStudio project. © 2014 RStudio, Inc.

## 2.37 Execution scheduling

# Execution scheduling

ADDED: 06 JAN 2014

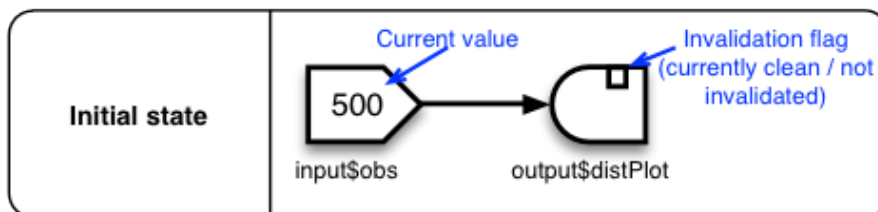
At the core of Shiny is its reactive engine: this is how Shiny knows when to re-execute each component of an application. We'll trace into some examples to get a better understanding of how it works.

## A simple example

At an abstract level, we can describe the `01_hello` example as containing one source and one endpoint. When we talk about it more concretely, we can describe it as having one reactive value, `input$obs`, and one reactive observer, `output$distPlot`.

```
shinyServer(function(input, output) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })
})
```

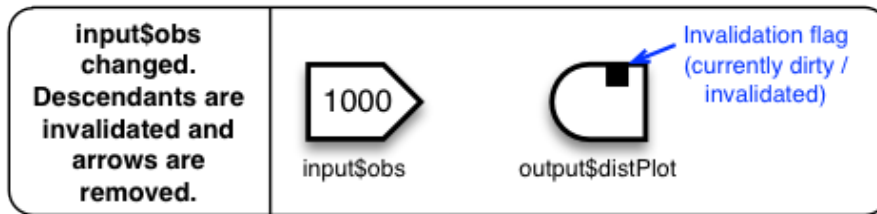
As shown in the diagram below, a reactive value has a value. A reactive observer, on the other hand, doesn't have a value. Instead, it contains an R expression which, when executed, has some side effect (in most cases, this involves sending data to the web browser). But the observer doesn't return a value. Reactive observers have another property: they have a flag that indicates whether they have been *invalidated*. We'll see what that means shortly.



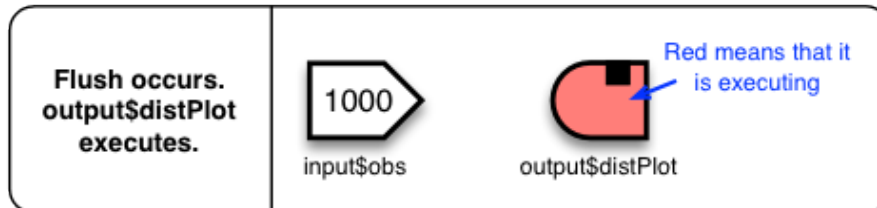
After you load this application in a web page, it be in the state shown above, with `input$obs` having the value 500 (this is set in the `ui.r` file, which isn't shown here). The arrow represents the direction that invalidations will flow. If you change the value to 1000, it triggers a series of events that result in a new image being sent to your browser.

When the value of `input$obs` changes, two things happen: \* All of its descendants in the graph are invalidated. Sometimes for brevity we'll say that an observer is *dirty*, meaning that it is invalidated, or *clean*, meaning that it is *not* invalidated. \* The arrows that have been followed are removed; they are no longer considered descendants, and changing the reactive value again won't have any effect on them. Notice that the arrows are dynamic, not static.

In this case, the only descendant is `output$distPlot`:



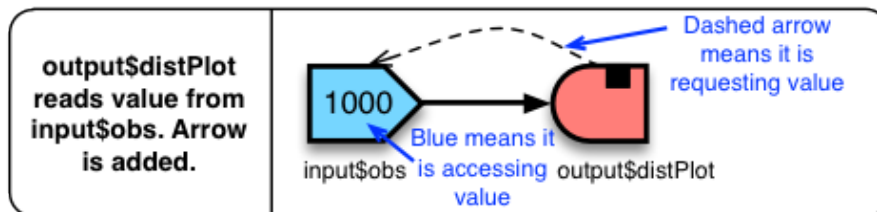
Once all the descendants are invalidated, a *flush* occurs. When this happens, all invalidated observers re-execute.



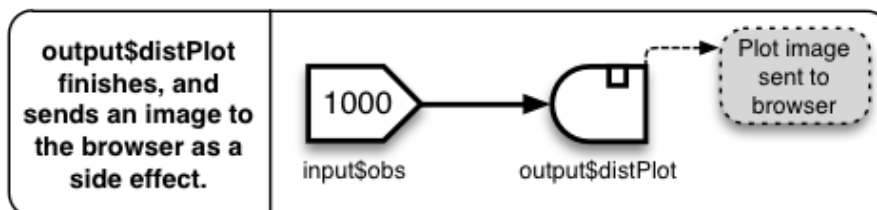
Remember that the code we assigned to `output$distPlot` makes use of `input$obs` :

```
output$distPlot <- renderPlot({
  hist(rnorm(input$obs))
})
```

As `output$distPlot` re-executes, it accesses the reactive value `input$obs`. When it does this, it becomes a dependent of that value, represented by the arrow. When `input$obs` changes, it invalidates all of its children; in this case, that's just `output$distPlot`.



As it finishes executing, `output$distPlot` creates a PNG image file, which is sent to the browser, and finally it is marked as clean (not invalidated).



Now the cycle is complete, and the application is ready to accept input again.

When someone first starts a session with a Shiny application, all of the endpoints start out invalidated, triggering this series of events.

## An app with reactive conductors

Here's the code for our Fibonacci program:

```

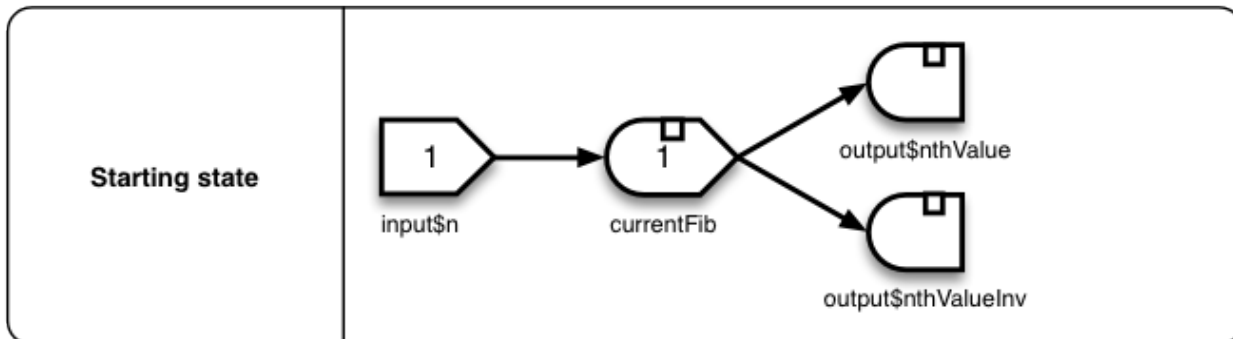
fib <- function(n) ifelse(n<3, 1, fib(n-1)+fib(n-2))

shinyServer(function(input, output) {
  currentFib <- reactive({ fib(as.numeric(input$n)) })

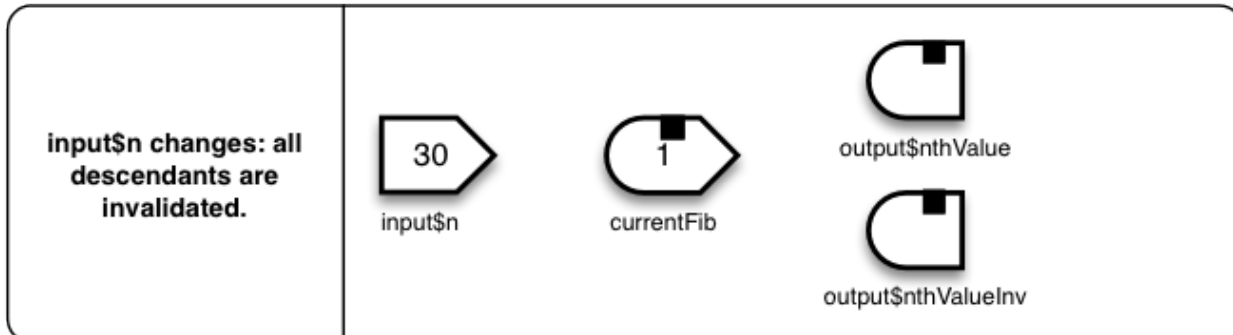
  output$nthValue <- renderText({ currentFib() })
  output$nthValueInv <- renderText({ 1 / currentFib() })
})

```

Here's the structure. It's shown in its state after the initial run, with the values and invalidation flags (the starting value for `input$n` is set in `ui.r`, which isn't displayed).

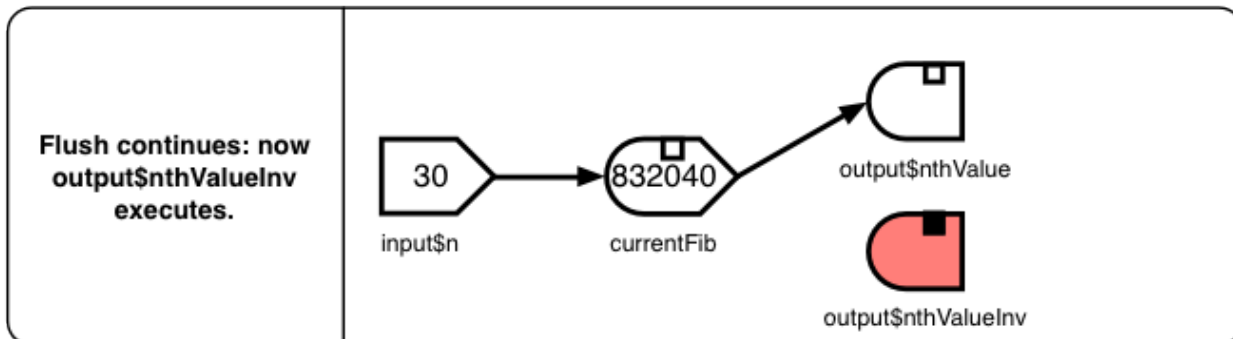
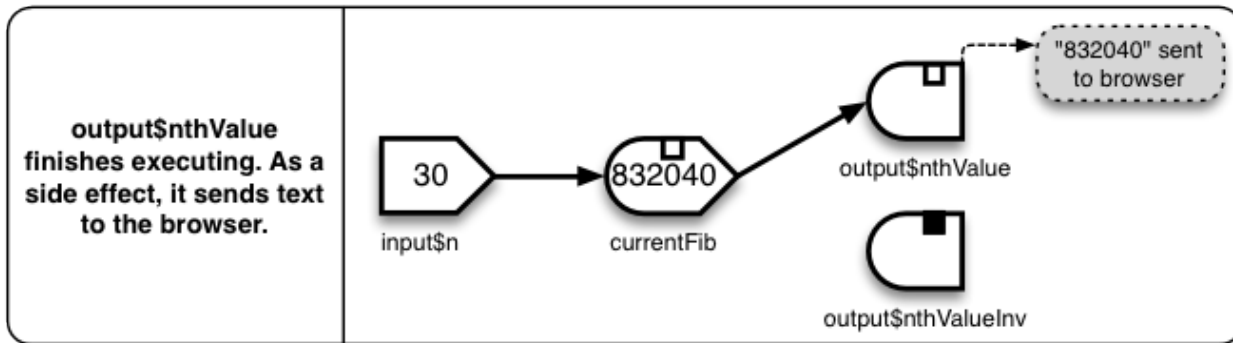


Suppose the user sets `input$n` to 30. This is a new value, so it immediately invalidates its children, `currentFib`, which in turn invalidates its children, `output$nthValue` and `output$nthValueInv`. As the invalidations are made, the invalidation arrows are removed:



After the invalidations finish, the reactive environment is flushed, so the endpoints re-execute. If a flush occurs when multiple endpoints are invalidated, there isn't a guaranteed order that the endpoints will execute, so `nthValue` may run before `nthValueInv`, or vice versa. The execution order of endpoints will not affect the results, as long as they don't modify and read non-reactive variables (which aren't part of the reactive graph).

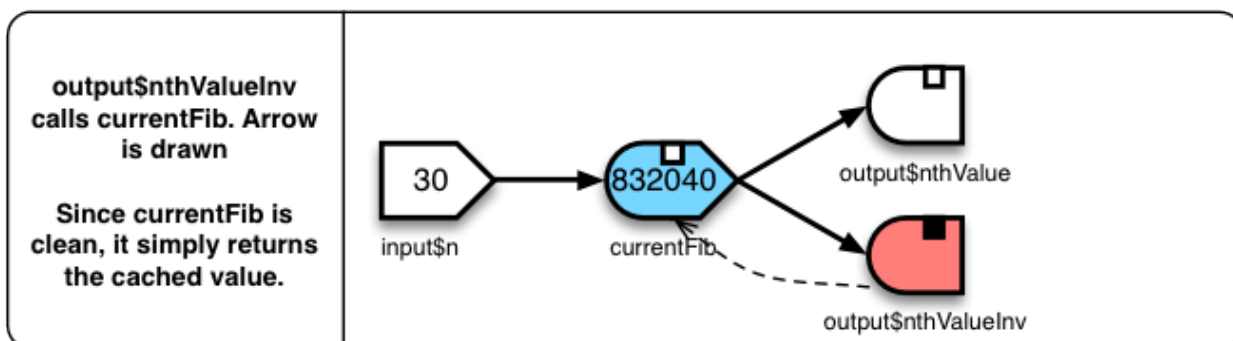
Suppose in this case that `nthValue()` executes first. The next several steps are straightforward:



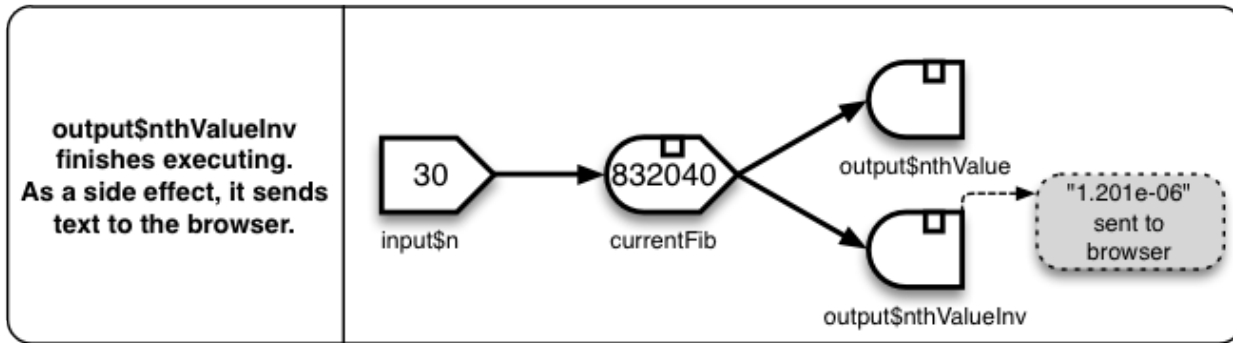
As `output$nthValueInv()` executes, it calls `currentFib()`. If `currentFib()` were an ordinary R expression, it would simply re-execute, taking another several seconds. But it's not an ordinary expression; it's a reactive expression, and it now happens to be marked clean. Because it is clean, Shiny knows that all of `currentFib`'s reactive parents have not changed values since the previous run `currentFib()`. This means that running the function again would simply return the same value as the previous run. (Shiny assumes that the non-reactive objects used by `currentFib()` also have not changed. If, for example, it called `Sys.time()`, then a second run of `currentFib()` could return a different value. If you wanted the changing values of `Sys.time()` to be able to invalidate `currentFib()`, it would have to be wrapped up in an object that acted as a reactive source. If you were to do this, that object would also be added as a node on the reactive graph.)

Acting on this assumption, that clean reactive expressions will return the same value as they did the previous run, Shiny caches the return value when reactive expressions are executed. On subsequent calls to the reactive expression, it simply returns the cached value, without re-executing the expression, as long as it remains clean.

In our example, when `output$nthValueInv()` calls `currentFib()`, Shiny just hands it the cached value, 832040. This happens almost instantaneously, instead of taking several more seconds to re-execute `currentFib()`:



Finally, `output$nthValueInv()` takes that value, finds the inverse, and then as a side effect, sends the value to the browser.



## Summary

In this section we've learned about:

- Invalidation flags: reactive expressions and observers are invalidated (marked dirty) when their parents change or are invalidated, and they are marked as clean after they re-execute.
- Arrow creation and removal: After a parent object invalidates its children, the arrows will be removed. New arrows will be created when a reactive object accesses another reactive object.
- Flush events trigger the execution of endpoints. Flush events occur whenever the browser sends data to the server.

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Yuji Zhang

nice article! Thank you Garrett!

A question: how do we access the flag of a reactive object? For example in server I have something like:

```
# get input data from file upload
```

```
getData = reactive({
  inFile = input$file1
  if (is.null(inFile)) return(NULL)
  read.csv(inFile$datapath)
})
```

```
# display input data
output$table = renderTable({ getData() })
# do calculation based on input data
```

```
result = reactive({ BigCalculation( getData() ) })
```

When some input data gets in, Shiny will run renderTable and the Calculation at the



same time. As a result renderable will be slow. I want renderable to run first, and  
comments powered by [Disqus](#)

Shiny is an [RStudio](#) project. © 2014 RStudio, Inc.

## 2.38 How to understand reactivity in R

# How to understand reactivity in R

ADDED: 21 MAY 2015

BY: GARRETT GROLEMUND

Reactivity is what makes your Shiny apps responsive. It lets the app instantly update itself whenever the user makes a change. You don't need to know how reactivity occurs to use it (just follow the steps laid out in [Lesson 4](#)), but understanding reactivity will make you a better Shiny programmer. You'll be able to

1. create more efficient and sophisticated Shiny apps, and
2. avoid the errors that come from misusing reactive values in R (which is easy to do!).

Let's take a look at reactivity by building a very simple Shiny app. You can use the `app.R` file below to make this app.



```
# app.R
```

```
library(shiny)
```

```
ui <- fluidPage(  
  headerPanel("basic app"),  
  
  sidebarPanel(  
  
    sliderInput("a",  
      label = "Select an input to display",  
      min = 0, max = 100, value = 50)  
  ),  
  
  mainPanel(h1(textOutput("text")))
```

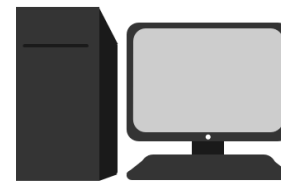
```

server <- function(input, output) {
  output$text <- renderText({
    print(input$a)
  })
}

```

```
shinyApp(ui = ui, server = server)
```

The app sets up a very basic reactive system: it has a single input value that can change (`input$a`); it has a single output value that can respond (`print(input$a)`); and it has a server that can oversee the process. Every Shiny app will have these same components, although most apps will have multiple input values and multiple output expressions.



Server

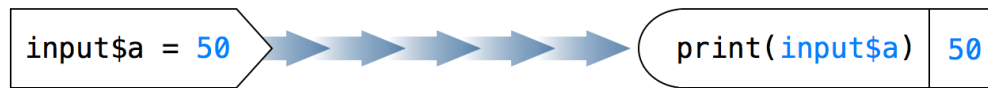
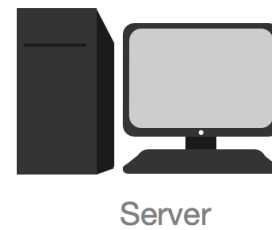
`input$a = 50`

<code>print(input\$a)</code>	50
------------------------------	----

When you move the slider, you can see reactivity in action: the number to the left of the slider automatically updates to show the current slider value. This may seem simple, but it is very special. Let's look at why.

## Reactivity is unexpected

Reactivity creates the illusion that changes in input values automatically flow to the plots, text, and tables that use the input—and cause them to update. You can think of this flow as a current of electricity, or a stream of water that *pushes* information from input to output. You saw this illusion in action when you moved the slider bar. Changes in the slider bar seemed to automatically propagate to the number beside the bar.



This illusion is amazing, because information in R only travels through *pull* mechanisms, not *push* mechanisms. In other words, if you have a simple R expression like `{a + 1}`, R will retrieve information from `a` to evaluate the expression, but R won't modify the result of `{a + 1}` if you later change the value of `a`.

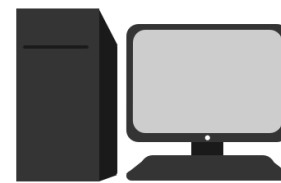
```
a <- 1
a + 1
## 2

a <- 2
## (nothing happens)
```

Pictorially, the system looks like this. Notice that the arrow in the diagram goes from right to left. This is to imply that the expression on the right is doing the work. It is telling R to look up the value of `a`. `a` is just sitting passively in memory.



For our app, this suggests that R should look up the value of `input$a` once, print the value, and then not notice when `input$a` changes.



Server



Incredibly, this isn't what happens, as you saw above. Reactivity appears to reverse the flow of information in R. How does it do that?

## What is reactivity?

Think of reactivity as a magic trick: reactivity creates the illusion that one thing is happening, when in fact something else is going on. The illusion is that information is being pushed from inputs to outputs (or at least that inputs and outputs are linked in an inseparable way). The reality is that Shiny is re-running your R expressions in a carefully scheduled way.

I've prepared four maxims to help you understand this process. We'll look at each of them (and the process itself), with a simple thought experiment: how could we recreate our basic app without breaking the rules of R?

Here are our maxims

1. *R expressions update themselves, if you ask*
2. *Nothing needs to happen instantly*
3. *The app must do as little as possible*
4. *Think carrier pigeons, not electricity*

### 1. R expressions update themselves, *if you ask*

Reactivity ensures that the output of `print(input$a)` is always up to date, but what does it mean for output to be *out of date*? Let's consider output – and the expression that made it – to be out of date if one of the objects in the expression has been given a new value since the expression was called. For example, at the end of this code, the expression `print(a)` is out of date. The last time `print(a)` ran, `a` was 1.

```
a <- 1
print(a)
## 1
```

```
a <- 2
```

Updating an out of date expression is not hard: you just need to re-run the expression. Everything in R updates itself each time it is run. This isn't reactivity; it's just standard R behavior.

```
a <- 1
print(a)
## 1
```

```
a <- 2  
print(a)  
## 2
```

Think of it like this: every time you run an expression, the expression updates itself. It looks up the current value of each object that it uses and computes new output. However, you must tell R to run the expression for this to happen because R uses a style of execution known as *lazy evaluation*. In other words, R will not execute an expression until you force it to.

You could use this behavior – and nothing else – to create a reactive web app. All you need to do is manually re-run the expressions in the app whenever the user makes a change.

## 2. Nothing needs to happen instantly

How quickly do you need to re-run an expression after a user makes a change? If the update appears instantaneous, the user will feel like they caused it. In other words, the update will create the illusion of reactivity. However, humans aren't very good at noticing small windows of time. You could actually let a few microseconds pass between change and update and your user wouldn't notice. This suggests a new feature for our plan.

Instead of watching the user (which would require logistics we haven't thought through), you could just have your server re-run each expression in the app every few microseconds. That way whenever the user makes a change, an update will follow within a few microseconds. If you re-run every expression in the app, you don't even need to worry about which part of the app the user is changing.

What if the user doesn't make a change? Then the expressions will re-compute their previous results and the app will appear to be in the same state it was before.

This plan creates the illusion of reactivity without violating the rules of R. Information still travels from input to output in a pull fashion. For example, `print(input$a)` only learns the new value of `input$a` because the server re-executes `print(input$a)`. Since `print(input$a)` is re-executed so often, it seems to learn of the change very fast, as if it were connected to `input$a` or as if `input$a` pushed its new value to `print(input$a)`.

Shiny uses this approach to create reactivity. That is why your R session becomes busy when you launch a Shiny app. Your server is using the R session to monitor the app and re-run expressions. However, Shiny takes this approach one step further. It creates an alert system that lets Shiny know exactly which expressions need to be re-run.

## 3. The app must do as little as possible

It takes a very powerful computer to re-run every expression in an app every few microseconds without bogging down. If you used our approach in reality, your app would quickly become slow and unresponsive, which would destroy the illusion of reactivity.

If you want your updates to run so fast that they appear instantaneous, you'll need to save your computer power for just the expressions that are out of date. However, your app may use hundreds of expressions. How will you know which ones are out of date?

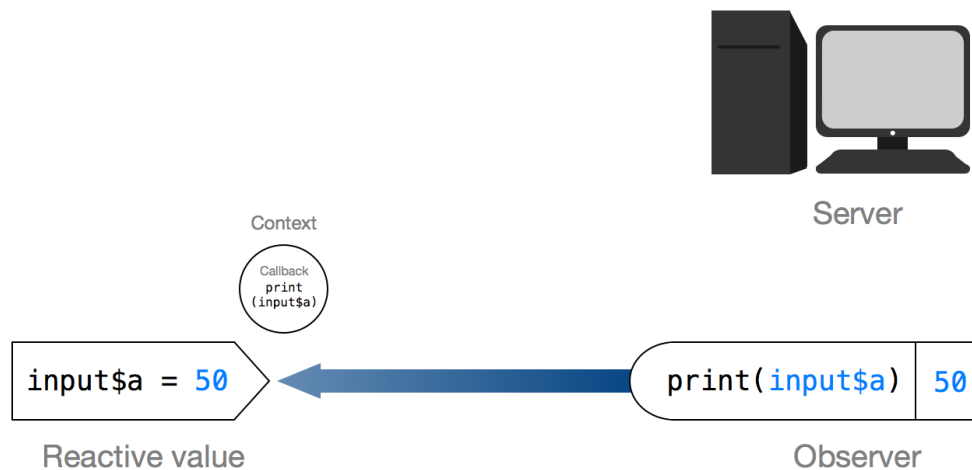
Shiny solves this problem by creating a system of alerts that lets the server know when an expression becomes out of date. The server still checks in on your app every few microseconds, but instead of re-running each expression each time, it only runs the expressions that the alert system has flagged as out of date. If no alerts have appeared, the server does not have to run anything at all. It can rest until the next check. If alerts have appeared, the server runs all of the expressions that are out of date at that moment, an event known as a *flush*.

This alert system is the key to reactivity. It allows your server to update your app as fast as possible, so fast that changes seem to travel instantly from inputs to outputs. Let's not try to brainstorm our own alert system. Instead let's examine the system that Shiny uses.

## 4. Think carrier pigeons, not electricity

The details of the alert system are fairly complicated. If they sound confusing in this next paragraph, don't worry. We're going to break them down step by step with an analogy that will make them more transparent.

Shiny implements reactivity with two special object classes, `reactivevalues` and `observers`. In our example `input$a` is a reactive values object and `print(input$a)` is an observer. These two classes behave like regular R values and R expressions with a few exceptions. Whenever an observer uses a reactive value, it registers a *reactive context* with the value. This context contains an expression to be run if the value ever changes. The expression is called a *callback* and it is always a command to re-run the observer. A single reactive value can hold many contexts if multiple observers use that value.

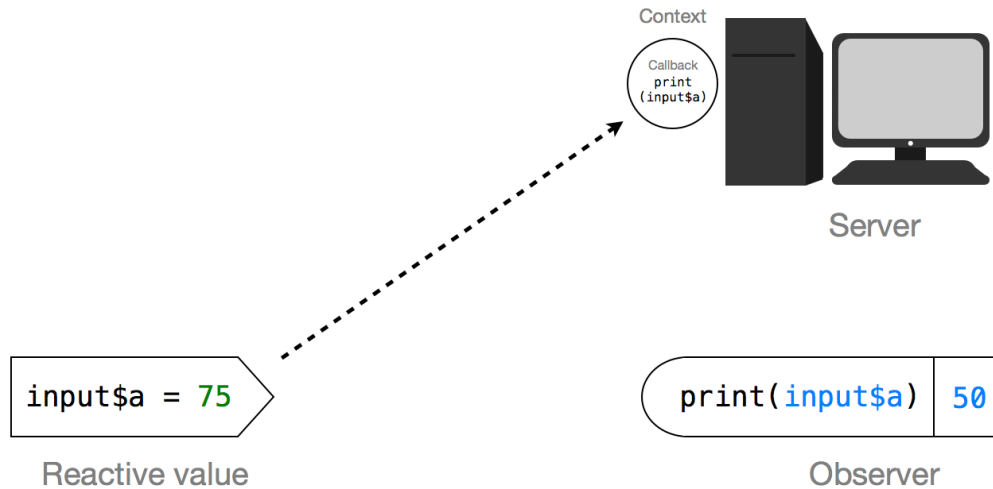


When the value of a reactive values object changes, the object will send any callbacks that it has collected to the server. Lets look at how this happens.

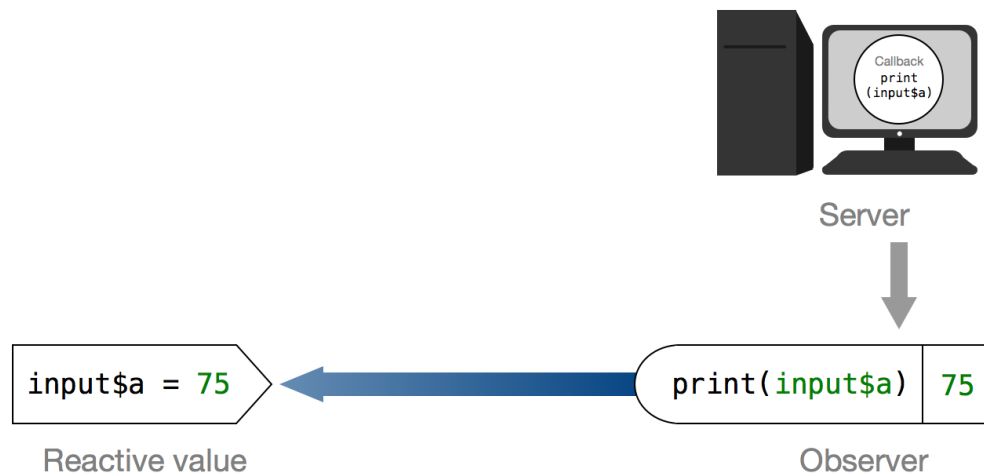
A reactive values object is a type of list. In R, you change the value of a list by calling a setter function, either `$<-` or `[[<-`. These are the functions that R calls in the background whenever you combine the assignment arrow, `<-`, with the subsetting symbols `$` or `[[ ]]`. For example, R will call `$<-` when you run the second line of code below.

```
myList <- list(a = 1, b = 2)
myList$b <- 3
```

Since reactive values objects are a special class of object, they have their own setter methods. The setter methods of reactive values objects (`$<-` and `[[<-`) include instructions to send any callbacks that the reactive value has received to the server. If the reactive values object is set to a new value, it executes these instructions and the server receives the callbacks.



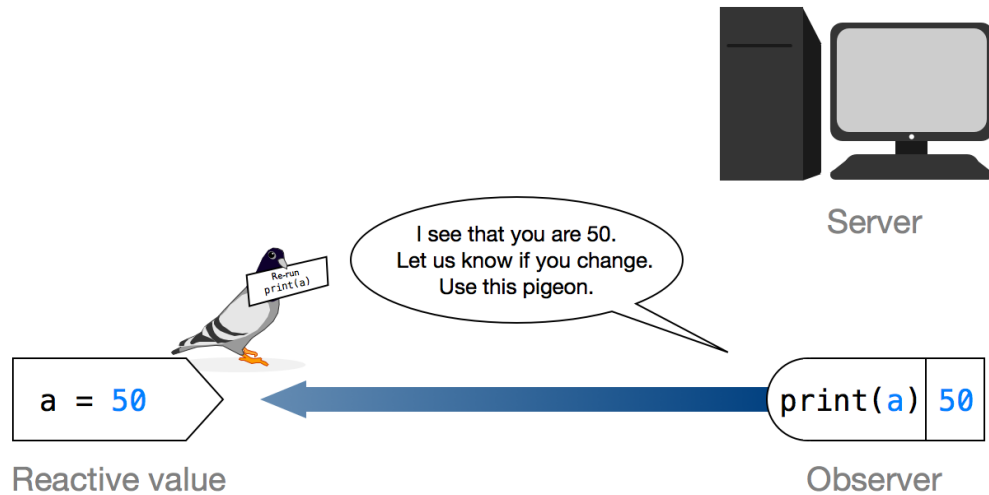
The server saves the callbacks in a queue which acts as a list of observers that have become out of date. On the next flush, the server runs each callback in the queue which re-runs each out of date observer, which restarts the cycle.



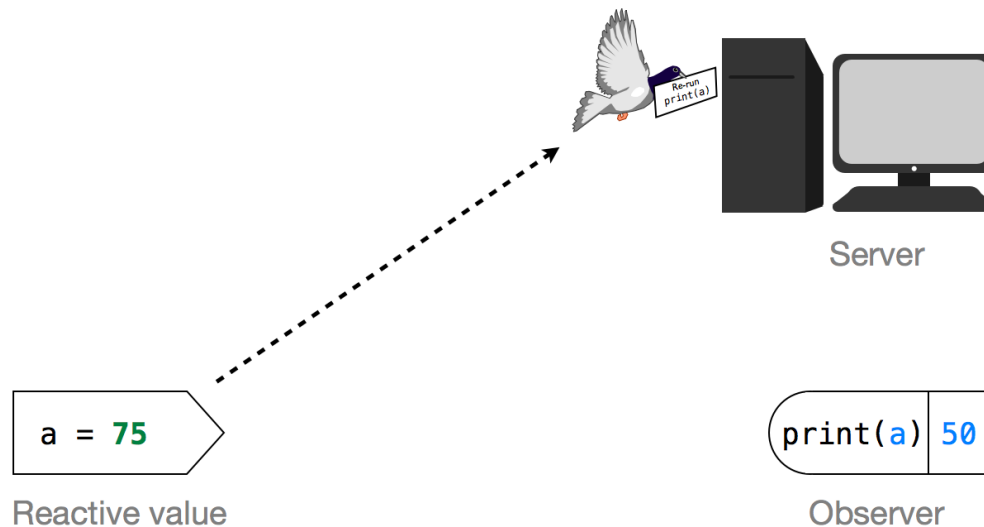
If this seems complicated, think of reactivity as a carrier pigeon system between three objects. If you don't know what carrier pigeons are, check out this [link](#) – it's pretty fascinating. Basically, you can take a carrier pigeon anywhere and when you release it, it will always fly back to the same location. Soldiers on the move used carrier pigeons to deliver messages to their headquarters. We're going to use them to deliver messages to the server.

A context is like a virtual carrier pigeon that an observer leaves with a reactive value. The context contains a message (its callback) that it will deliver to the server when released. The observer writes this message for the context, and it is a simple instruction to re-run the observer. An observer leaves behind a context each time it looks up a reactive values object. In fact, a reactive values object will return an error if an expression tries to access its value without leaving behind a context.



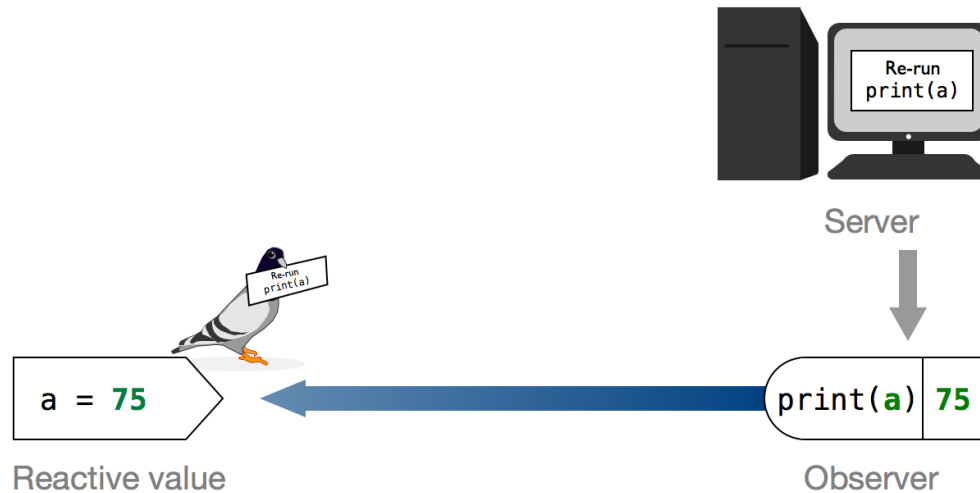


When a reactive values object receives a context, it simply holds onto it. It will collect multiple contexts if other observers look up the object as well. If the reactive value object ever changes, it will release all of the contexts it has collected (a process known as *invalidating the contexts*). This behavior is like releasing carrier pigeons, the pigeons are free to fly back to the server and deliver the callbacks that they have been holding onto. When a context is invalidated, it places its callback in the server's queue to be run on the next flush. Then the context ceases to be relevant, just like a pigeon that has delivered its message.



The callback of a context is an R command that when run, will re-execute the observer that created the context. This will cause the observer to update itself with the new value of the reactive values object. When the server checks in on the app, all it needs to do is run any callbacks that have arrived. This will automatically update the app.

Running the callbacks also sets up a new reactive cycle. When an observer is re-run, it looks up the reactive value objects that it uses, which causes it to register new contexts with each value. In short, the observer leaves a new homing pigeon behind and the cycle is ready to repeat itself.



This system enables reactivity because it lets your server work fast enough to create the illusion of instant responses. Instead of re-running every expression in your app every few seconds, the server only needs to check its queue for new callbacks. The result is the quick, responsive updates you see in your Shiny app.

Now you know how reactivity works in Shiny. Notice that this system doesn't ask R to behave in a new way. Your observers are still looking up information from the reactive values. The values are not being pushed to the observer like a flow of electricity, or a stream—they only appear to be doing that. The key to this system is speed. Shiny enacts the pull mechanisms of R so fast that they look like push mechanisms.

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)  
[Chrome](#)  
[Internet Explorer 10+](#)  
[Safari](#)

Jackie C

There is some valuable information in this article, but IMHO they are really not making it easier to understand with the carrier pigeon stuff. There has to be a short and simpler explanation.

Something like...

- Reactive Values(inputs) contain 2 pieces of data,
  - 1) the actual value and
  - 2) a list of all reactive observers (outputs) which depend on that value.
- The shiny server looks for updates to observers in a iterative cycle, once per cycle it checks its update queue to see if any of the observers need to be updated.
- The queue is a list of observers that will need to be updated that iteration.
- The list is empty if nothing has changed since the last iteration.
- Each time an reactive value is changed by the user:

- 1) the value is updated and
  - 2) each observer in its list of dependent observes is added to the servers update queue.
- They will be updated on the next iteration.

comments powered by [Disqus](#)

## 2.39 Write error messages for your UI with

# Write error messages for your UI with validate

ADDED: 20 MAY 2014

BY: GARRETT GROLEMUND WITH JOE CHENG

*Note: This article requires Shiny version 0.9.1.9008 (not yet released). You can install the development version of Shiny ( $\geq 0.9.1.9008$ ) with: `devtools::install_github("shiny", "rstudio")`.*

Have you ever seen a Shiny app go wrong? Shiny delivers a bold red error message to your user. This message is often unhelpful because it mentions things that *you* may understand as a developer, but that your user may not.

This article will show you how to craft “validation errors,” errors designed to lead your user through the UI of your Shiny app. Validation errors are user-friendly and, unlike the bold red error message, pleasing to the eye. Best of all, validation errors respond directly to your user’s input.

We’ll start by creating an app that quickly returns an error message. The `server.R` and `ui.R` scripts below make a simple app that displays a table and draws a plot. To make this app, copy these scripts into your [working directory](#) and run:

```
library(shiny)
runApp()
```

*Note: these files need to be the only ones named `server.R` and `ui.R` in your working directory.*

```
## server.R
```

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

  data <- reactive({ get(input$data, 'package:datasets') })

  output$plot <- renderPlot({
    hist(data()[, 1], col = 'forestgreen', border = 'white')
  })

  output$table <- renderTable({
    head(data())
  })

})
```

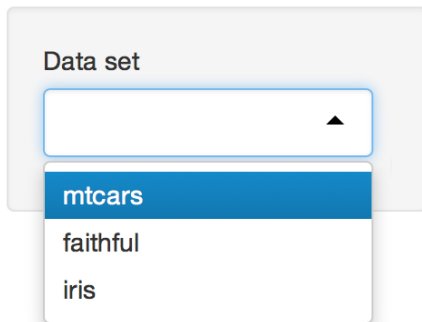
```
## ui.R
```

```
shinyUI(fluidPage(
```

```
sidebarLayout(  
  sidebarPanel(  
    selectInput("data", label = "Data set",  
              choices = c("", "mtcars", "faithful", "iris"))  
  ),  
  
  # Show a plot of the generated distribution  
  mainPanel(  
    tableOutput("table"),  
    plotOutput("plot")  
  )  
)  
)  
)
```

When you first launch the app, it should look like this picture:

## Validation App



**Error: invalid first argument**

**Error: invalid first argument**

The app displays a table and then draws a plot beneath it, but both the table and plot depend on the value of the select box. Until your user selects a data set, the app will display two red error messages.

Our goal is to replace these error messages. We want messages that:

- Help the user understand what went wrong
- Do not cause panic (i.e., are not bold red)

You can write these more helpful messages with Shiny's new `validate` function.

Note: `validate` requires Shiny version 0.9.1.9008 or greater. You can install the most recent development version of Shiny with: `devtools::install_github("shiny", "rstudio")`.

## validate

`validate` tests a condition and returns a validation error if the test fails. Validation errors are designed to interact with the Shiny framework in a pleasing way. Shiny will:

- recognize a validation error
- display a validation error in a neutral grey color
- pass a validation error to any reactive expression or observer object that depends on it

`validate` takes one or more specially formatted arguments. You can provide these arguments `need`, a new function designed to work with `validate`. You can also provide these arguments with your own functions if you like.

## need

`need` provides a simple way to tell Shiny what to test and what to return if the test goes wrong. `need` uses two arguments

- An R expression that returns `TRUE` or `FALSE`.
- A character string. Shiny will display this string as a validation error message if the R expression returns `FALSE`. If the R expression returns `TRUE`, Shiny treats the validation test as if it passed and continues with the app.

Let's put these ideas all together.

You can create a complete validation test by calling `validate` and passing it the output of `need`:

```
validate(
  need(input$data != "", "Please select a data set")
)
```

The validation test above checks whether an object named `input$data` is an empty string. If the object is an empty string, the test returns the message: "Please select a data set."

To use this validation test in your app, place it at the start of any `reactive` or `render*` expression that calls `input$data`. In our app, our validation test appears in this `server.R` script after `reactive({`:

```
## server.R

shinyServer(function(input, output) {

  data <- reactive({
    validate(
      need(input$data != "", "Please select a data set")
    )
    get(input$data, 'package:datasets')
  })

  output$plot <- renderPlot({
    hist(data()[, 1], col = 'forestgreen', border = 'white')
  })

  output$table <- renderTable({
    head(data())
  })

})
```

Modify your script and relaunch the app. Now Shiny runs the validation test before it uses `input$data` and encounters an error, and the app does not show the bold red error message. Instead it displays your user-friendly validation error message.

## Validation App

A screenshot of a web form. It features a label 'Data set' above a dropdown menu. The dropdown menu is currently empty, showing only a downward-pointing arrow.

Please select a data set

Please select a data set

### Best practices

Notice that neither `output$plot` nor `output$table` call the validation test. However, both the plot and table objects display the validation error message.

When these objects call `data()`, `data()` retrieves the value of the reactive expression `data`. In our example, the value of the reactive expression `data` is the validation error message because the validation test fails.

You can use this arrangement to write efficient apps: one that fail fast and in a useful manner. To do this:

1. Separate `input` objects that might cause trouble into their own reactive expressions.
2. Have each reactive expression run a validation test on the `input`.
3. Arrange for other objects to access the `input` by calling the reactive expression.

This arrangement will let you use one validation test per input to catch any errors generated by your apps UI.

### Labels

You do not have to provide `need` with a full message to display. If you prefer, you can skip the message and pass `need` a `label` argument. If you do, `need` will construct a message by adding “must be provided” to the end of your label.

You can see this behavior in this app:

## Validation App

A screenshot of a web form. It features a label 'Data set' above a dropdown menu. The dropdown menu is currently empty, showing only a downward-pointing arrow.

data set must be provided

data set must be provided

It uses this `server.R` file.

```
## server.R

shinyServer(function(input, output) {

  data <- reactive({
    validate(
```

```

    need(input$data != "", label = "data set")
  )
  get(input$data, 'package:datasets')
})

output$plot <- renderPlot({
  hist(data()[, 1], col = 'forestgreen', border = 'white')
})

output$table <- renderTable({
  head(data())
})

})

```

## Errors vs. Validation errors

Validation tests do not remove the possibility of other types of errors. Shiny will still display system error messages in the familiar bold red font (designed to catch the developer's eye) when they happen.

For example, Shiny will display a red error message if the R expression in `need` returns an error. In the code below, the `need` expression calls the object `foo`, but `foo` does not exist.

```

## server.R

shinyServer(function(input, output) {

  data <- reactive({
    validate(
      need(input$data != foo, "Please select a data set")
    )
    get(input$data, 'package:datasets')
  })

  output$plot <- renderPlot({
    hist(data()[, 1], col = 'forestgreen', border = 'white')
  })

  output$table <- renderTable({
    head(data())
  })

})

```

Since Shiny cannot find `foo`, it displays a system error message.



# Validation App

**Error: object 'foo' not found**

**Error: object 'foo' not found**

You can prevent validation tests from generating system errors by wrapping the first argument of `need` in `try`:

```
validate(
  need(try(input$data != foo), "Please select a data set")
)
```

`try` returns a try error if `input$data != foo` fails. `need` treats try errors the same way it treats `FALSE`s. If the first argument of `need` returns a try error, `need` returns a validation error that displays its message.

Several other types of output also trigger `need` to return a validation error. You can write the first argument of `need` to return any output from the list below (if the validation fails). `need` returns a validation error for each of these outputs.

- `FALSE`
- `NULL`
- `""`
- An empty atomic vector
- An atomic vector that contains only missing values
- A logical vector that contains all `FALSE` or missing values
- An object of class `try-error`
- A value that represents an unclicked `actionButton`

## Write your own tests

Shiny power users can write their own `need` functions. This can be useful if you test for the same conditions across many apps. You can use any function in place of `need` as long as your function returns one of three objects:

1. `NULL`
2. A character string
3. `FALSE`

`validate` will run the function and then proceed in one of three ways.

- If your function returns `NULL`, `validate` will consider the check to have passed, and proceed as normal.
- If your function returns a character string, `validate` will consider the check to have failed and will return the string as a validation error to be displayed.
- If your function returns `FALSE`, `validate` will fail silently. Shiny will not continue with the app (which would result in a red error message), but it will not display a grey validation error message either.

Here is an example of a `need` type function:

```
not_mtcars <- function(input) {
  if (input == "mtcars") {
    "Choose another data set. No mtcars please!"
  }
}
```

Write error messages for your UI with

```

} else if (input == "") {
  FALSE
} else {
  NULL
}
}
}

```

Here is the function in use:

```

## server.R

not_mtcars <- function(input) {
  if (input == "mtcars") {
    "Choose another data set. No mtcars please!"
  } else if (input == "") {
    FALSE
  } else {
    NULL
  }
}

shinyServer(function(input, output) {

  data <- reactive({
    validate(
      not_mtcars(input$data)
    )
    get(input$data, 'package:datasets')
  })

  output$plot <- renderPlot({
    hist(data()[, 1], col = 'forestgreen', border = 'white')
  })

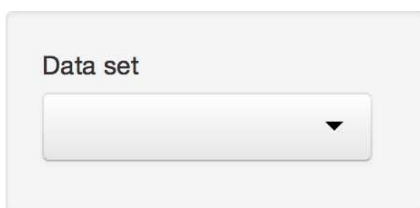
  output$table <- renderTable({
    head(data())
  })

})

```

When this app first launches, it looks like this app:

## Validation App



The image shows a rectangular box with a light gray background. Inside the box, at the top left, is the text "Data set". Below this text is a white rectangular dropdown menu with a small black downward-pointing triangle on its right side.

When you select `mtcars` in the select box, the app looks like this:

## Multiple conditions

You can check multiple conditions in a single `validate` call. To do this, pass `validate` multiple `need` statements (or similar functions, as described above) separated by commas. Shiny will display the message of every condition that fails.

This code contains three conditions that fail and one that passes:

```
## server.R

shinyServer(function(input, output) {

  data <- reactive({
    validate(
      need(input$data != "", "Please select a data set"),
      need(input$data %in% c("mtcars", "faithful", "iris"),
        "Unrecognized data set"),
      need(input$data, "Input is an empty string"),
      need(!is.null(input$data),
        "Input is not an empty string, it is NULL")
    )
    get(input$data, 'package:datasets')
  })

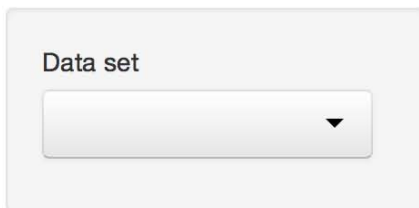
  output$plot <- renderPlot({
    hist(data()[, 1], col = 'forestgreen', border = 'white')
  })

  output$table <- renderTable({
    head(data())
  })

})
```

When you run it, the code creates this app:

## Validation App



The screenshot shows a Shiny app interface with a label 'Data set' above a dropdown menu. The dropdown menu is currently empty, indicating that no data set has been selected.

Please select a data set  
Unrecognized data set  
Input is an empty string

Please select a data set  
Unrecognized data set  
Input is an empty string

### %then%

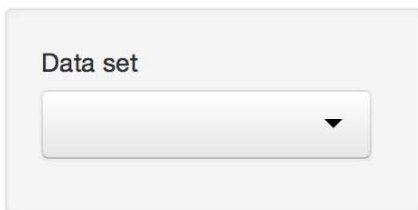
If you prefer to display one validation error message at a time, you may chain conditions together with the `%then%` operator.

Note: `%then%` does not exist in current preview implementations of Shiny. You can use the first line of code below to create a `%then%` operator.

```
`%then%` <- shiny::`%OR%`  
  
shinyServer(function(input, output) {  
  
  data <- reactive({  
    validate(  
      need(input$data != "", "Please select a data set") %then%  
      need(input$data %in% c("mtcars", "faithful", "iris"),  
        "Unrecognized data set") %then%  
      need(input$data, "Input is an empty string") %then%  
      need(!is.null(input$data),  
        "Input is not an empty string, it is NULL")  
    )  
    get(input$data, 'package:datasets')  
  })  
  
  output$plot <- renderPlot({  
    hist(data()[, 1], col = 'forestgreen', border = 'white')  
  })  
  
  output$table <- renderTable({  
    head(data())  
  })  
  
})
```

Shiny will display only the message of the first condition that fails. Here is an example:

## Validation App



Please select a data set

Please select a data set

Be careful not to use `%then%` in a way that might frustrate your user. A user may not enjoy fixing one validation error to find another (and then another) take its place.

## Style validation errors

Once you create a validation test, you can style its output with CSS (just as you can [style any element](#) in the Shiny user-interface).

Validation errors are HTML div objects with the class `shiny-output-error-validation`. Provide a CSS style for this class to change the appearance of every validation error message. For example, this `ui.R` script adds CSS that colors the messages green.

*Note: if your `server.R` script matches the last script (above), you need to select `mtcars` in your select box before*

```
## ui.R

shinyUI(fluidPage(

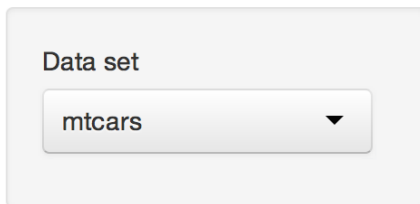
  tags$head(
    tags$style(HTML("
      .shiny-output-error-validation {
        color: green;
      }
    "))
  ),

  titlePanel("Validation App"),

  sidebarLayout(
    sidebarPanel(
      selectInput("data", label = "Data set",
        choices = c("", "mtcars", "faithful", "iris"))
    ),

    # Show a plot of the generated distribution
    mainPanel(
      plotOutput("plot"),
      tableOutput("table")
    )
  )
))
```

## Validation App



Choose another data set. No mtcars please!

Choose another data set. No mtcars please!

If you would like to style an individual validate message, give the message its own class with the `errorClass` of `validate`. Shiny will assign the message a class that begins with “shiny-output-error-“ and ends with the character string that you pass `errorClass`.

For example, this `validate` call returns a message of class “shiny-output-error-myclass” that you can style with CSS.

```
validate(
  need(input$data != "", "Please select a data set"),
  errorClass = "myClass"
)
```

## Recap

You can make your Shiny apps more attractive and user friendly with `validate`. `validate` tests inputs and delivers messages to your user, which creates an agreeable alternative to Shiny's default error messages.

Pair `validate` with one or more `need` calls to validate an input. You need to validate an input only once (in a `reactive` or `render*` expression). Shiny will pass the valuation results to any observer or expression that calls upon the input.

You can personalize validation error messages by writing your own `need` functions or by styling validation output with CSS.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

aquacalc

---

Just what I was looking for.

Thanks for adding these features and explaining their use in this article with practical examples.

Garrett

---

Aquacalc,

I'm glad you found this useful. We're very happy to get the word out about `validate`. It's such a useful feature.

Amos

---

Can you explain how this interacts with `ggvis`? It seems that if `ggvis` is passed a reactive closure as its data (i.e., `myData` instead of `myData()`), that a validation failure in the closure or in any reactive dependency of the closure will not get propagated, and shiny either fails with an error or the plot just freezes. I tried asking about this [comments powered by Disqus](#)

## 2.40 Scoping rules for Shiny apps

# Scoping rules for Shiny apps

ADDED: 06 JAN 2014

## Scoping

Where you define objects will determine where the objects are visible. There are three different levels of visibility that you'll want to be aware of when writing Shiny apps. Some objects are visible within the `server.R` code of each user session; other objects are visible in the `server.R` code across all sessions (multiple users could use a shared variable); and yet others are visible in the `server.R` and the `ui.R` code across all user sessions.

## Per-session objects

In `server.R`, when you call `shinyServer()`, you pass it a function `func` which takes two arguments, `input` and `output`:

```
shinyServer(func = function(input, output) {  
  # Server code here  
  # ...  
})
```

The function that you pass to `shinyServer()` is called once for each session. In other words, `func` is called each time a web browser is pointed to the Shiny application.

Everything within this function is instantiated separately for each session. This includes the `input` and `output` objects that are passed to it: each session has its own `input` and `output` objects, visible within this function.

Other objects inside the function, such as variables and functions, are also instantiated for each session. In this example, each session will have its own variable named `startTime`, which records the start time for the session:

```
shinyServer(function(input, output) {  
  startTime <- Sys.time()  
  
  # ...  
})
```

## Objects visible across all sessions

You might want some objects to be visible across all sessions. For example, if you have large data structures, or if you have utility functions that are not reactive (ones that don't involve the `input` or `output` objects), then you can create these objects once and share them across all user sessions, by placing them in `server.R`, but outside of the call to `shinyServer()`.

For example:

```

# A read-only data set that will load once, when shiny starts, and will be
# available to each user session
bigDataSet <- read.csv('bigdata.csv')

# A non-reactive function that will be available to each user session
utilityFunction <- function(x) {
  # Function code here
  # ...
}

shinyServer(function(input, output) {
  # Server code here
  # ...
})

```

You could put `bigDataSet` and `utilityFunction` inside of the function passed to `shinyServer()`, but doing so will be less efficient, because they will be created each time a user connects.

If the objects change, then the changed objects will be visible in every user session. But note that you would need to use the `<<-` assignment operator to change `bigDataSet`, because the `<-` operator only assigns values in the local environment.

```

varA <- 1
varB <- 1
listA <- list(X=1, Y=2)
listB <- list(X=1, Y=2)

shinyServer(function(input, output) {
  # Create a local variable varA, which will be a copy of the shared variable
  # varA plus 1. This local copy of varA is not be visible in other sessions.
  varA <- varA + 1

  # Modify the shared variable varB. It will be visible in other sessions.
  varB <<- varB + 1

  # Makes a local copy of listA
  listASX <- 5

  # Modify the shared copy of listB
  listBSX <<- 5

  # ...
})

```

Things work this way because `server.R` is sourced when you start your Shiny app. Everything in the script is run immediately, including the call to `shinyServer()` —but the function which is passed to `shinyServer()` is called only when a web browser connects and a new session is started.

## Global objects

Objects defined in `global.R` are similar to those defined in `server.R` outside `shinyServer()`, with one important difference: they are also visible to the code in `ui.R`. This is because they are loaded into the global environment of the R session; all R code in a Shiny app is run in the global environment or a child of it.

In practice, there aren't many times where it's necessary to share variables between `server.R` and `ui.R`. The code in `ui.R` is run once, when the Shiny app is started and it generates an HTML file which is cached and sent to



each web browser that connects. This may be useful for setting some shared configuration options.

## Scope for included R files

If you want to split the server or ui code into multiple files, you can use `source(local=TRUE)` to load each file. You can think of this as putting the code in-line, so the code from the sourced files will receive the same scope as if you copied and pasted the text right there.

This example `server.R` file shows how sourced files will be scoped:

```
# Objects in this file are shared across all sessions
source('all_sessions.R', local=TRUE)

shinyServer(function(input, output) {
  # Objects in this file are defined in each session
  source('each_session.R', local=TRUE)

  output$text <- renderText({
    # Objects in this file are defined each time this function is called
    source('each_call.R', local=TRUE)

    # ...
  })
})
```

If you use the default value of `local=FALSE`, then the file will be sourced in the global environment.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

aduncan5

---

This is a very well written help file. Super clear and very crisp. Chapeau.

Dean Attali

---

This is a great technique to split up your UI as well, really useful. If anyone is running into the problem that their UI includes a TRUE/FALSE value at the end, it's because you want to use the ``value`` of the object returned from ``source``. For example, ``source("file.R", local=TRUE)$value``. See this answer on SO for more details <http://stackoverflow.com/a/305...>

ArjunaCap

to be clear, server.R is run before global.R?

tom

---

in server.R

comments powered by [Disqus](#)

## 2.41 Debugging techniques for Shiny apps

# Debugging techniques for Shiny apps

ADDED: 06 JAN 2014

## Printing

There are several techniques available for debugging Shiny applications. The first is to add calls to the `cat` function which print diagnostics where appropriate. For example, these two calls to `cat` print diagnostics to standard output and standard error respectively:

```
cat("foo\n")
cat("bar\n", file=stderr())
```

## Using browser

The second technique is to add explicit calls to the `browser` function to interrupt execution and inspect the environment where `browser` was called from. Note that using `browser` requires that you start the application from an interactive session (as opposed to using `R -e` as described above).

For example, to unconditionally stop execution at a certain point in the code:

```
# Always stop execution here
browser()
```

You can also use this technique to stop only on certain conditions. For example, to stop the MPG application only when the user selects "Transmission" as the variable:

```
# Stop execution when the user selects "am"
browser(expr = identical(input$variable, "am"))
```

## Establishing a custom error handler

You can also set the R "error" option to automatically enter the browser when an error occurs:

```
# Immediately enter the browser when an error occurs
options(error = browser)
```

Alternatively, you can specify the `recover` function as your error handler, which will print a list of the call stack and allow you to browse at any point in the stack:

```
# Call the recover function when an error occurs
options(error = recover)
```

If you want to set the error option automatically for every R session, you can do this in your `.Rprofile` file as described in [this article on R Startup](#).

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Dean Attali

---

One extremely useful debugging tool is to set ``options(shiny.error=browser)`` so that RStudio will enter its debug mode and show you where an error occurs when a Shiny app crashes.

Thanks to Joe for mentioning this, in my opinion it's the best shiny-specific debugging tool

Mithilesh Kumar

---

hi Garrett,

I am facing the same problem:

My codes are as under:

Error MESSage: ERROR: non-numeric argument to binary operator

```
ui.R
```

```
---
```

```
# shiny plots on top_docs data  
library(shiny)  
comments powered by Disqus
```

## 2.42 Learn about your user with

# Learn about your user with session\$clientData

ADDED: 06 JAN 2014

## Getting Non-Input Data From the Client

On the server side, Shiny applications use the `input` object to receive user input from the client web browser. The values in `input` are set by UI objects on the client web page. There are also non-input values (in the sense that the user doesn't enter these values through UI components) that are stored in an object called `session$clientData`. These values include the URL, the pixel ratio (for high-resolution "Retina" displays), the hidden state of output objects, and the height and width of plot outputs. You can see an example app which uses client data [here](#).

## Using session\$clientData

To access `session$clientData` values, you need to pass a function to `shinyServer()` that takes `session` as an argument (`session` is a special object that is used for finer control over a user's app session). Once it's in there, you can access `session$clientData` just as you would `input`.

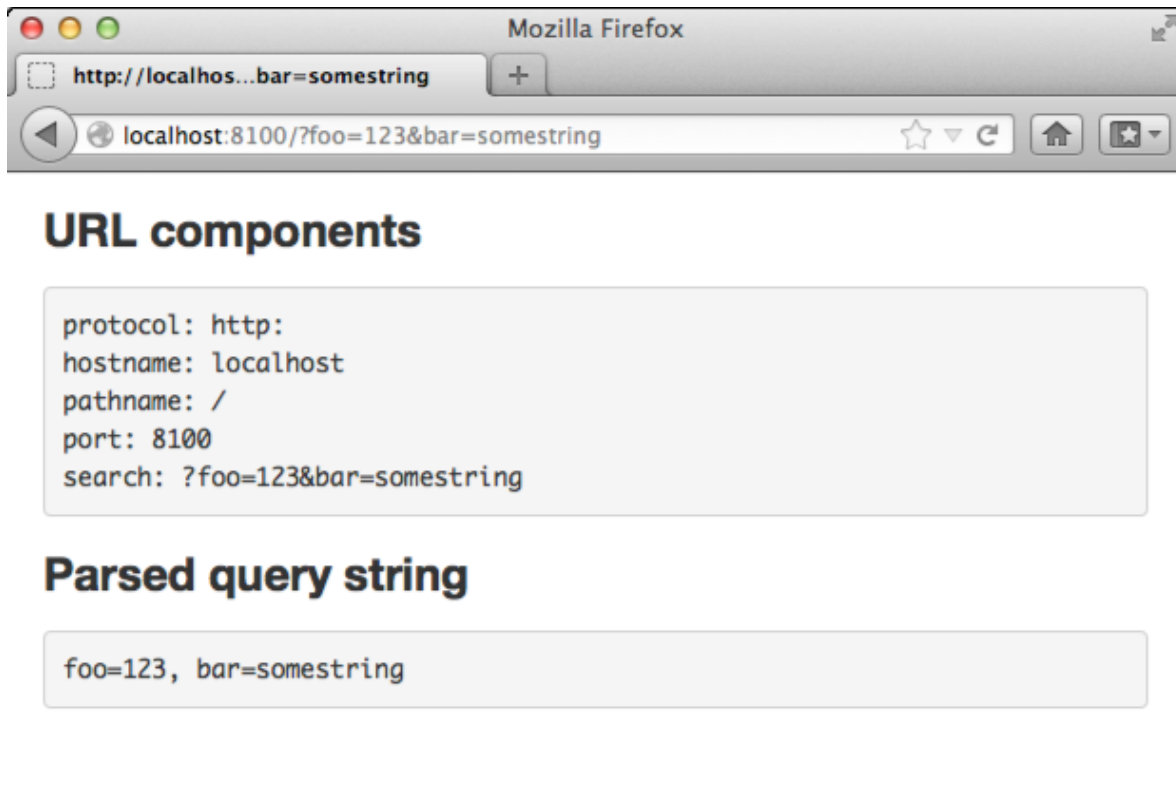
In the example below, the client browser will display out the components of the URL and also parse and print the query/search string (the part of the URL after a "?"):

server.R

```
shinyServer(function(input, output, session) {  
  
  # Return the components of the URL in a string:  
  output$urlText <- renderText({  
    paste(sep = "  
      "protocol: ", session$clientData$url_protocol, "\n",  
      "hostname: ", session$clientData$url_hostname, "\n",  
      "pathname: ", session$clientData$url_pathname, "\n",  
      "port: ", session$clientData$url_port, "\n",  
      "search: ", session$clientData$url_search, "\n"  
    )  
  })  
  
  # Parse the GET query string  
  output$queryText <- renderText({  
    query <- parseQueryString(session$clientData$url_search)  
  
    # Return a string with key-value pairs  
    paste(names(query), query, sep = "=", collapse=" ")  
  })  
})
```

```
shinyUI(bootstrapPage(  
  h3("URL components"),  
  verbatimTextOutput("urlText"),  
  
  h3("Parsed query string"),  
  verbatimTextOutput("queryText")  
))
```

This app will display the following:



## Viewing all available values in clientData

The values in `session$clientData` will depend to some extent on the outputs. For example, a plot output object will report its height, width, and hidden status. The app below has a plot output, and displays all the values in `session$clientData`:

```
shinyServer(function(input, output, session) {  
  # Store in a convenience variable  
  cdata <- session$clientData  
  
  # Values from cdata returned as text  
  output$clientdataText <- renderText({  
    cnames <- names(cdata)  
  
    allvalues <- lapply(cnames, function(name) {  
      paste(name, cdata[[name]], sep=" = ")  
    })  
    paste(allvalues, collapse = "\n")  
  })  
})
```

```

# A histogram
output$myplot <- renderPlot({
  hist(rnorm(input$obs), main="Generated in renderPlot()")
})
})

```

Notice that, just as with `input`, values in `session$clientData` can be accessed with `session$clientData$myvar` or `session$clientData[['myvar']]`. Or, equivalently, since we've saved it into a convenience variable `cdata`, we can use `cdata$myvar` or `cdata[['myvar']]`.

ui.R

```

shinyUI(pageWithSidebar(
  headerPanel("Shiny Client Data"),
  sidebarPanel(
    sliderInput("obs", "Number of observations:",
              min = 0, max = 1000, value = 500)
  ),
  mainPanel(
    h3("clientData values"),
    verbatimTextOutput("clientdataText"),
    plotOutput("myplot")
  )
))

```

For the plot output `output$myplot`, there are three entries in `clientData`:

- `output_myplot_height`: The height of the plot on the web page, in pixels.
- `output_myplot_width`: The width of the plot on the web page, in pixels.
- `output_myplot_hidden`: If the object is hidden (not visible), this is TRUE. This is used because Shiny will by default suspend the output object when it is hidden. When suspended, the observer will not execute even when its inputs change.

Here is the view from the client, with all the `clientData` values:

Number of observations:

0 500 1,000

### clientData values

```
allowDataUriScheme = TRUE
output_clientdataText_hidden = FALSE
output_myplot_height = 250
output_myplot_hidden = FALSE
output_myplot_width = 488
pixelratio = 1
url_hostname = localhost
url_pathname = /
url_port = 8100
url_protocol = http:
url_search = ?foo=123&bar=somestring
```

Generated in renderPlot()

Bin Range	Frequency
-3.0 to -2.5	5
-2.5 to -2.0	10
-2.0 to -1.5	10
-1.5 to -1.0	55
-1.0 to -0.5	85
-0.5 to 0.0	100
0.0 to 0.5	90
0.5 to 1.0	80
1.0 to 1.5	45
1.5 to 2.0	25
2.0 to 2.5	10
2.5 to 3.0	5

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Leighton

Hi.



I've been trying to obtain the url search field (which changes according to user input) with `session$clientData$url_search`. However the returned result is always empty. I suspect that it's because the webpage starts off with an empty `url_search`. I have tried `reactiveEvent` and `observe` but neither works in retrieving it after I make my input. Any ideas?  
Thank you!

Abhijit Sahay

---

Hi Garrett:

I changed the original example slightly -- adding a `selectInput` control, whose choices and label I would like to update based on the query part of the URL. The choices change successfully but the label does not?

Thank you for any help,  
Abhijit

-----  
[comments powered by Disqus](#)

## 2.43 Unicode characters in Shiny apps

# Unicode characters in Shiny apps

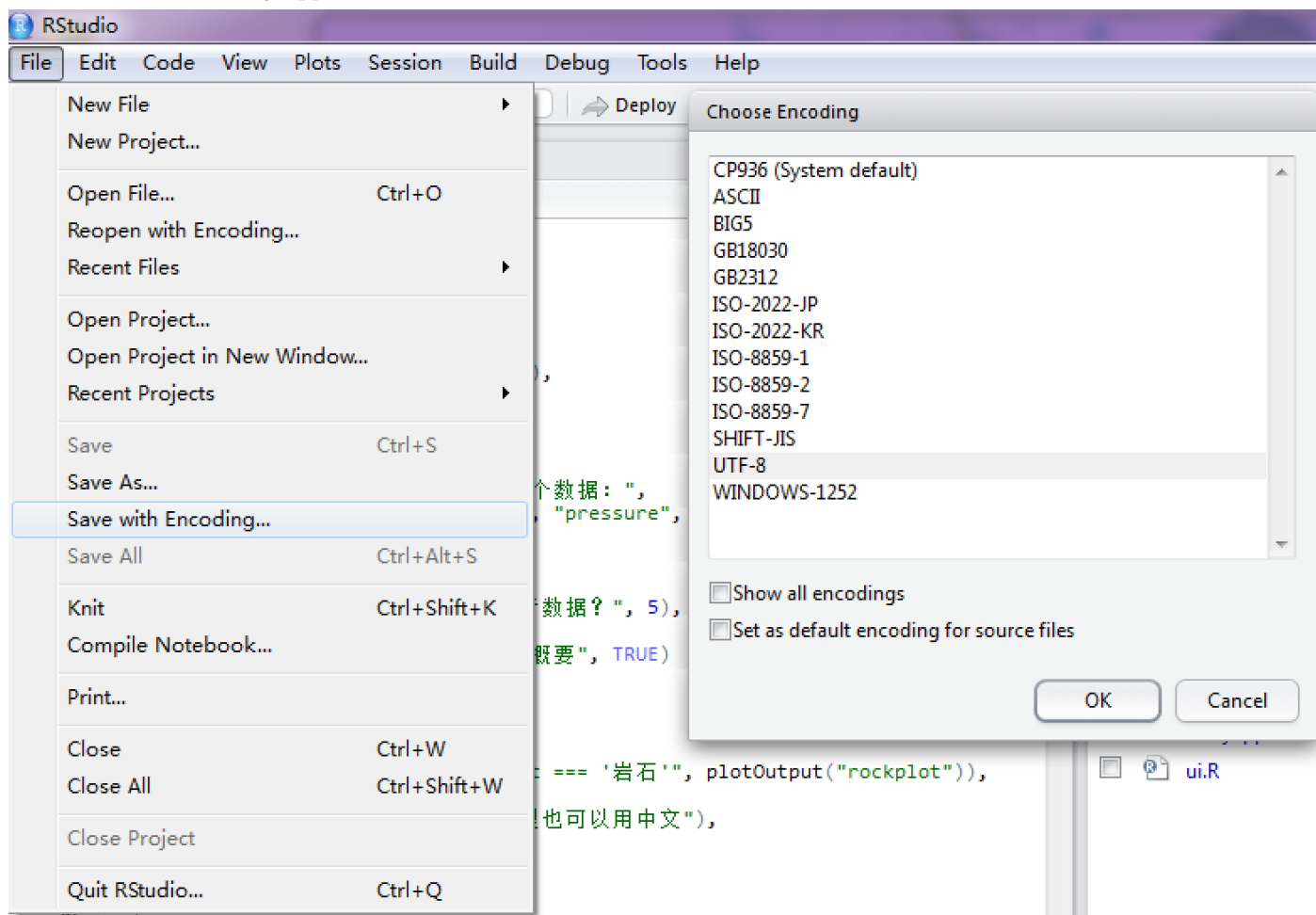
ADDED: 09 JUL 2014

BY: YIHUI XIE

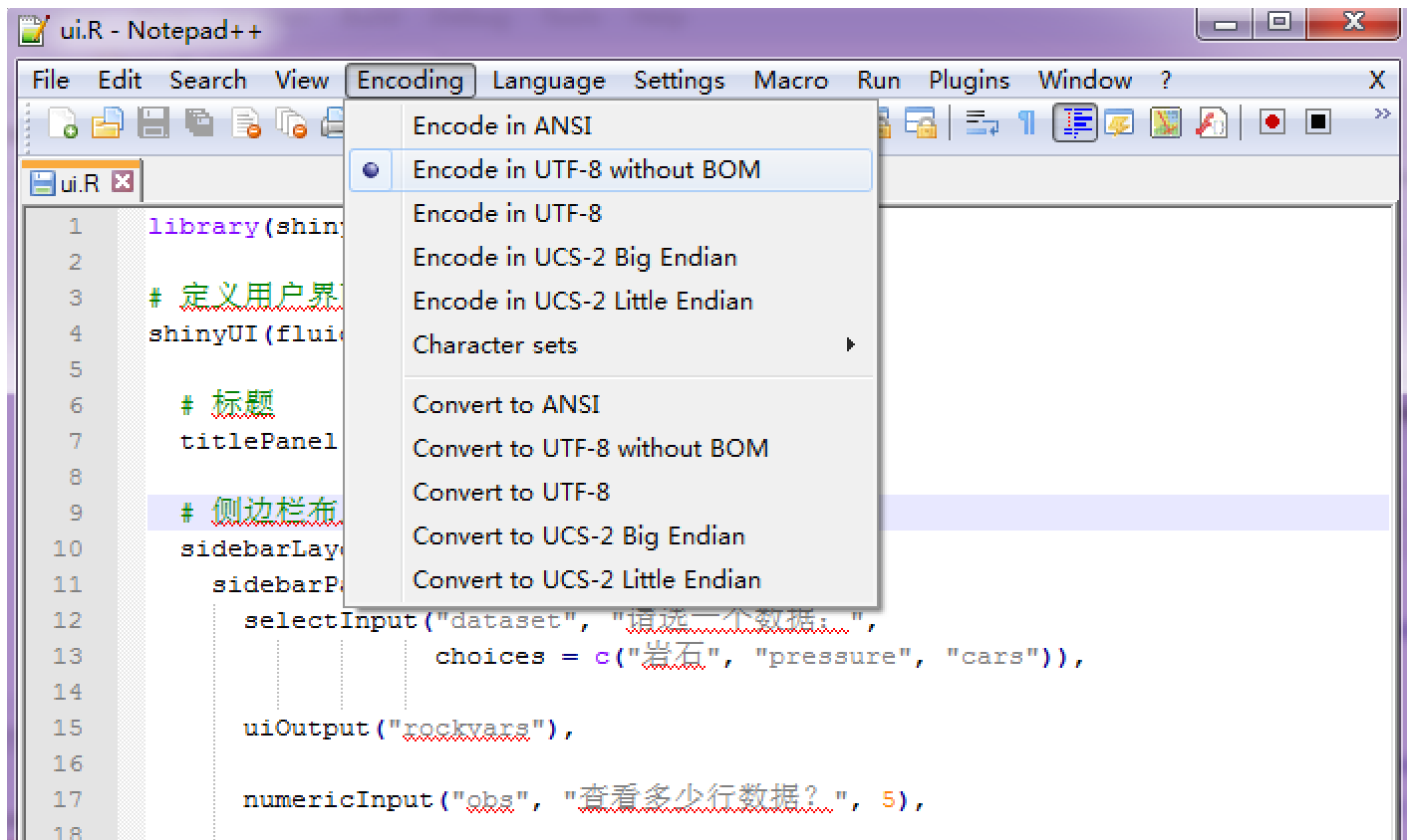
Since Shiny v0.10.1, we have added support for multi-byte characters in Shiny apps on Windows. Linux and Mac OS X users normally do not need to worry about character encodings or non-ASCII characters, and they can basically ignore this article, since their system locale is often UTF-8 based. However, Windows does not have a single consistent locale or native character encoding, which makes it tricky to support multi-byte characters there. For the sake of consistency and portability, Shiny requires the character encoding of all its components to be UTF-8, which include `ui.R`, `server.R`, `global.R`, `DESCRIPTION`, and/or `README.md`. Note a Shiny app may not contain all of these files, but all of them must be encoded in UTF-8 if they exist.

## Text Editors

A modern text editor should allow you to save a text file with a specified encoding. For example, if you use the RStudio IDE, you can click the menu `File -> Save with Encoding` to (re)save a file with the UTF-8 encoding:



If you do not use RStudio, there is one more thing to keep in mind: when you save a file with UTF-8, you should make sure *not* to include the **byte order mark** (BOM). Some text editors do include it by default, such as Notepad (the default text editor on Windows). Shiny will try to detect the BOM character, and give a warning if it exists. For a file that is encoded in UTF-8 with BOM, you can open it with the UTF-8 encoding in RStudio, re-save it with the UTF-8 encoding, and the BOM will be removed. There are many other text editors that support UTF-8 with or without BOM, such as Notepad++:



## An Example (Chinese Characters)

There is [an example](#) in the gallery demonstrating Simplified Chinese characters in a Shiny app, in which we used Chinese characters in many places, such as R comments, the title of the page, the label and choices of the select input, the JavaScript condition of the conditional panel, the id of the verbatim text output, the R formula, and so on.

## File Input/Output

When your Shiny app involves file input/output, the character encoding does not have to be UTF-8. Although we recommend UTF-8 in Shiny, it is not the default encoding on Windows anyway, so your app users may have trouble especially when they have file interactions with your app.

Many I/O functions in R have an argument named `encoding` (sometimes `fileEncoding`). If the data to be read or written is not encoded with the native encoding of your system, you may have to use the `encoding` argument. For example, when reading a text file encoded in UTF-8 into a Shiny app, you may use

```
readLines('foo.txt', encoding = 'UTF-8')
```

Similarly, when writing a CSV file with the GB2312 encoding (a commonly used encoding for Simplified Chinese), you can use `write.csv(data, fileEncoding = 'GB2312')`. This is very important when using the `fileInput()` or `downloadHandler()` functions in the shiny package.

If you read a file into R as a character vector, and the file is not encoded with your system's native encoding, you are recommended to convert the encoding of the character vector to your system's native encoding before you process the text data. Some character string processing functions such as `gsub(..., fixed = TRUE)` may not work if the string does not have the native encoding.

```
x <- readLines('foo.txt', encoding = 'UTF-8')
x <- enc2native(x)
gsub(' ', '-', x, fixed = TRUE)
```

## The Global `encoding` Option

The function `options()` in base R can be used to set some global options for the current R session, among which there is an `encoding` option. Its default value is `native.enc` (native encoding), which is not really a standard encoding name, and its meaning differs on different platforms. On Linux and Mac OS X, the native encoding is often UTF-8. If you are not sure what your native encoding is, the function `localeToCharset()` in base R should give a reasonable *guess* in most cases.

When dealing with encoding problems, you are not recommended to set the `encoding` option to a specific encoding name, e.g. `options(encoding = 'UTF-8')`. This may have very bad consequences, since it makes a strong assumption that all file connections and character manipulations should use this encoding by default.

## Shinyapps.io

For shinyapps.io users, the platform is based on Linux containers, and has a UTF-8 locale. If your app reads/writes data files that contain multi-byte characters, you are strongly recommended to be specific about the encodings when calling the I/O functions, because your local environment may not be based on UTF-8. The functions `iconv()`, `iconvlist()`, `enc2native()`, and `enc2utf8()` may be useful if you need to convert the encoding from one to another.

## Summary

To sum up, three things to keep in mind when dealing with character strings in R:

1. The encoding should be specified explicitly per (file) connection basis, if you want your R code to be portable;
2. After you read Unicode characters into R, convert them to the native encoding of your system, e.g. using `enc2native()` ;
3. Do not set `options(encoding)` .

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

milkfan

---

With regard to Chinese Characters. I am able to run the shiny app locally but once I run the `deployApp()` command to deploy to [shinyapps.io](#), it starts to complain about "invalid multibyte string." Any suggestions?

milkfan

---

I solved this by running `deployApp(lint=F)`

Xiushi Le

---

Didn't work as intended. I was trying the following code.

```
i <- 1, j <- 1
val <- 'val'
a <- 'æµè•'
actionButton(paste0(l[i], '_val_', j), label=a, value=val)
<button id="B_val_1" type="button" class="btn action-button"
value="BO"><u+6d4b><u+8 warning="" message="" in="" readlines(conn)="":=""
incomplete="" final="" line="" found="" on="" "=">
comments powered by Disqus
```

## 2.44 Style your apps with CSS

# Style your apps with CSS

ADDED: 06 MAY 2014

BY: GARRETT GROLEMUND WITH JOE CHENG

You can give your Shiny app a special look with cascading style sheets (CSS).

CSS is a style language which gives HTML documents a sophisticated look. Since the Shiny app user-interface (UI) is an HTML document, you can use CSS to control how you want your Shiny app to look.

The CSS language is easy to learn and widely used. If you would like to learn CSS code, I recommend the free interactive [codecademy tutorial](#). There is also a free concise tutorial at [udemy.com](#).

In this article, I describe the three ways Shiny works with CSS. To get CSS into your Shiny App, you can:

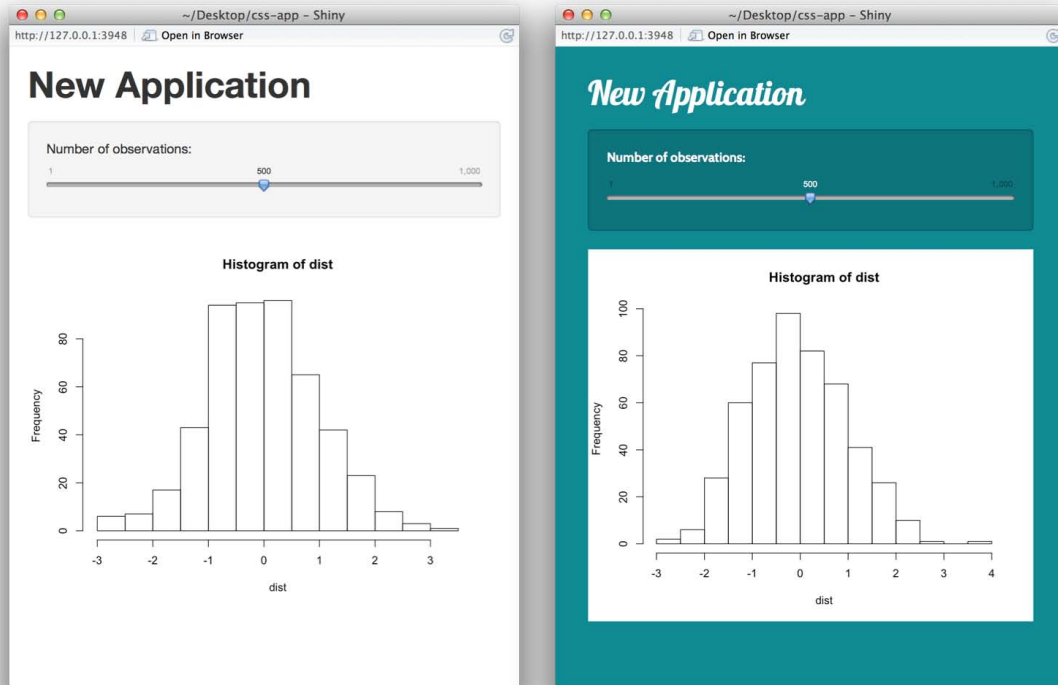
1. Add style sheets with the `www` directory
2. Add CSS to your HTML header
3. Add styling directly to HTML tags

These methods correspond to the three ways that you can add CSS to an HTML document. In HTML, you can:

1. Link to a stylesheet file
2. Write CSS in the document's header, and
3. Write CSS in the style attribute of an HTML element.

Recall that the C in CSS refers to “cascading”. CSS in a style attribute will overrule CSS in a document's header, which will overrule CSS in an external file. This cascading arrangement lets you create global style rules (in an external file, or header), and special cases (elements that have their own style attribute).

## 1. Add style sheets with the `www` directory



These two 'New Application' Shiny apps are identical, except for the appearance of the UI. The basic one on the left is the default Shiny App. The one on the right uses a CSS file to enhance its look.

You can apply a CSS file to your entire Shiny app by linking to it from within your Shiny App.

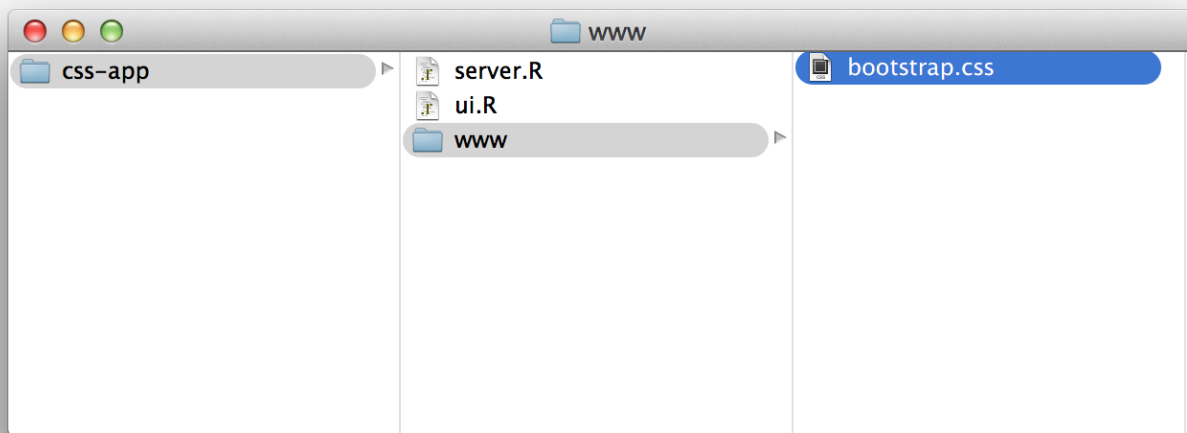
- Create a subdirectory named `www` in your Shiny app directory. This subdirectory name `www` is special. Shiny makes every file in `www` available to your user's browser. The `www` subdirectory is a great place to put CSS files, images, and other things a browser needs to build your Shiny App.
- Place your CSS file in the `www` subdirectory.

For this UI change, I am using a CSS file named `bootstrap.css`. I downloaded it from [Bootstrap](#), a great place to get free CSS themes for bootstrap webpages.

*Note: The Shiny UI is built with the Bootstrap 3.3.1 HTML/CSS framework. CSS files designed to work with Bootstrap 3.3.1 will work best with Shiny.*

After you get `bootstrap.css` and put it in your `www` subdirectory, your Shiny app directory should look like mine:





Once your Shiny app directory is set, you have two choices to link to your CSS file (your app won't use the CSS until you do). You can:

1. set the theme argument of `fluidPage` to your document's name or
2. include the file with the `tags` object.

## The theme argument

The simplest choice is to set the theme argument of `fluidPage` to your document's name (as a character string). Your `ui.R` script will look like the following code:

```
shinyUI(fluidPage(theme = "bootstrap.css",
  headerPanel("New Application"),
  sidebarPanel(
    sliderInput("obs", "Number of observations:",
      min = 1, max = 1000, value = 500)
  ),
  mainPanel(plotOutput("distPlot"))
))
```

I used this code to make the 'New Application' Shiny app on the right. To make the default Shiny app (the one on the left), remove `theme = "bootstrap.css"`.

## Link to a stylesheet with `tags`

You can link to the CSS file with the `tags` object too. `tags` recreates popular HTML tags, and has its own article [here](#).

The standard way to link to a CSS file in an HTML document is with a link tag embedded inside a head tag. For example, you might write an HTML document like the one below:

```
<!DOCTYPE html >
<html >
  <head>
```

```

<link type="text/css" rel="stylesheet" href="bootstrap.css"/>
</head>
<body>
</body>
</html>

```

You can recreate this arrangement in Shiny with:

```

shinyUI(fluidPage(

  tags$head(
    tags$link(rel = "stylesheet", type = "text/css", href = "bootstrap.css")
  ),

  headerPanel("New Application"),

  sidebarPanel(
    sliderInput("obs", "Number of observations:",
               min = 1, max = 1000, value = 500)
  ),

  mainPanel(plotOutput("distPlot"))
))

```

This code will make the ‘New Application’ Shiny app on the right.

Notice that `bootstrap.css` is stored in the `www` subdirectory. You do not need to add `www/` to the file path that you give `href`. Shiny places the files in your Shiny App’s home directory before sending them to your user’s browser.

Remember that I said `www` is a special subdirectory name and important to use. Shiny will share a file with your user’s browser if the file appears in `www`. Shiny will not share files that you do not place in `www`.

Both the theme argument of `fluidPage` and the `href` argument of `tags$link` can point to a URL (that contains a CSS file). You may find this method a convenient way to share the same CSS file across many Shiny apps.

## 2. Add CSS to your HTML header

You can also add CSS directly to your Shiny UI. This is the equivalent of adding CSS to the head tag of an HTML document. This CSS will override any CSS imported from an external file (should a conflict arise).

As before, you have two options for adding CSS at this level. You can

1. Add CSS with `tags`, or
2. Include a whole file of CSS with `includeCSS`

### Add CSS to the header with `tags`

Use `tags$style` instead of `tags$link` to include raw CSS. Write the CSS as a character string wrapped in `HTML()` to prevent Shiny from escaping out HTML specific characters.

As with `tags$link`, nest `tags$style` inside of `tags$head`.

```

shinyUI(fluidPage(

  tags$head(
    tags$style(HTML("

```

```

@import url('//fonts.googleapis.com/css?family=Lobster|Cabin:400,700');

h1 {
  font-family: 'Lobster', cursive;
  font-weight: 500;
  line-height: 1.1;
  color: #48ca3b;
}

"))
),

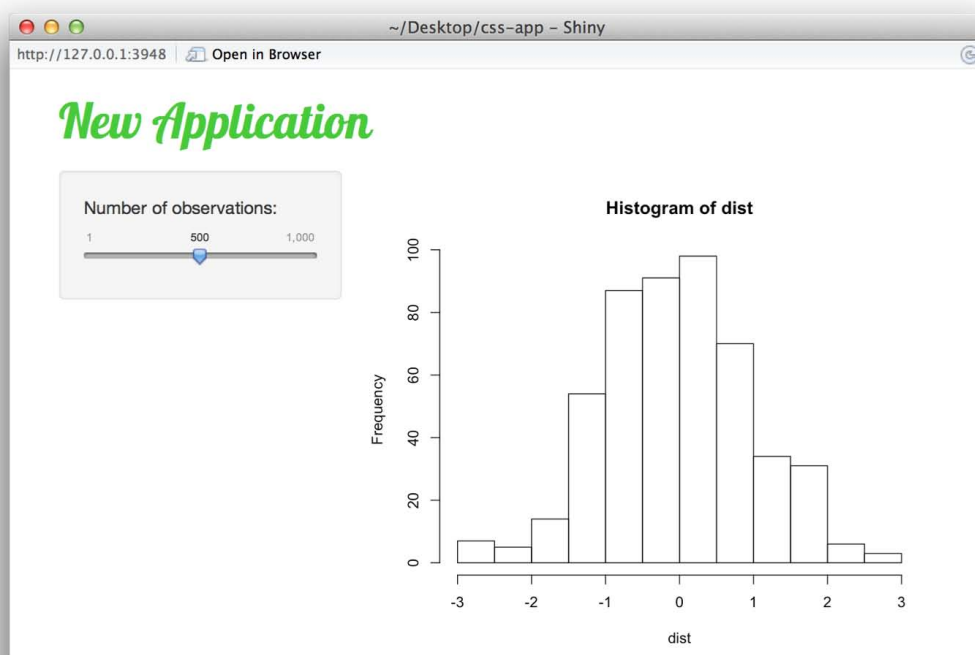
headerPanel("New Application"),

sidebarPanel(
  sliderInput("obs", "Number of observations:",
             min = 1, max = 1000, value = 500)
),

mainPanel(plotOutput("distPlot"))
))

```

This code makes the following Shiny App. Note that I changed only the title's appearance.

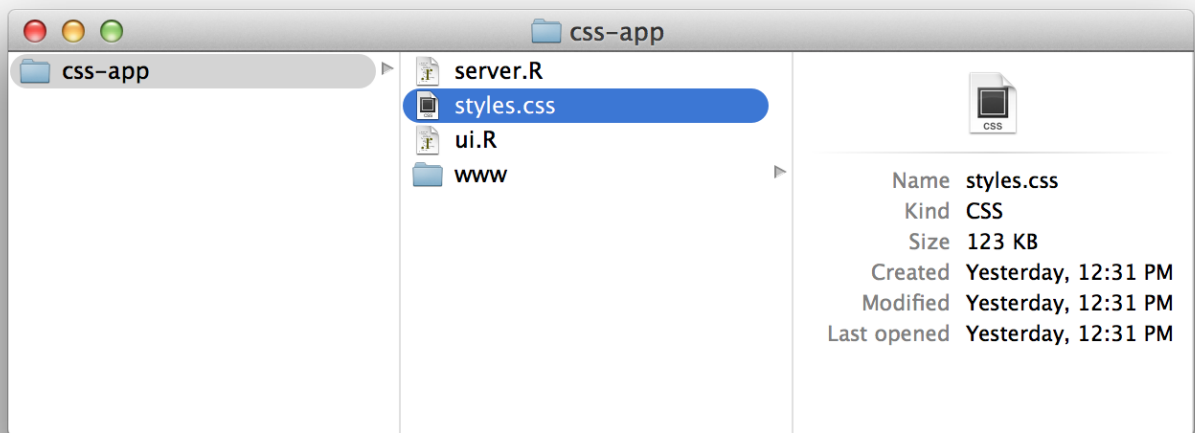


## Add CSS to the header with `includeCSS`

If you have CSS saved in a file, you can add it to the header of your Shiny app with `includeCSS`.

Shiny will place the contents of the file into the header of the HTML document it gives to web browsers, as if you had used `tags$style`. You do not need to save the file in `www`. Shiny will expect it in your Shiny app directory unless you specify otherwise.

For example, I've placed a lightweight style sheet named `styles.css` in my app directory below.



This CSS file changes the title of a Shiny app and nothing else. Here is the actual CSS saved in the file.

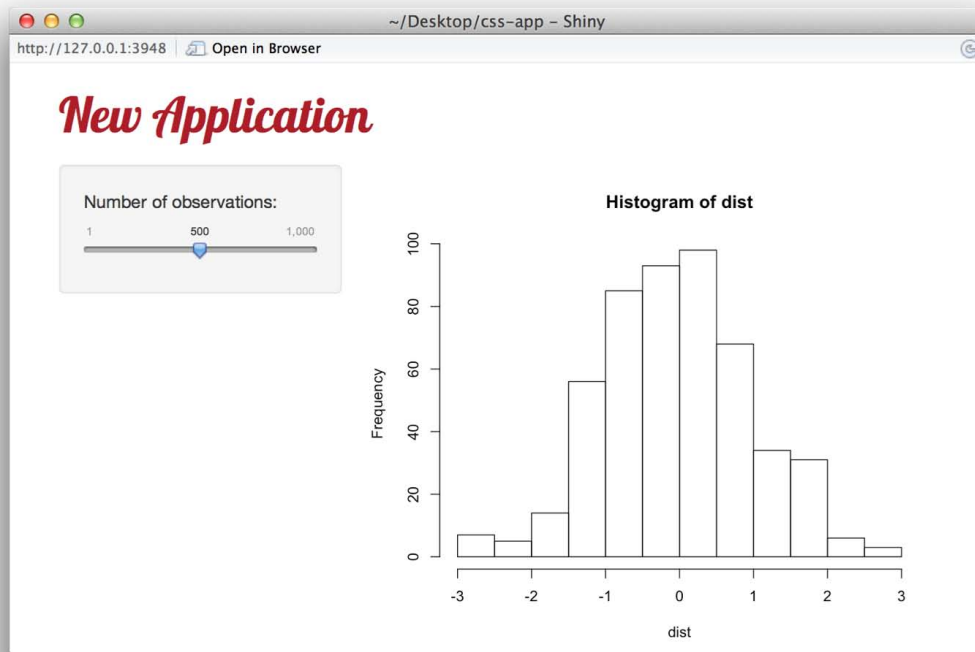
```
@import url ("//fonts.googleapis.com/css?family=Lobster|Cabin:400,700");

h1 {
  font-family: 'Lobster', cursive;
  font-weight: 500;
  line-height: 1.1;
  color: #ad1d28;
}

body {
  background-color: #fff;
}
```

The code below includes `styles.css` to make the app in the next image.

```
shinyUI(fluidPage(
  includeCSS("styles.css"),
  headerPanel("New Application"),
  sidebarPanel(
    sliderInput("obs", "Number of observations:",
               min = 1, max = 1000, value = 500)
  ),
  mainPanel(plotOutput("distPlot"))
))
```



### 3. Add styling directly to HTML tags

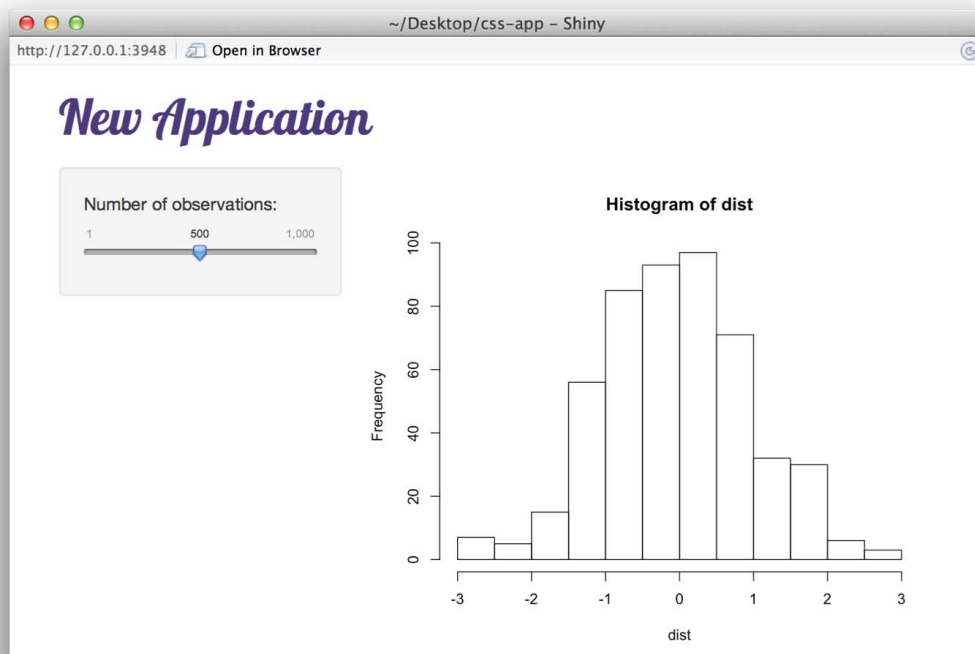
You can add CSS styling directly to individual HTML elements in your UI, just as you add styling directly to HTML tags in a web document. CSS provided at this level takes precedence over any other sources of CSS (should a conflict occur).

To add CSS to an individual element, pass it to the `style` argument of the Shiny function that you use to create that element.

In the script below, I set the style of the title of the Shiny app with the `style` argument of `h1` in `headerPanel`. The style relies on a font that I import with `tags$style` in `tags$head`.

```
shinyUI(fluidPage(
  tags$head(
    tags$style(HTML("
      @import url('//fonts.googleapis.com/css?family=Lobster|Cabin:400,700');
    ")),
  ),
  headerPanel(
    h1("New Application",
      style = "font-family: 'Lobster', cursive;
        font-weight: 500; line-height: 1.1;
        color: #4d3a7d;"),
  ),
  sidebarPanel(
    sliderInput("obs", "Number of observations:",
      min = 1, max = 1000, value = 500)
  ),
  mainPanel(plotOutput("distPlot"))
))
```

Here's what the Shiny app looks like with this code:



## Recap

Add CSS to a Shiny UI just as you would add CSS to a web page.

To get CSS into your Shiny App, you can:

1. Link to an external CSS file
2. Include CSS in the header of the web page that the app is built on, or
3. Pass style information to individual HTML elements in your app.

My examples give a glimpse into the options CSS offers your Shiny App. Explore more at [Bootswatch](#) and see what you can create.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

- [Firefox](#)
- [Chrome](#)
- [Internet Explorer 10+](#)
- [Safari](#)

In the google groups theres a discussion about navbarPages and how some of the methods in this article can cause a ghost tab to appear. If you take the title out of the navbarPage command than no ghost tab appears. Seems to be a bug. Is there a better place to call this out?

Laurent Franckx

---

Nice, but do you have any illustration of how you can for instance change the width of an textInput field?

Aurora Data

---

It looks like changing the title font no longer working. I was able to get it to work last week. Did somethings got changed?

Mona Jalal

---

comments powered by [Disqus](#)

## 2.45 Build custom input objects

# Build custom input objects

ADDED: 06 JAN 2014

## Building Inputs

Shiny comes equipped with a variety of useful input components, but as you build more ambitious applications, you may find yourself needing input widgets that we don't include. Fortunately, Shiny is designed to let you create your own custom input components. If you can implement it using HTML, CSS, and JavaScript, you can use it as a Shiny input! See the demo apps with custom inputs [here](#) and [here](#).

(If you're only familiar with R and not with HTML/CSS/JavaScript, then you will likely find it tough to create all but the simplest custom input components on your own. However, other people can – and hopefully will – bundle up their custom Shiny input components as R packages and make them available to the rest of the community.)

## Design the Component

The first steps in creating a custom input component is no different than in any other form of web development. You write HTML markup that lays out the component, CSS rules to style it, and use JavaScript (mostly event handlers) to give it behavior, if necessary.

Shiny input components should try to adhere to the following principles, if possible:

- Designed to be used from HTML and R: Shiny user interfaces can either be written using R code (that generates HTML), or by writing the HTML directly. A well-designed Shiny input component will take both styles into account: offer an R function for creating the component, but also have thoughtfully designed and documented HTML markup.
- Configurable using HTML attributes: Avoid requiring the user to make JavaScript calls to configure the component. Instead, it's better to use HTML attributes. In your component's JavaScript logic, you can [easily access these values using jQuery](#) (or simply by reading the DOM attribute directly).

When used in a Shiny application, your component's HTML markup will be repeated once for each instance of the component on the page, but the CSS and JavaScript will generally only need to appear once, most likely in the `<head>`. For R-based interface code, you can use the functions `singleton` and `tags$head` together to ensure these tags appear once and only once, in the head. (See the full example below.)

## Write an Input Binding

Each custom input component also needs an *input binding*, an object you create that tells Shiny how to identify instances of your component and how to interact with them. (Note that each *instance* of the input component doesn't need its own input binding object; rather, all instances of a particular type of input component share a single input binding object.)

An input binding object needs to have the following methods:

`find(scope)`

Given an HTML document or element (`scope`), find any descendant elements that are an instance of your



component and return them as an array (or array-like object). The other input binding methods all take an `el` argument; that value will always be an element that was returned from `find`.

A very common implementation is to use jQuery's `find` method to identify elements with a specific class, for example:

```
exampleInputBinding.find = function(scope) {
  return $(scope).find(".exampleComponentClass");
};
```

#### `getId(el)`

Return the Shiny input ID for the element `el`, or `null` if the element doesn't have an ID and should therefore be ignored. The default implementation in `Shiny.InputBinding` reads the `data-input-id` attribute and falls back to the element's `id` if not present.

#### `getValue(el)`

Return the Shiny value for the element `el`. This can be any JSON-compatible value.

#### `setValue(el, value)`

Set the element to the specified value. (This is not currently used, but in the future we anticipate adding features that will require the server to push input values to the client.)

#### `subscribe(el, callback)`

Subscribe to DOM events on the element `el` that indicate the value has changed. When the DOM events fire, call `callback` (a function) which will tell Shiny to retrieve the value.

We recommend using jQuery's event namespacing feature when subscribing, as unsubscribing becomes very easy (see `unsubscribe`, below). In this example, `exampleComponentName` is used as a namespace:

```
exampleInputBinding.subscribe = function(el, callback) {
  $(el).on("keyup.exampleComponentName", function(event) {
    callback(true);
  });
  $(el).on("change.exampleComponentName", function(event) {
    callback();
  });
};
```

Later on, we can unsubscribe `".exampleComponentName"` which will remove all of our handlers without touching anyone else's.

The `callback` function optionally takes an argument: a boolean value that indicates whether the component's rate policy should apply (`true` means the rate policy should apply). See `getRatePolicy` below for more details.

#### `unsubscribe(el)`

Unsubscribe DOM event listeners that were bound in `subscribe`.

Example:

```
exampleInputBinding.unsubscribe = function(el) {
  $(el).off(".exampleComponentName");
};
```

#### `getRatePolicy()`

Return an object that describes the rate policy of this component (or `null` for default).

Rate policies are helpful for slowing down the rate at which input events get sent to the server. For example, as the user drags a slider from value A to value B, dozens of change events may occur. It would be wasteful to send all of those events to the server, where each event would potentially cause expensive computations to occur.

A rate policy slows down the rate of events using one of two algorithms (so far). Throttling means no more than one event will be sent per X milliseconds. Debouncing means all of the events will be ignored until no events have been received for X milliseconds, at which time the most recent event will be sent. [This blog post](#) goes into more detail about the difference between throttle and debounce.

A rate policy object has two members:

- `policy` - Valid values are the strings `"direct"`, `"debounce"`, and `"throttle"`. `"direct"` means that all events are sent immediately.
- `delay` - Number indicating the number of milliseconds that should be used when debouncing or throttling. Has no effect if the policy is `direct`.

Rate policies are only applied when the `callback` function in `subscribe` is called with `true` as the first parameter. It's important that input components be able to control which events are rate-limited and which are not, as different events may have different expectations to the user. For example, for a textbox, it would make sense to rate-limit events while the user is typing, but if the user hits Enter or focus leaves the textbox, then the input should always be sent immediately.

## Register Input Binding

Once you've created an input binding object, you need to tell Shiny to use it:

```
Shiny.setInputBindings.register(exampleInputBinding, "yourname.exampleInputBinding");
```

The second argument is a name the user can use to change the priority of the binding. On the off chance that the user has multiple bindings that all want to claim the same HTML element as their own, this call can be used to control the priority of the bindings:

```
Shiny.setInputBindings.setPriority("yourname.exampleInputBinding", 10);
```

Higher numbers indicate a higher priority; the default priority is 0. All of Shiny's built-in input component bindings default to a priority of 0.

If two bindings have the same priority value, then the more recently registered binding has the higher priority.

## Example

For this example, we'll create a button that displays a number, whose value increases by one each time the button is clicked. Here's what the end result will look like:

0

To start, let's design the HTML markup for this component:

```
<button id="inputId" class="increment btn btn-default" type="button">0</button>
```

The CSS class `increment` is what will differentiate our buttons from any other kind of buttons. (The `btn btn-default` classes are there to make the button look decent in [Bootstrap](#).)

Now we'll write the JavaScript that drives the button's basic behavior:

```
$(document).on("click", "button.increment", function(evt) {
  // evt.target is the button that was clicked
  var el = $(evt.target);
```

```
// Set the button's text to its current value plus 1
el.text(parseInt(el.text()) + 1);

// Raise an event to signal that the value changed
el.trigger("change");
});
```

This code uses jQuery's [delegated events feature](#) to bind all increment buttons at once.

Now we'll create the Shiny binding object for our component, and register it:

```
var incrementBinding = new Shiny.InputBinding();
$.extend(incrementBinding, {
  find: function(scope) {
    return $(scope).find(".increment");
  },
  getValue: function(el) {
    return parseInt($(el).text());
  },
  setValue: function(el, value) {
    $(el).text(value);
  },
  subscribe: function(el, callback) {
    $(el).on("change.increment", function(e) {
      callback();
    });
  },
  unsubscribe: function(el) {
    $(el).off(".increment");
  }
});
```

```
Shiny.inputBindings.register(incrementBinding);
```

Both the behavioral JavaScript code and the Shiny binding code should generally be run when the page loads. (It's important that they run before Shiny initialization, which occurs after all the document ready event handlers are executed.)

The cleanest way to do this is to put both chunks of JavaScript into a file. In this case, we'll use the path `./www/js/increment.js`, which we can then access as `http://localhost:8100/js/increment.js`.

If you're using an `index.html` style user interface, you'll just need to add this line to your `<head>` (make sure it comes after the script tag that loads `shiny.js`):

```
<script src="js/increment.js"></script>
```

On the other hand, if you're using `ui.R`, then you can define this function before the call to `shinyUI`:

```
incrementButton <- function(inputId, value = 0) {
  tagList(
    singleton(tags$head(tags$script(src = "js/increment.js"))),
    tags$button(id = inputId,
      class = "increment btn btn-default",
      type = "button",
      as.character(value))
  )
}
```

Then in your `shinyUI` page definition you can call `incrementButton` wherever you want an increment button rendered. Notice the line that begins with `singleton` will ensure that the `increment.js` file will be included just one time, in the `<head>`, no matter how many buttons you insert into the page or where you place them.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Dean Attali

---

For others: based on looking at the source (<https://github.com/rstudio/shi...>) it looks like you can also add a ``initialize`` method that will get called when the input is being initialized, which can be useful if you need to perform some custom initialization. I'm not sure if we're meant to do this or not because I didn't see it in the docs, I just wanted to share that I saw it in source code

comments powered by Disqus

## 2.46 Build custom output objects

# Build custom output objects

ADDED: 06 JAN 2014

Right out of the box, Shiny makes it easy to include plots, simple tables, and text as outputs in your application; but we imagine that you'll also want to display outputs that don't fit into those categories. Perhaps you need an interactive [choropleth map](#) or a [googleVis motion chart](#).

Similar to [custom inputs](#), if you have some knowledge of HTML/CSS/JavaScript you can also build reusable, custom output components. And you can bundle up output components as R packages for other Shiny users to use.

## Server-Side Output Functions

Start by deciding the kind of values your output component is going to receive from the user's server side R code.

Whatever value the user's R code returns is going to need to somehow be turned into a JSON-compatible value (Shiny uses [RJSONIO](#) to do the conversion). If the user's code is naturally going to return something RJSONIO-compatible – like a character vector, a data frame, or even a list that contains atomic vectors – then you can just direct the user to use a function on the server. However, if the output needs to undergo some other kind of transformation, then you'll need to write a wrapper function that your users will use instead (analogous to `renderPlot` or `renderTable`).

For example, if the user wants to output [time series objects](#) then you might create a `renderTimeSeries` function that knows how to translate `ts` objects to a simple list or data frame:

```
renderTimeSeries <- function(expr, env=parent.frame(), quoted=FALSE) {  
  # Convert the expression + environment into a function  
  func <- exprToFunction(expr, env, quoted)  
  
  function() {  
    val <- func()  
    list(start = tsp(val)[1],  
         end = tsp(val)[2],  
         freq = tsp(val)[3],  
         data = as.vector(val))  
  }  
}
```

which would then be used by the user like so:

```
output$timeSeries1 <- renderTimeSeries({  
  ts(matrix(rnorm(300), 100, 3), start=c(1961, 1), frequency=12)  
})
```

## Design Output Component Markup

At this point, we're ready to design the HTML markup and write the JavaScript code for our output component.

For many components, you'll be able to have extremely simple HTML markup, something like this:

```
<div id="timeSeries1" class="timeseries-output"></div>
```

We'll use the `timeseries-output` CSS class as an indicator that the element is one that we should bind to. When new output values for `timeSeries1` come down from the server, we'll fill up the div with our visualization using JavaScript.

## Write an Output Binding

Each custom output component needs an *output binding*, an object you create that tells Shiny how to identify instances of your component and how to interact with them. (Note that each *instance* of the output component doesn't need its own output binding object; rather, all instances of a particular type of output component share a single output binding object.)

An output binding object needs to have the following methods:

`find(scope)`

Given an HTML document or element (`scope`), find any descendant elements that are an instance of your component and return them as an array (or array-like object). The other output binding methods all take an `el` argument; that value will always be an element that was returned from `find`.

A very common implementation is to use jQuery's `find` method to identify elements with a specific class, for example:

```
exampleOutputBinding.find = function(scope) {
  return $(scope).find(".exampleComponentClass");
};
```

`getId(el)`

Return the Shiny output ID for the element `el`, or `null` if the element doesn't have an ID and should therefore be ignored. The default implementation in `Shiny.OutputBinding` reads the `data-output-id` attribute and falls back to the element's `id` if not present.

`renderValue(el, data)`

Called when a new value that matches this element's ID is received from the server. The function should render the data on the element. The type/shape of the `data` argument depends on the server logic that generated it; whatever value is returned from the R code is converted to JSON using the RJSONIO package.

`renderError(el, err)`

Called when the server attempts to update the output value for this element, and an error occurs. The function should render the error on the element. `err` is an object with a `message` String property.

`clearError(el)`

If the element `el` is currently displaying an error, clear it.

## Register Output Binding

Once you've created an output binding object, you need to tell Shiny to use it:

```
Shiny.outputBindings.register(exampleOutputBinding, "yourname.exampleOutputBinding");
```

The second argument is a string that uniquely identifies your output binding. At the moment it is unused but future features may depend on it.

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

William Buchanan

---

Are there any simple examples that you could show of how to bind a javascript object to a shiny output? For example, I have a .js file to build out a choropleth using d3.js and can integrate it and display it in a shiny app that I'm working on, but I've not been able to figure out how to bind the map to avoid having it show up on every single page. It'd probably help if I had some javascript experience, but if there's anything you could suggest that would be helpful as well.

Xiushi Le

---

I am still a bit confused of this guide, why convert it to a function?

I tried to create a wrapper class on renderDataTable as follows:

```
renderPlainTable <- function(expr, ...) {  
  options = list(searching = FALSE,ordering = FALSE,paging = FALSE,info=FALSE)  
  renderDataTable(expr,options = options,...)
```

comments powered by [Disqus](#)

## 2.47 Add Google Analytics to a Shiny app

# Add Google Analytics to a Shiny app

ADDED: 04 SEP 2014

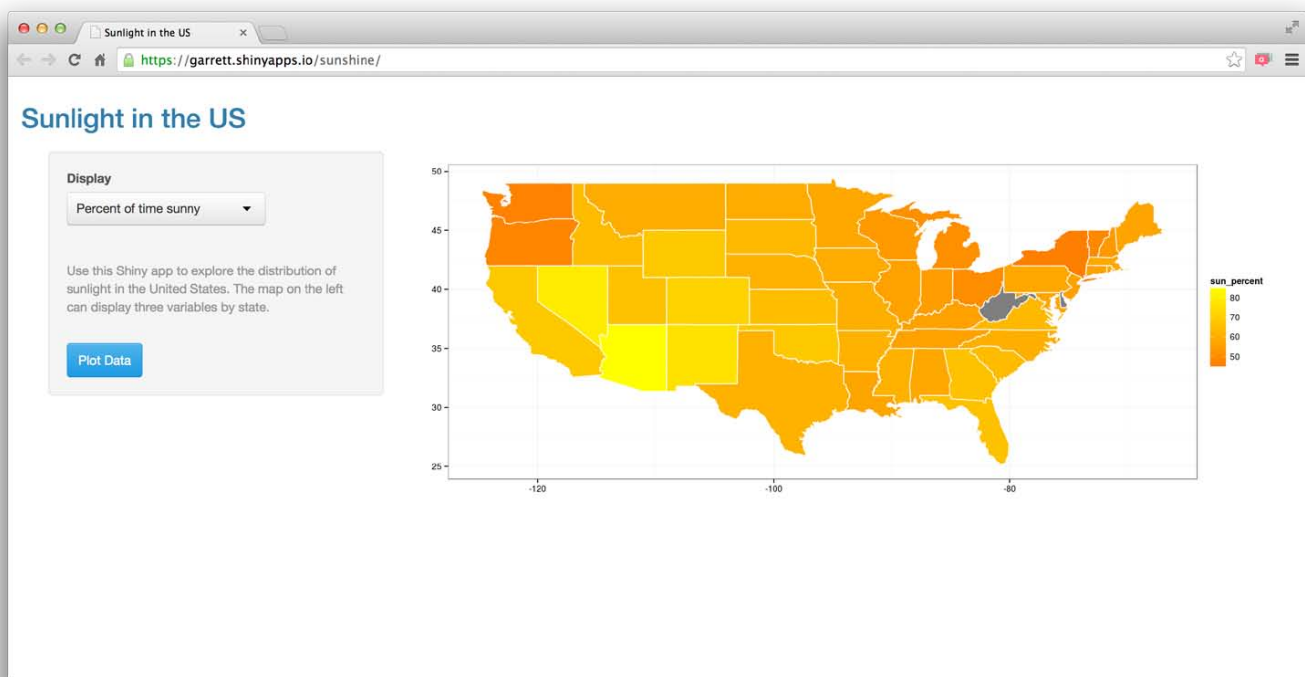
This article will show you how to add [Google Analytics](#) code to a Shiny app. You will need to know a little about JavaScript and jQuery to use this method (which we will not teach here).

[Google Analytics](#) is a free service offered by Google that collects information about who visits your website and what they do while they are there. You can learn more about Google Analytics at [support.google.com/analytics](https://support.google.com/analytics), which explains how to set up and use Google Analytics with a web page.

## Add analytics to an app

You can use Google Analytics with a Shiny app, since Shiny apps are a type of web page. In this article, we will add Google Analytics to the Sunshine app, pictured below.

The Sunshine app displays the distribution of annual sunlight in the United States. The app is hosted on [shinyapps.io](https://shinyapps.io) at [garrett.shinyapps.io/sunshine/](https://garrett.shinyapps.io/sunshine/). If you'd like a copy of the app, including its JavaScript and CSS files, you can download them [here](#).



You can add analytics to the app (and collect results) with the five steps below.

## Step 1 - Create an account



To use Google Analytics, you must open a free account at [www.google.com/analytics/](https://www.google.com/analytics/).

**Start analyzing your site's traffic in 3 steps**

- 1 Sign up for Google Analytics**  
All we need is some basic info about what site you'd like to monitor.
- 2 Add tracking code**  
You'll get a tracking code to paste onto your pages so Google knows when your site is visited.
- 3 Learn about your audience**  
In a few hours you'll be able to start seeing data about your site.

**Start using Google Analytics**

[Sign up](#)

Sign up now, it's easy and free!  
Still have questions? [Help Center](#)

## Step 2 - Add a property

Next, you must register your Shiny app as a *property* in your Google Analytics account. You can do this on the sign up page, or—if you already have a Google Analytics account—you can do this in the Admin tab.

**New Account**

What would you like to track?  
Website Mobile app

**Tracking Method**  
This property works using Universal Analytics. Click Get Tracking ID and implement the Universal Analytics tracking code snippet to complete your set up.

**Setting up your account**

**Account Name** required  
Accounts are the top-most level of organization and contain one or more tracking IDs.  
Name:

**Setting up your property**

**Website Name** required  
Sunshine App

**Website URL** required  
http:// https://garrett.shinyapps.io/sunshine/

**Industry Category**  
Select One

**Reporting Time Zone**

**Admin**

**Upgrade to Universal Analytics**  
Upgrade to get a deeper understanding of your users through new tools and more accurate data. Learn more at the [Universal Analytics Upgrade Center](#).

Select an account and a property, then click **Universal Analytics Upgrade**. Only users with **edit** permission can transfer a property. Each property must be transferred individually.

**Account**  
RStudio-shiny

**PROPERTY**  
Shiny Development Center

**VIEW**  
All Web Site Data

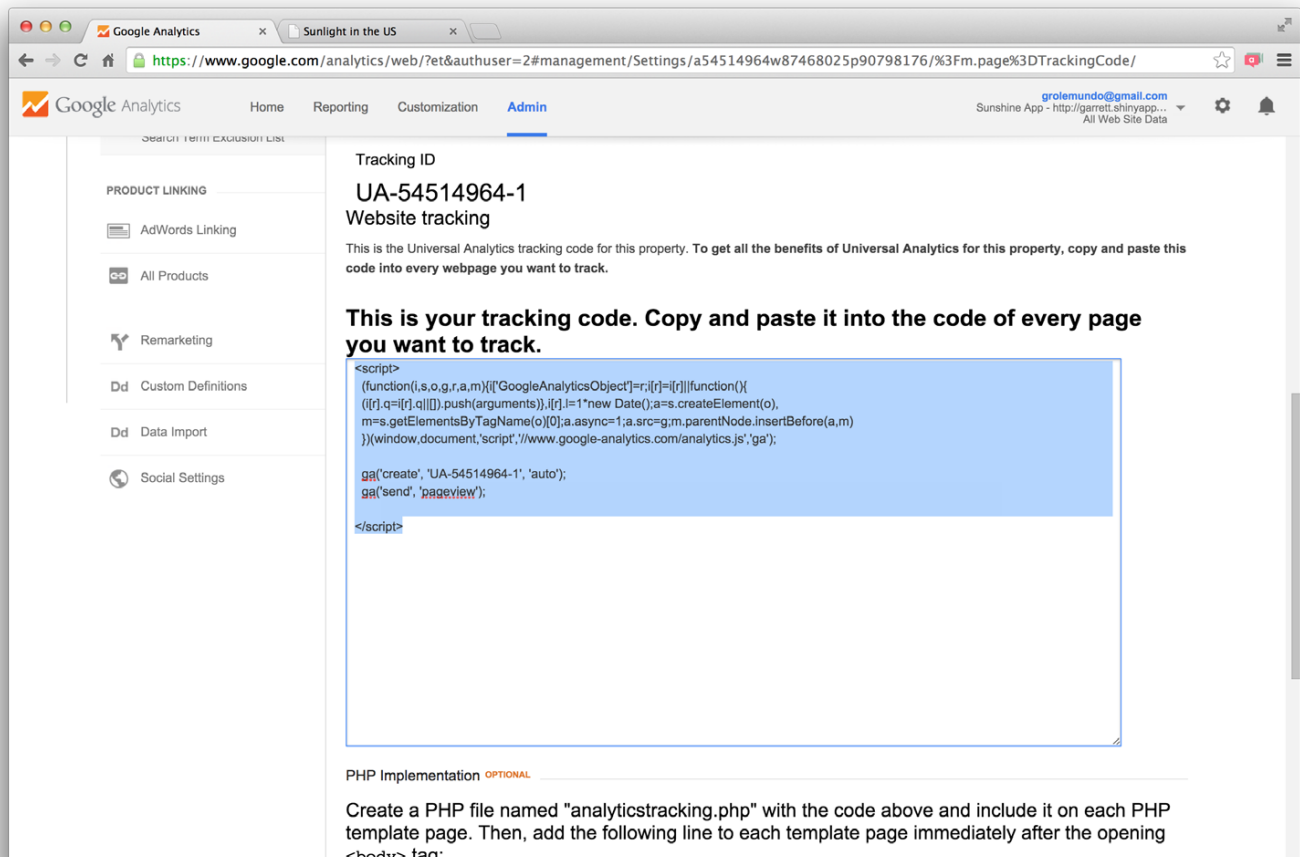
You will need to provide a web address for your app to Google Analytics. Since the Sunshine app is hosted at <https://garrett.shinyapps.io/sunshine>, I'll use this address.

**Website Name** required

Sunshine App

**Website URL** requiredhttp://  https://garrett.shinyapps.io/sunshine/

Once you have entered the necessary information, click “Get Tracking ID” at the bottom of the form. Google Analytics will open a new window that contains a tracking ID number for your app as well as a short JavaScript script.



The screenshot shows the Google Analytics Admin interface. The left sidebar contains navigation options: Search Term Exclusion List, Product Linking, AdWords Linking, All Products, Remarketing, Custom Definitions, Data Import, and Social Settings. The main content area displays the Tracking ID: UA-54514964-1. Below the ID, it states: "This is the Universal Analytics tracking code for this property. To get all the benefits of Universal Analytics for this property, copy and paste this code into every webpage you want to track." A large blue box highlights the JavaScript code to be copied:

```
<script>
(function(i,s,o,g,r,a,m){(['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
(i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date());a=s.createElement(o),
m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)
})(window,document,'script','//www.google-analytics.com/analytics.js','ga');

ga('create', 'UA-54514964-1', 'auto');
ga('send', 'pageview');
</script>
```

Below the code, it says "PHP Implementation **OPTIONAL**". The text continues: "Create a PHP file named "analyticstracking.php" with the code above and include it on each PHP template page. Then, add the following line to each template page immediately after the opening <body> tag:"

This script will allow Google Analytics to track traffic to and from your app. To use it, you'll need to put the script in the head of your app's DOM, the subject of Step 3.

## Step 3 - Embed tracking script

You should place the Google Analytics tracking script at the end of the head section of the HTML DOM that describes your Shiny app.

This is very easy to do if you build your Shiny app around an HTML file, as described in [Build your entire UI with HTML](#).

If you built your Shiny app with a `ui.R` file (the traditional method), use the `tags$head` and `includeScript` functions to include the script.

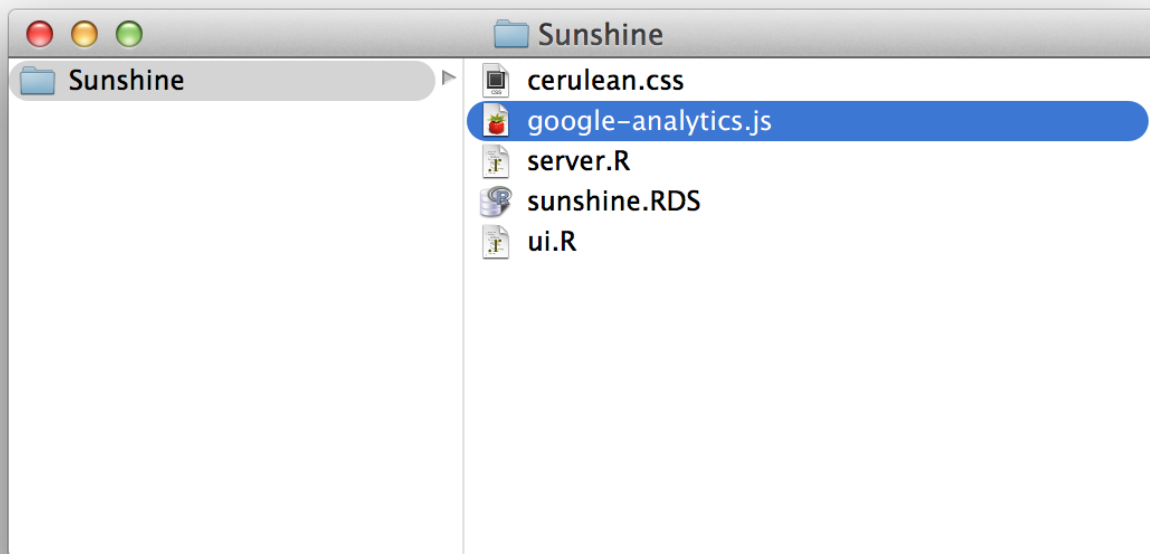
For example, Google Analytics has given us the script below to embed in the Sunshine app. (*Notice that both this code and the event trackers below use Universal Analytics, the more recent version of Google Analytics that replaces Classic Analytics.*)

```
<script>
(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
  (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date();
a=s.createElement(o), m=s.getElementsByTagName(o)[0];
a.async=1;
a.src=g;m.parentNode.insertBefore(a,m)
})(window,document,'script','//www.google-analytics.com/analytics.js','ga');

ga('create','UA-54514964-1','auto');
ga('send','pageview');

</script>
```

To include the script in your app, first save it to its own file. Here I've saved the script as a file named `google-analytics.js`, which I have placed in the working directory of the Sunshine app.



I've also removed the `<script>` and `</script>` tags from the code so my `google-analytics.js` file looks like this. This is necessary because I will add the script to my `ui.R` file with `includeScript`, which will append its own `<script></script>` tags.

```
(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
  (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date();
a=s.createElement(o), m=s.getElementsByTagName(o)[0];
a.async=1;
a.src=g;m.parentNode.insertBefore(a,m)
})(window,document,'script','//www.google-analytics.com/analytics.js','ga');
```

```
ga('create', 'UA-54514964-1', 'auto');
ga('send', 'pageview');
```

To embed the script in the Sunshine app, call `tags$head(includeScript("google-analytics.js"))` in the `ui.R` file. `includeScript` will import the script and pass it to `tags$head()`, which will place the script in the head section of your app's DOM.

`includeScript` requires one argument: `"google-analytics.js"`, the file path from the App directory to the `google-analytics.js` file.

The final `ui.R` file will look like this.

```
# ui.R

library(shiny)
shinyUI(fluidPage(

  tags$head(includeScript("google-analytics.js")),
  includeCSS("cerulean.css"),

  titlePanel("Sunlight in the US"),

  sidebarPanel(
    selectInput("var", "Display",
      choices = c(
        "Percent of time sunny" = "sun_percent",
        "Annual hours of sunshine" = "total_hours",
        "Annual clear days" = "clear_days")),
    br(),
    helpText("Use this Shiny app to explore the
      distribution of sunlight in the United
      States. The map on the left can
      display three variables by state."),
    br(),

    submitButton("Plot Data")
  ),

  mainPanel(width = 10, plotOutput("map"))
))
```

Since the Sunshine app is hosted on `shinyapps.io`, I will need to redeploy the app to `shinyapps.io` for the new `ui.R` file to take effect.

The Google Analytics script tracks how visitors move from one web page to the next. With it, you can learn a little about:

- who visits your app
- where they come from
- how long they stay on the app while they are there

You can also use Google Analytics to track how visitors use your app, but to do that you will need to set up specific event trackers.

## Step 4 - Create event trackers

An event tracker is a piece of code that notifies Google Analytics whenever a visitor interacts with a specific part of

your app, such as a link or a widget. You create a separate event tracker for each unique event that you want to track.

With Google Analytics, you use the basic script to track how people move to and from your page, and you use event trackers to track what they do while they are there. *Note that you will need to embed the basic Google Analytics tracking script into your app before you create any event trackers. Event trackers will not work without the basic script.*

To create an event tracker, you arrange to have a web element's event handler execute a simple JavaScript command that looks like this

```
ga('send', 'event', 'category', 'action', 'label', value);
```

When the element executes the command, `ga` sends an event notification to Google Analytics that let's Google Analytics know that an event occurred. The notification will contain the values of `category`, `action`, `label` and `value` that you have supplied for this type of event. Later on, you will be able to see these values, as well as when the event occurred, from your Google Analytics dashboard.

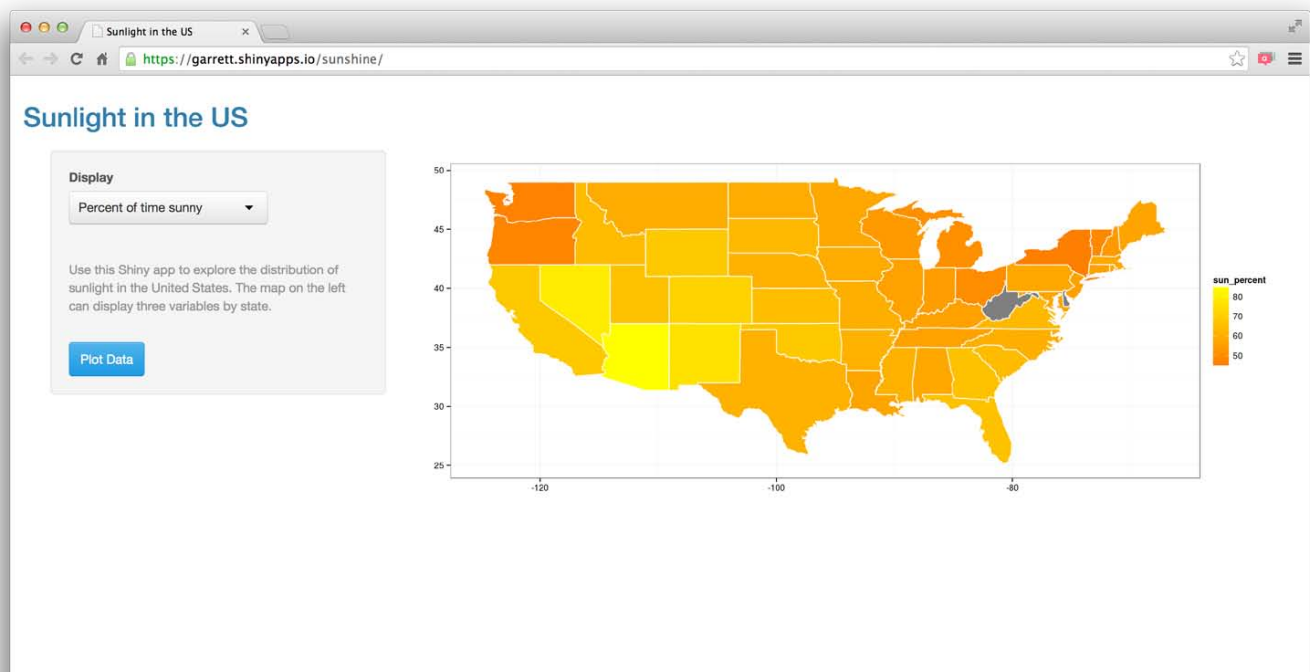
For example, you can track when users click on a specific link by having the links' `onClick` attribute call `ga`.

```
<a href="http://www.example.com" onclick="ga('send', 'event', 'click', 'link', 'I know', 1)">I know when you click me</a>;
```

This works for tracking events on simple web elements, but you must use a different approach to track events on Shiny widgets.

Shiny does not let you set arbitrary attributes on widgets when you create them. To set an event tracker on a Shiny widget, you will need to identify and modify the widget after it has been created, which you can do with a jQuery script.

For example, the Sunshine app has two widgets, a select box and a button. I would like to track how users interact with each of the widgets. To do this I will need to create two event trackers, one for the select box widget and one for the button widget. I will also need to set these event trackers on the widgets with jQuery.



The following script attaches an event handler to the select box widget. The handler will execute for change events that occur on the widget. In other words, the tracker will notify Google Analytics whenever a visitor changes the value of the select box widget.

I've chosen to have the event report include `widget` for the `category` value and `select` data for the `action` value. The last argument will return the value of the new selection as the `label` argument. This tracker will not return a `value` value; `value` arguments are optional.

```
$(document).on('change', 'select', function(e) {
  ga('send', 'event', 'widget', 'select data', $(e.currentTarget).val());
});
```

I can track the app's Plot Data button in a similar way. The script below will send an event notification whenever a user clicks on a button class object in the DOM, e.g. the Plot Data button.

```
$(document).on('click', 'button', function() {
  ga('send', 'event', 'button', 'plot data');
});
```

*Note: to write effective jQuery code, you will need to be able to uniquely identify the widgets that you wish to track. This may require you to explore the document structure of the finished app, for example in your browser's developer tools console.*

When working with Shiny apps, use jQuery code that creates [delegated](#) event handling, like the code above does. Delegated events work more nimbly with the dynamic nature of Shiny apps than do direct events.

Now, that I've written the code that will allow event tracking, I need to add it to the `ui.R` file of the Sunshine app. The easiest way to do this is to include the code in the `google-analytics.js` file that gets added to the app's head.

My final `google-analytics.js` file will look like this

```
(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
  (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date();
a=s.createElement(o), m=s.getElementsByTagName(o)[0];
a.async=1;
a.src=g;m.parentNode.insertBefore(a,m)
})(window,document,'script','//www.google-analytics.com/analytics.js','ga');

ga('create', 'UA-54514964-1', 'auto');
ga('send', 'pageview');

$(document).on('change', 'select', function(e) {
  ga('send', 'event', 'widget', 'select data', $(e.currentTarget).val());
});

$(document).on('click', 'button', function() {
  ga('send', 'event', 'button', 'plot data');
});
```

I'll need to redeploy the app to [shinyapps.io](#) to make sure the hosted version of the app uses this code.

## Step 5 - Collect reports

Once you have set up your app to use Google Analytics, you can use your account portal as a dashboard to track traffic on your app. The Real-Time tracking features should begin almost immediately, but other tracking features may take a couple hours to a couple days before they start reporting results.

You can also extract and analyze traffic data from Google Analytics data with R. Computer World magazine documents [some of the possibilities](#).

## Add Google Analytics with Shiny Server

Alternatively, you can use Shiny Server to add Google Analytics to your apps. This will let you manage your Google Analytics configuration at a higher level—so if you want all apps deployed on a server to share Google Analytics code, you could put this code at the top level of your config and all your apps would automatically inherit it. To learn more, visit the [Shiny Server user guide](#).

## Recap

To add Google Analytics to a Shiny app:

1. Set up a Google Analytics account.
2. Add the Shiny app to the account as a web property.
3. Place the Google Analytics tracking script into the head section of your app's DOM.
4. Create event trackers to track specific events within your app. The easiest way to do this is to modify Shiny widgets with delegated event handlers managed by jQuery.
5. Use R or your Google Analytics dashboard to explore the results.

Google Analytics isn't the only website monitoring service, but it is the most popular. You can add similar services to your app with the same techniques.

These techniques serve as a proof of concept that will allow you to track how visitors use your apps. We will look to make it easier to track Shiny apps in future development.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

ImAndy

---

Garrett Before delving too deeply, can you confirm that this also works with markdown shiny docs?

Orestis

---

if i am not wrong you load the data from sunshine.rds This application must load the data from the google analytics account? and if yes how me can do it?

Joe

---

Is there a way to label the page so that Google Analytics does not just think it is the homepage?

Joe

---

For anybody else interested, I found the answer. GA may not know how to distinguish the page name from [shinyapps.io](http://shinyapps.io). To give the page a name for GA, you can use the following code as an example.

comments powered by Disqus



## 2.48 How to create User Privileges

# How to create User Privileges

ADDED: 03 APR 2014

BY: JEFF ALLEN WITH GARRETT GROLEMUND

It is easy to create User Privileges in Shiny apps when you use [Shiny Server Pro](#). Your app can recognize a user based on log-in information and deliver personalized content in response. You can use this feature to control who gets to see what content and when they see it.

## Personalized Data Access

The [Sales Report](#) app demonstrates how you can use this feature to control data your app displays. You can view this app and its code in the [Shiny Gallery](#).

When a sales person logs into the Sales Report app, Shiny displays the data related only to that sales person.

Shiny by RStudio BACK TO GALLERY GET CODE SHARE Search

Logged in as sales1 Logout

This app requires users to log in. The data it makes available in the user interface will vary, depending on who is logged in.

Choose one of the username/password pairs below:

Username	Password
manager	password
sales1	password
sales2	password

### Sales Reports

#### Monthly Sales Report for 'sales1'

April Sales Projections

Total Sales (USD)

Day of Month

25 records per page Search:

salesperson	day	dailySales	salesTotal
sales1	0	0.00	0.00
sales1	1	758.85	758.85
sales1	2	113.94	872.79
sales1	3	690.76	1563.55
sales1	4	516.40	2079.95
sales1	5	67.74	2147.69
sales1	6	738.71	2886.40

When a manager logs into the Sales Report app, Shiny displays the data for every sales person. This information can help the manager compare performance.

Shiny by RStudio BACK TO GALLERY GET CODE SHARE Search

Logged in as manager Logout

## Sales Reports

This app requires users to log in. The data it makes available in the user interface will vary, depending on who is logged in.

Choose one of the username/password pairs below:

Username	Password
manager	password
sales1	password
sales2	password

### Monthly Manager Sales Report

April Sales Projections

25 records per page Search:

salesperson	day	dailySales	salesTotal
sales1	0	0.00	0.00
sales2	0	0.00	0.00
sales1	1	758.85	758.85
sales2	1	824.13	824.13
sales1	2	113.94	872.79
sales2	2	90.77	914.90
sales1	3	690.76	1583.55

The Sales Report app creates this effect by creating personal output for each log-in.

If you want to create a similar effect, begin in the `server.R` file. In that file, you can access your user's log-in information with `session$user` (the topic of [Learn about your user with session\\$clientData](#)).

First include `session` as the third argument of the `shinyServer` function. The Sales Report app does this in line 29 of its `server.R` `server.R` file. Including `session` will ensure that the code in `shinyServer` has access to the variable at runtime.

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

Next access and store your user's username with `session$user`. The Sales Report app does this in lines 34-36 of its `server.R` script.

```
  user <- reactive({
    session$user
  })
```

After you store usernames with `session$user`, you can use the user information with `switch`, `if`, and other conditional functions. These functions will build outputs tailored for the user. The Sales Report app does this by testing whether the user's log-in matches the names of known manager log-ins (here "manager"). The Sales Report app includes a helper function that runs this test.

```
  isManager <- reactive({
    if (user() == "manager"){
      return(TRUE)
    } else{
      return(FALSE)
    }
  })
```

It uses the results to determine the scope of the data set to display.

```
# based on the logged in user, pull out only the data this user should be able
# to see.
myData <- reactive({
  if (isManager()){
    # If a manager, show everything.
    return(salesData)
  } else{
    # If a regular salesperson, only show their own sales.
    return(salesData[salesData$salesperson == user(),])
  }
})
```

This approach is very versatile and provides a convenient way to control who has access to which data and which widgets in your Shiny apps.

The [Airline Delays](#) app pushes this method a step further. The Airline Delays app compares the user log-in to a table of known users. Then it uses `renderUI` to create a personalized user-interface for each user. See [Build a dynamic UI that reacts to user input](#) for more tips on rendering a custom UI with `renderUI`.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

## 2.49 Allow different libraries for different

# Allow different libraries for different apps

ADDED: 21 JUL 2014

BY: JEFF ALLEN

This article describes two ways to configure applications to use different sets of libraries in Shiny Server.

- Method 1 relies on Shiny Server's `exec_supervisor` feature. It will only work with Shiny Server Pro.
- Method 2 relies on Shiny Server's `run_as` feature. It will work with both Shiny Server and Shiny Server Pro.

## Method 1 - `exec_supervisor`

With Shiny Server Pro, you can run R sessions under a program supervisor that modifies the environment of the sessions. You can use this supervisor to set the `R_LIBS_USER` environmental variable, which controls which libraries a session may use.

Section 3.6 of the Shiny Server [Administrator's Guide](#) explains how to add a program supervisor. You add an `exec_supervisor` setting to your server config file to specify a supervisor (and the arguments which control its behavior).

The file below uses `exec_supervisor` to modify the default `/etc/shiny-server/shiny-server.conf` file that ships with Shiny Server. `exec_supervisor` partitions the applications up by setting the `R_LIBS_USER` environment variable.

Near the bottom of the file, a `/finance` sub-location is defined for apps that use a specific set of libraries. Beneath that, a specific app is given its own set of libraries.

```
# Instruct Shiny Server to run applications as the user "shiny"
run_as shiny;

# Specify the authentication method to be used.
# Initially, a flat-file database stored at the path below.
auth_passwd_file /etc/shiny-server/passwd;

# Define a server that listens on port 3838
server {
  listen 3838;

  # Define a location at the base URL
  location / {
    # Define a default library for applications
    exec_supervisor "R_LIBS_USER=/usr/lib/LibraryA";

    # Only up to 20 connections per Shiny process and
    # at most 3 Shiny processes per application.
```

```

# proactively spawn a new process when our processes
# reach 90% capacity.
utilization_scheduler 20 .9 3;

# Host the directory of Shiny apps stored in this directory
site_dir /srv/shiny-server;

# Log all Shiny output to files in this directory
log_dir /var/log/shiny-server;

# When a user visits the base URL rather than a particular application,
# an index of the applications available in this directory will be shown.
directory_index on;

# Now define a sub-location at /finance
location /finance {
    # Define a library that should be used by the finance department
    exec_supervisor "R_LIBS_USER=/usr/lib/FinanceLibrary";

    # Further, define another sub-location that happens to correspond to
    # a particular app.
    location /app1 {
        #Define a specific library to be used by this application
        exec_supervisor "R_LIBS_USER=/usr/lib/LibraryC";
    }
}

# Provide the admin interface on port 4151
admin 4151 {

    # Restrict the admin interface to the usernames listed here. Currently
    # just one user named "admin"
    required_user admin;
}

```

In this case, you could set up as many different libraries as you want and specify a different library for each location, or even each application that you want to deploy. This would give you fine-grained control over each of your applications.

## Method 2 - run\_as

Shiny Server (and Shiny Server Pro) use a `run_as` setting to determine which user should spawn each R Shiny processes. The user setting determines which library R will look in for packages (as well as which directories the app will be able to read and write to).

The `run_as` setting can be configured globally, or for a particular server or location. As a result, you can set up different locations for hosting apps that use different packages. Each location can be affiliated with its own user—each of which presumably has a different set of libraries specified with a `~/.Rprofile` file.

With this approach, you would probably have only a handful of users that you create that each maintain their own separate libraries.

Section 2.3 of the [Shiny Server Administrator's Guide](#) explains how to set a user with `run_as`.

The file below uses `run_as` to modify the default `/etc/shiny-server/shiny-server.conf` file that ships with Shiny Server. `run_as` defines a global user, `shiny`, for the server. It then defines a different user for the `/finance` location, `shinyFinance`. The finance department can deploy apps in this location. Those apps will be restricted to the packages in the library of `shinyFinance`, which may be different than the packages in the library of the user named `shiny`.

```
# Specify the authentication method to be used.
# Initially, a flat-file database stored at the path below.
auth_passwd_file /etc/shiny-server/passwd;

# Define a server that listens on port 3838
server {
  listen 3838;

  # Define a location at the base URL
  location / {
    # Instruct Shiny Server to run applications as
    # the user "shiny" by default
    run_as shiny;

    # Only up to 20 connections per Shiny process and
    # at most 3 Shiny processes per application.
    # Proactively spawn a new process when our processes
    # reach 90% capacity.
    utilization_scheduler 20 .9 3;

    # Host the directory of Shiny apps stored in this directory
    site_dir /srv/shiny-server;

    # Log all Shiny output to files in this directory
    log_dir /var/log/shiny-server;

    # When a user visits the base URL rather than a particular application,
    # an index of the applications available in this directory will be shown.
    directory_index on;

    location /finance {
      # Run as a different user for this location
      run_as shinyFinance;
    }
  }
}

# Provide the admin interface on port 4151
admin 4151 {

  # Restrict the admin interface to the usernames listed here. Currently
  # just one user named "admin"
  required_user admin;
}
```

questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

comments powered by Disqus

## 2.50 Interactive plots

# Interactive plots

ADDED: 13 MAY 2015

As of version 0.12.0, Shiny has built-in support for interacting with static plots generated by R's base graphics functions, and those generated by ggplot2.

This makes it easy to add features like selecting points and regions, as well as zooming in and out of images.

## Basics

To get the position of the mouse when a plot is clicked, you simply need to use the `click` option with the `plotOutput()`. For example, this will define a new input value, `input$plot_click`, which contains the location of the previous mouse click.

```
plotOutput("plot1", click = "plot_click")
```

For example, this app will print out the x and y position of the mouse cursor when a click occurs (to see it in action, cut and paste this code into the R console):

```
library(shiny)

ui <- basicPage(
  plotOutput("plot1", click = "plot_click"),
  verbatimTextOutput("info")
)

server <- function(input, output) {
  output$plot1 <- renderPlot({
    plot(mtcars$wt, mtcars$mpg)
  })

  output$info <- renderText({
    paste0("x=", input$plot_click$x, "\ny=", input$plot_click$y)
  })
}

shinyApp(ui, server)
```





please wait

⋮

Notice that the x and y coordinates are scaled to the data, as opposed to simply being the pixel coordinates. This makes it easy to use those values to select or filter data.

The other types of interactions are double-clicking, hovering, and brushing. (Brushing is clicking and dragging a selection box.) They can be enabled with the `dbl click`, `hover`, and `brush` options. In the example below, all of these are enabled, and the coordinates are displayed below

```
ui <- basicPage(
  plotOutput("plot1",
    click = "plot_click",
    dbl click = "plot_dbl click",
    hover = "plot_hover",
    brush = "plot_brush"
  ),
  verbatimTextOutput("info")
)

server <- function(input, output) {
  output$plot1 <- renderPlot({
    plot(mtcars$wt, mtcars$mpg)
  })

  output$info <- renderText({
    xy_str <- function(e) {
      if(is.null(e)) return("NULL\n")
      paste0("x=", round(e$x, 1), " y=", round(e$y, 1), "\n")
    }
    xy_range_str <- function(e) {
      if(is.null(e)) return("NULL\n")
      paste0("xmi n=", round(e$xmi n, 1), " xmax=", round(e$xmax, 1),
        " ymi n=", round(e$ymi n, 1), " ymax=", round(e$ymax, 1))
    }

    paste0(
      "click: ", xy_str(input$plot_click),
      "dbl click: ", xy_str(input$plot_dbl click),
```

```
"hover: ", xy_str(input$plot_hover),  
"brush: ", xy_range_str(input$plot_brush)  
)  
})  
}  
  
shinyApp(ui, server)
```

please wait

⋮⋮⋮

While `click`, `dblclick`, and `hover` have x and y coordinates, `brush` is slightly different: because it's a box, it has `xmin`, `xmax`, `ymin`, and `ymax`.

Next: learn about how to easily [select rows of data](#) with interactive plots.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

## 2.51 Selecting rows of data

# Selecting rows of data

ADDED: 13 MAY 2015

A common use of mouse interactions is to select rows of data from an input data frame. Although you could write code that uses the `x` and `y` (or the corresponding min and max) values to filter rows from the data frame, there is an easier way to do it. Shiny provides two convenience functions for selecting rows of data:

- `nearPoints()` : Uses the `x` and `y` value from the interaction data; to be used with `click`, `dblclick`, and `hover`.
- `brushedPoints()` : Uses the `xmin`, `xmax`, `ymin`, and `ymax` values from the interaction data; to be used with `brush`.

Note that these functions are only appropriate if the `x` and `y` variables are present in the data frame, without any transformation. If, for example, you have a plot where a the `x` position is *calculated* from a column of data, then these functions won't work. In such a case, it may be useful to first calculate a new column and store it in the data frame.

### Selection with `nearPoints()`

Here is a basic example of the `nearPoints` function. If you pass it the data frame with the plotted data, the mouse interaction object from `input`, and the names of the `x` and `y` variables, it will return a data frame with just selected rows.

```
ui <- basicPage(
  plotOutput("plot1", click = "plot_click"),
  verbatimTextOutput("info")
)

server <- function(input, output) {
  output$plot1 <- renderPlot({
    plot(mtcars$wt, mtcars$mpg)
  })

  output$info <- renderPrint({
    # With base graphics, need to tell it what the x and y variables are.
    nearPoints(mtcars, input$plot_click, xvar = "wt", yvar = "mpg")
    # nearPoints() also works with hover and dblclick events
  })
}

shinyApp(ui, server)
```



please wait

▢▢▢

By default, `nearPoints` will return all rows of data that are within 5 pixels of the mouse event, and they will be sorted by distance, with the nearest first. The radius can be customized with `threshold`, and the number of rows returned can be customized with `maxpoints`.

If you're using plots created by `ggplot2`, it's not necessary to specify `xvar` and `yvar`, since they can be autodetected. (Bear in mind that if the variables are *calculated* from the data – for example with `aes(x = wt/2)` – this won't work.)

The version below uses a plot with `ggplot2`, and displays the one point that is closest to the click, and within 10 pixels.

```
library(ggplot2)
ui <- basicPage(
  plotOutput("plot1", click = "plot_click"),
  verbatimTextOutput("info")
)

server <- function(input, output) {
  output$plot1 <- renderPlot({
    ggplot(mtcars, aes(x=wt, y=mpg)) + geom_point()
  })

  output$info <- renderPrint({
    # With ggplot2, no need to tell it what the x and y variables are.
    # threshold: set max distance, in pixels
    # maxpoints: maximum number of rows to return
    # addDist: add column with distance, in pixels
    nearPoints(mtcars, input$plot_click, threshold = 10, maxpoints = 1,
              addDist = TRUE)
  })
}

shinyApp(ui, server)
```

please wait



## Selection with `brushedPoints()`

To select rows using a brush, use the `brushedPoints()` function. Basic usage is similar to `nearPoints()` : it returns the rows of data that are under the brush.

```
ui <- basicPage(  
  plotOutput("plot1", brush = "plot_brush"),  
  verbatimTextOutput("info")  
)  
  
server <- function(input, output) {  
  output$plot1 <- renderPlot({  
    plot(mtcars$wt, mtcars$mpg)  
  })  
  
  output$info <- renderPrint({  
    # With base graphics, need to tell it what the x and y variables are.  
    brushedPoints(mtcars, input$plot_brush, xvar = "wt", yvar = "mpg")  
  })  
}  
  
shinyApp(ui, server)
```

please wait



With `ggplot2` graphics, you don't need to supply `xvar` and `yvar` because they can be inferred automatically. Also, `brushedPoints()` and `nearPoints()` both work with facets in `ggplot2`.

```
library(ggplot2)
ui <- basicPage(
  plotOutput("plot1", brush = "plot_brush", height = 250),
  verbatimTextOutput("info")
)

server <- function(input, output) {
  output$plot1 <- renderPlot({
    ggplot(mtcars, aes(x=wt, y=mpg)) + geom_point() +
      facet_grid(. ~ cyl) +
      theme_bw()
  })

  output$info <- renderPrint({
    brushedPoints(mtcars, input$plot_brush)
  })
}

shinyApp(ui, server)
```

please wait



## Getting the position of selected rows

Instead of getting just the selected rows, it's sometimes useful to get all the rows, but with a column indicating which rows are selected. For both `nearPoints()` and `brushedPoints()`, you can do this with the `allRows` option.

```
library(ggplot2)
ui <- basicPage(
  plotOutput("plot1", brush = "plot_brush"),
  verbatimTextOutput("info")
)

server <- function(input, output) {
  options(width = 100) # Increase text width for printing table
  output$plot1 <- renderPlot({
    ggplot(mtcars, aes(x=wt, y=mpg)) + geom_point()
  })

  output$info <- renderPrint({
    brushedPoints(mtcars, input$plot_brush, allRows = TRUE)
  })
}

shinyApp(ui, server)
```

This can be useful if you need the row position of the selected points. For example, this can be used to allow clicking on points to exclude them from an analysis, as in this example where you can exclude points from a linear model.

please wait



## Options

Mouse interactions have default settings that suitable for most use cases, but the settings can be customized.

To do this, for the `click`, `dblclick`, `hover`, and `brush` options, instead of passing them a string, you would pass them the value from `clickOpts()`, `dblclickOpts()`, `hoverOpts()`, or `brushOpts()`.

For example, by default, a brush is a light transparent blue, and it can be controlled in both the vertical and horizontal directions. In the code below, using the `brushOpts()` function, we use the same input ID as before, `"plot_brush"`, but now we can set the fill color to a light gray, and make the brush operate just in the horizontal direction.

```
library(ggplot2)
ui <- basicPage(
  plotOutput("plot1",
    brush = brushOpts(id = "plot_brush", fill = "#ccc", direction = "x"),
    height = 250
  )
)

server <- function(input, output) {
  output$plot1 <- renderPlot({
    ggplot(ChickWeight, aes(x=Time, y=weight, colour=factor(Chick))) +
      geom_line() +
      guides(colour=FALSE) +
      theme_bw()
  })
}

shinyApp(ui, server)
```

please wait

⋮



For more information about the available options, see `?clickOpts` , `?dblclickOpts` , `?hoverOpts` , and `?brushOpts` .

Next: learn about [advanced plot interaction features](#).

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Aspro

---

This is a really great tutorial.  
Is this applicable to 3D plots also?

Emma Bassein

---

This is super helpful, thanks! Is there any way to be able to make multiple brush selections at once? Like if you wanted to select two different areas of non-continuous data.

comments powered by [Disqus](#)

## 2.52 Interactive plots - advanced

# Interactive plots - advanced

ADDED: 25 MAY 2015

This article contains information about using Shiny's image and plot interaction features to perform some more advanced tasks.

To get a look at most of the features available in plot interactions, see the [advanced demo app](#).

## Interactions with bitmap images

The [plot interaction](#) article describes how to interact with plots generated by R's base graphics and ggplot2. Shiny also supports interactions with arbitrary bitmap (for example, PNG or JPEG) images. There is one change in the information returned for these mouse events: instead of plot coordinates scaled to the data, they will contain pixel coordinates. You may need to transform these coordinates to something useful for your data.

The only difference in the code is that, instead of using `renderPlot()`, you would use `renderImage()`. For an example, see the [image interaction demo app](#).

## Mouse event data

If you'd like to see the data structures returned by mouse interactions, see the [basic demo app](#).

## Zooming

Mouse interactions can be used to implement zooming in plots. The [zooming demo app](#) shows two ways of doing this: by zooming in a single plot, and by using one plot to control the zoom in a second plot.

## Excluding points from a scatter plot

It can be useful to interactively select outliers to exclude from a prediction model. The [exclude demo app](#) shows how to do this.

## Dates and date-times

When dates and date-times are used on the x or y axis, the selected values will be returned from the browser as numeric values. The `nearPoints()` and `brushedPoints()` functions will automatically handle the type conversions, but if you want to do the conversions manually, you would use something like the following:

```
# If the x variable is a Date
as.Date(input$plot_click$x, origin = "1970-01-01")

# If the y variable is POSIXct
as.POSIXct(input$plot_click$y, origin = "1970-01-01")
```

The `origin` is the date or time to count from, and midnight on 1970-01-01 is the usual value.

Note: for datetimes, it is generally preferable to use data of class `POSIXct` instead of `POSIXlt`, because the storage format of `POSIXlt` is more difficult to work with.

Another possibility is, instead of converting the mouse coordinates to dates or times, you could convert the data values to numbers, and then do some comparison with the input values:

```
# If the x variable, in data$dates, is a Date
# Find which rows are within 1 day of the click
selectedRows <- abs(as.numeric(data$dates) - input$plot_click$x) < 1
```

## Categorical axes (including bar graphs)

For plots that have axes with categorical values (factors or character vectors), the values returned from the browser will be numeric. To compare the mouse coordinate values the data values, you will need to coerce the data to numeric values.

For mouse click/double-click/hover events, you will typically want to round the mouse's x or y value so that it can be compared to the data values. The app below demonstrates how to do this:

```
library(ggplot2)

ui <- fluidPage(
  fluidRow(
    plotOutput("plot1", height = 300, width = 300,
      click = "plot1_click",
    )
  ),
  verbatimTextOutput("x_value"),
  verbatimTextOutput("selected_rows")
)

server <- function(input, output) {
  output$plot1 <- renderPlot({
    plot(ToothGrowth$supp, ToothGrowth$len)
  })

  # Print the name of the x value
  output$x_value <- renderPrint({
    if (is.null(input$plot1_click$x)) return()

    lvl <- levels(ToothGrowth$supp)
    lvl[round(input$plot1_click$x)]
  })

  # Print the rows of the data frame which match the x value
  output$selected_rows <- renderPrint({
    if (is.null(input$plot1_click$x)) return()

    keeprows <- round(input$plot1_click$x) == as.numeric(ToothGrowth$supp)
    ToothGrowth[keeprows, ]
  })
}

shinyApp(ui, server)
```

For brushing, it usually make more sense to check if a factor level's corresponding numeric value is within the `xmin` and `xmax` (or `ymin` and `ymax`).

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

comments powered by Disqus

## 2.53 Upgrade notes for Shiny 0.11

# Upgrade notes for Shiny 0.11

ADDED: 06 JAN 2015

BY: WINSTON CHANG

Shiny 0.11 switches away from the Bootstrap 2 web framework to the next version, Bootstrap 3. This is in part because Bootstrap 2 is no longer being developed, and in part because it allows us to tap into the ecosystem of Bootstrap 3 themes.

## Known issues for migration

- In Bootstrap 3, images in `<img>` tags are no longer automatically scaled to the width of their container. If you use `img()` in your UI code, or `<img>` tags in your raw HTML source, it's possible that they will be too large in the new version of Shiny. To address this you can add the `img-responsive` class:

```
img(src = "picture.png", class = "img-responsive")
```

The R code above will generate the following HTML:

```

```

- The sliders have been replaced. Previously, Shiny used the `jquery.slider` library, but now it uses `ion.RangeSlider`. The new sliders have an updated appearance, and they have allowed us to fix many long-standing interface issues with the sliders.
  - The `sliderInput()` function no longer uses the `format` or `locale` options. Instead, you can use `pre`, `post`, and `sep` options to control the prefix, postfix, and thousands separator.
  - `updateSliderInput()` can now control the min, max, value, and step size of a slider. Previously, only the value could be controlled this way, and if you wanted to change other values, you needed to use Shiny's dynamic UI.
- If in your HTML you are using custom CSS classes that are specific to Bootstrap, you may need to update them for Bootstrap 3. See the Bootstrap [migration guide](#).

If you encounter other migration issues, please let us know on the [shiny-discuss](#) mailing list, or on the Shiny [issue tracker](#).

## Using shinybootstrap2

If you would like to use Shiny 0.11 with Bootstrap 2, you can use the `shinybootstrap2` package. Installation and usage instructions are available on the [project page](#). We recommend that you do this only as a temporary solution because future development on Shiny will use Bootstrap 3.

# Installing an older version of Shiny

If you want to install a specific version of Shiny other than the latest CRAN release, you can use the `install_version()` function from devtools:

```
# Install devtools if you don't already have it:
# install.packages("devtools")

# Install the last version of Shiny prior to 0.11
devtools::install_version("shiny", "0.10.2.2")
```

## Themes

Along with the release of Shiny 0.11, we've packaged up some Bootstrap 3 themes in the [shinythemes](#) package. This package makes it easy to use Bootstrap themes with Shiny.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

Alex Lemm

---

For those of you who struggled with downgrading Shiny using `install_version` under Windows like I did, here is an alternative which worked for me:  
`devtools::install_github("rstudio/shiny@v0.10.2.2")`

Samer Mouksassi

---

Thanks indeed : `devtools::install_version("shiny", "0.10.2.2")` was not working on my end !

## 2.54 Upgrade notes for Shiny 0.12

# Upgrade notes for Shiny 0.12

ADDED: 29 MAY 2015

BY: WINSTON CHANG

In addition to the changes listed in the [NEWS](#) file for Shiny 0.12.0, there is an infrastructure change that could affect existing Shiny apps.

## JSON serialization

In Shiny 0.12.0, we've switched from RJSONIO to jsonlite. For the vast majority of users, this will result in no noticeable changes; however, if you use any packages in your Shiny apps which rely on the [htmlwidgets](#), you will also need to update htmlwidgets to 0.4.0. Both of these packages will issue a message when loaded, if the other package needs to be upgraded.

POSIXt objects are now serialized to JSON in UTC8601 format (like "2015-03-20T20:00:00Z"), instead of as seconds from the epoch. If you have a Shiny app which uses `sendCustomMessage()` to send datetime (POSIXt) objects, then you may need to modify your Javascript code to receive time data in this format.

## A note about Data Tables

Shiny 0.12.0 deprecated Shiny's `dataTableOutput` and `renderDataTable` functions and instructed you to migrate to the nascent [DT](#) package instead. (We'll talk more about DT in a future blog post.) User feedback has indicated this transition was too sudden and abrupt, so we've undeprecated these functions in 0.12.1. We'll continue to support these functions until DT has had more time to mature.

---

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](#) for in depth discussion of Shiny related questions and [How to get help](#) for a list of the best ways to get help with R code.

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

[Firefox](#)

[Chrome](#)

[Internet Explorer 10+](#)

[Safari](#)

## 3 Function Reference

### 3.1 Function reference version 0.12.1

# Function reference version 0.12.1

## UI Layout

---

Functions for laying out the user interface for your application.

<code>absolutePanel</code> ( <code>fixedPanel</code> )	Panel with absolute positioning
<code>bootstrapPage</code> ( <code>basicPage</code> )	Create a Bootstrap page
<code>column</code>	Create a column within a UI definition
<code>conditionalPanel</code>	Conditional Panel
<code>fixedPage</code> ( <code>fixedRow</code> )	Create a page with a fixed layout
<code>fluidPage</code> ( <code>fluidRow</code> )	Create a page with fluid layout
<code>headerPanel</code>	Create a header panel
<code>helpText</code>	Create a help text element
<code>icon</code>	Create an icon
<code>mainPanel</code>	Create a main panel
<code>navbarPage</code> ( <code>navbarMenu</code> )	Create a page with a top level navigation bar
<code>navlistPanel</code>	Create a navigation list panel
<code>pageWithSidebar</code>	Create a page with a sidebar
<code>sidebarLayout</code>	Layout a sidebar and main area
<code>sidebarPanel</code>	Create a sidebar panel
<code>tabpanel</code>	Create a tab panel
<code>tabsetPanel</code>	Create a tabset panel
<code>titlePanel</code>	Create a panel containing an application title.
<code>inputPanel</code>	Input panel
<code>flowLayout</code>	Flow layout
<code>splitLayout</code>	Split layout
<code>verticalLayout</code>	Lay out UI elements vertically
<code>wellPanel</code>	Create a well panel
<code>withMathJax</code>	Load the MathJax library and typeset math expressions



## UI Inputs

---

Functions for creating user interface elements that prompt the user for input values or interaction.

<code>actionButton</code> ( <code>actionLink</code> )	Action button/link
<code>checkboxGroupInput</code>	Checkbox Group Input Control
<code>checkboxInput</code>	Checkbox Input Control
<code>dateInput</code>	Create date input
<code>dateRangeInput</code>	Create date range input
<code>fileInput</code>	File Upload Control
<code>numericInput</code>	Create a numeric input control
<code>radioButtons</code>	Create radio buttons
<code>selectInput</code> ( <code>selectizeInput</code> )	Create a select list input control
<code>sliderInput</code> ( <code>animationOptions</code> )	Slider Input Widget
<code>submitButton</code>	Create a submit button
<code>textInput</code>	Create a text input control
<code>passwordInput</code>	Create a password input control
<code>updateCheckboxGroupInput</code>	Change the value of a checkbox group input on the client
<code>updateCheckboxInput</code>	Change the value of a checkbox input on the client
<code>updateDateInput</code>	Change the value of a date input on the client
<code>updateDateRangeInput</code>	Change the start and end values of a date range input on the client
<code>updateNumericInput</code>	Change the value of a number input on the client
<code>updateRadioButtons</code>	Change the value of a radio input on the client
<code>updateSelectInput</code> ( <code>updateSelectizeInput</code> )	Change the value of a select input on the client
<code>updateSliderInput</code>	Change the value of a slider input on the client
<code>updateTabsetPanel</code>	Change the selected tab on the client
<code>updateTextInput</code>	Change the value of a text input on the client

## UI Outputs

---

Functions for creating user interface elements that, in conjunction with rendering functions, display different kinds of output from your application.

<code>htmlOutput</code> ( <code>uiOutput</code> )	Create an HTML output element
<code>imageOutput</code> ( <code>plotOutput</code> )	Create an plot or image output element
<code>outputOptions</code>	Set options for an output object.
<code>tableOutput</code>	Create a table output element

`(dataTableOutput)``textOutput`

Create a text output element

`verbatimTextOutput`

Create a verbatim text output element

`downloadButton``(downloadLink)`

Create a download button or link

`Progress`

Reporting progress (object-oriented API)

`withProgress``(incProgress,``setProgress)`

Reporting progress (functional API)

## Interface builder functions

---

A sub-library for writing HTML using R functions. These functions form the foundation on which the higher level user interface functions are built, and can also be used in your Shiny UI to provide custom HTML, CSS, and JavaScript.

`builder (a, br, code,  
div, em, h1, h2, h3, h4,  
h5, h6, hr, img, p, pre,  
span, strong, tags)`

HTML Builder Functions

`HTML`

Mark Characters as HTML

`include (includeCSS,  
includeHTML,  
includeMarkdown,  
includeScript,  
includeText)`

Include Content From a File

`singleton (is.singleton)`

Include content only once

`tag (tagAppendAttributes,  
tagAppendChild,  
tagAppendChildren,  
tagList, tagSetChildren)`

HTML Tag Object

`validateCssUnit`

Validate proper CSS formatting of a unit

`withTags`

Evaluate an expression using

## Rendering functions

---

Functions that you use in your application's server side code, assigning them to outputs that appear in your user interface.

`renderPlot`

Plot Output

`renderText`

Text Output

`renderPrint`

Printable Output

`renderDataTable`

Table output with the JavaScript library DataTables

`renderImage`

Image file output

`renderTable`

Table Output

<code>renderUI</code>	UI Output
<code>downloadHandler</code>	File Downloads
<code>reactivePlot</code>	Plot output (deprecated)
<code>reactivePrint</code>	Print output (deprecated)
<code>reactiveTable</code>	Table output (deprecated)
<code>reactiveText</code>	Text output (deprecated)
<code>reactiveUI</code>	UI output (deprecated)

## Reactive constructs

---

A sub-library that provides reactive programming facilities for R.

<code>invalidateLater</code>	Scheduled Invalidation
<code>is.reactivevalues</code>	Checks whether an object is a reactivevalues object
<code>isolate</code>	Create a non-reactive scope for an expression
<code>makeReactiveBinding</code>	Make a reactive variable
<code>observe</code>	Create a reactive observer
<code>observeEvent</code> ( <code>eventReactive</code> )	Event handler
<code>reactive</code> ( <code>is.reactive</code> )	Create a reactive expression
<code>reactiveFileReader</code>	Reactive file reader
<code>reactivePoll</code>	Reactive polling
<code>reactiveTimer</code>	Timer
<code>reactiveValues</code>	Create an object for storing reactive values
<code>reactiveValuesToList</code>	Convert a reactivevalues object to a list
<code>domains</code> ( <code>getDefaultReactiveDomain</code> , <code>onReactiveDomainEnded</code> , <code>withReactiveDomain</code> )	Reactive domains
<code>showReactLog</code>	Reactive Log Visualizer

## Boilerplate

---

Functions that are required boilerplate in `ui.R` and `server.R`.

<code>shinyUI</code>	Create a Shiny UI handler
<code>shinyServer</code>	Define Server Functionality

## Running

---

Functions that are used to run or stop Shiny applications.

<code>runApp</code>	Run Shiny Application
<code>runExample</code>	Run Shiny Example Applications
<code>runUrl</code> ( <code>runGist</code> , <code>runGitHub</code> )	Run a Shiny application from a URL
<code>stopApp</code>	Stop the currently running Shiny app

## Extending Shiny

---

Functions that are intended to be called by third-party packages that extend Shiny.

<code>createWebDependency</code>	Create a web dependency
<code>addResourcePath</code>	Resource Publishing
<code>registerInputHandler</code>	Register an Input Handler
<code>removeInputHandler</code>	Deregister an Input Handler
<code>markRenderFunction</code>	Mark a function as a render function

## Utility functions

---

Miscellaneous utilities that may be useful to advanced users or when extending Shiny.

<code>validate</code> ( <code>need</code> )	Validate input values and other conditions
<code>session</code>	Session object
<code>exprToFunction</code>	Convert an expression to a function
<code>installExprFunction</code>	Install an expression as a function
<code>parseQueryString</code>	Parse a GET query string from a URL
<code>plotPNG</code>	Run a plotting function and save the output as a PNG
<code>repeatable</code>	Make a random number generator repeatable
<code>shinyDeprecated</code>	Print message for deprecated functions in Shiny
<code>serverInfo</code>	Collect information about the Shiny Server environment
<code>shiny-options</code>	Global options for Shiny

## Plot interaction

---

Functions related to interactive plots

<code>brushedPoints</code>	Find rows of data that are selected by a brush
<code>brushOpts</code>	Create an object representing brushing options
<code>clickOpts</code>	Create an object representing click options
<code>dblclickOpts</code>	Create an object representing double-click options
<code>hoverOpts</code>	Create an object representing hover options
<code>nearPoints</code>	Find rows of data that are near a click/hover/double-click

## Embedding

---

Functions that are intended for third-party packages that embed Shiny applications.

**shinyApp** Create a Shiny app object

(as. shiny. appobj ,  
as. shiny. appobj . character ,  
as. shiny. appobj . list ,  
as. shiny. appobj . shiny. appobj ,  
as. tags. shiny. appobj ,  
is. shiny. appobj ,  
print. shiny. appobj ,  
shinyAppDir)

**maskReactiveContext** Evaluate an expression without a reactive context

## Panel with absolute positioning

# 3.2 Panel with absolute positioning

**right** Horizontal distance from right edge of the parent, when the right or top edge of the parent is centered.

**bottom** Distance between the bottom of the panel, and the bottom of the page or parent container.

**width** Width of the panel.

**height** Height of the panel.

**draggable** `Boolean`, allows the user to move the panel by clicking and dragging.

**fixed** Positions the panel relative to the browser window and prevents it from being scrolled with the rest of the page.

**cursor** The type of cursor that should appear when the user moves over the panel. Use `"move"` for a north-south-east-west icon, `"default"` for the usual cursor arrow or `"inherit"` for the usual cursor behavior (including changing to an I-beam when the cursor is over text). The default is `"inherit"`, which is equivalent to `inherit(visibility: "move", "inherit")`.

**Value**  
An HTML element or list of elements.

**Description**  
Creates a panel whose contents are absolutely positioned.

**Details**  
The `absolutepanel` function creates a tag whose CSS position is set to `absolute` (or fixed if `fixed = true`). The way absolute positioning works in HTML is that absolute coordinates are specified relative to its nearest parent element whose position is not set to `static` (which is the default), and if no such parent is found, then relative to the page headers. If you're not sure what that means, just keep in mind that you may get strange results if you use `absolutepanel` from inside of certain types of panels.  
The `fixedpanel` function is the same as `absolutepanel` with `fixed = true`.  
The `position` (`top`, `left`, `right`, `bottom`) and `size` (`width`, `height`) parameters are all optional, but you should specify exactly two of `top`, `bottom`, and `height` and exactly two of `left`, `right`, and `width` for predictable results.  
Like most other distance parameters in HTML, the `position` and `size` parameters take a number (interpreted as pixels) or a valid CSS size string, such as `"100px"`, `100%`, `100px/2`, or `100%`.  
For ancient HTML reasons, to have the panel fill the page or parent you should specify a `top`, `left`, `right`, and `bottom` rather than the more obvious `width = 100%` and `height = 100%`.

### 3.3 Create a Bootstrap page

# Create a Bootstrap page

```
bootstrapPage(..., title = NULL, responsive = NULL, theme = NULL)
```

```
basicPage(...)
```

## Arguments

...	The contents of the document body.
title	The browser window title (defaults to the host URL of the page)
responsive	This option is deprecated; it is no longer optional with Bootstrap 3.
theme	Alternative Bootstrap stylesheet (normally a css file within the www directory, e.g. <code>www/bootstrap.css</code> )

## Value

A UI definition that can be passed to the [shinyUI](#) function.

## Description

Create a Shiny UI page that loads the CSS and JavaScript for [Bootstrap](#), and has no content in the page body (other than what you provide).

## Details

This function is primarily intended for users who are proficient in HTML/CSS, and know how to lay out pages in Bootstrap. Most applications should use [fluidPage](#) along with layout functions like [fluidRow](#) and [sidebarLayout](#).

## Note

The `basicPage` function is deprecated, you should use the [fluidPage](#) function instead.

## See also

[fluidPage](#), [fixedPage](#)

### 3.4 Create a column within a UI definition

# Create a column within a UI definition

```
column(width, ..., offset = 0)
```

## Arguments

`width` The grid width of the column (must be between 1 and 12)

`...` Elements to include within the column

`offset` The number of columns to offset this column from the end of the previous column.

## Value

A column that can be included within a `fluidRow` or `fixedRow`.

## Description

Create a column for use within a `fluidRow` or `fixedRow`

## Examples

```
fluidRow(  
  column(4,  
    sliderInput("obs", "Number of observations:",  
               min = 1, max = 1000, value = 500)  
  ),  
  column(8,  
    plotOutput("distPlot")  
  )  
)
```

```
<div class="row">  
  <div class="col-sm-4">  
    <div class="form-group shiny-input-container">  
      <label class="control-label" for="obs">Number of observations:</label>  
      <input class="js-range-slider" id="obs" data-min="1" data-max="1000" data-from="500" data-step="1" data-grid="true" data-grid-num="9.99" data-grid-snap="false" data-prettify-separator="," data-keyboard="true" data-keyboard-step="0.1001001001001"/>  
    </div>  
  </div>  
  <div class="col-sm-8">  
    <div id="distPlot" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>  
  </div>  
</div>
```



```
fluidRow(  
  column(width = 4,  
    "4"  
  ),  
  column(width = 3, offset = 2,  
    "3 offset 2"  
  )  
)  
  
<div class="row">  
  <div class="col-sm-4">4</div>  
  <div class="col-sm-3 col-sm-offset-2">3 offset 2</div>  
</div>
```

## See also

[fluidRow](#) , [fixedRow](#) .

## 3.5 Conditional Panel

# Conditional Panel

`conditionalPanel (condition, ...)`

## Arguments

`condition` A JavaScript expression that will be evaluated repeatedly to determine whether the panel should be displayed.

`...` Elements to include in the panel.

## Description

Creates a panel that is visible or not, depending on the value of a JavaScript expression. The JS expression is evaluated once at startup and whenever Shiny detects a relevant change in input/output.

## Details

In the JS expression, you can refer to `input` and `output` JavaScript objects that contain the current values of input and output. For example, if you have an input with an id of `foo`, then you can use `input.foo` to read its value. (Be sure not to modify the input/output objects, as this may cause unpredictable behavior.)

## Note

You are not recommended to use special JavaScript characters such as a period `.` in the input id's, but if you do use them anyway, for example, `inputId = "foo.bar"`, you will have to use `input["foo.bar"]` instead of `input.foo.bar` to read the input value.

## Examples

```
sidebarPanel(  
  selectInput(  
    "plotType", "Plot Type",  
    c(Scatter = "scatter",  
      Histogram = "hist")),  
  
  # Only show this panel if the plot type is a histogram  
  conditionalPanel(  
    condition = "input.plotType == 'hist'",  
    selectInput(  
      "breaks", "Breaks",  
      c("Sturges",  
        "Scott",  
        "Freedman-Diaconis",
```

```

    "[Custom]" = "custom")),

  # Only show this panel if Custom is selected
  conditionalPanel(
    condition = "input.breaks == 'custom'",
    sliderInput("breakCount", "Break Count", min=1, max=1000, value=10)
  )
)
)

<div class="col-sm-4">
  <form class="well">
    <div class="form-group shiny-input-container">
      <label class="control-label" for="plotType">Plot Type</label>
      <div>
        <select id="plotType"><option value="scatter" selected>Scatter</option>
<option value="hist">Histogram</option></select>
        <script type="application/json" data-for="plotType" data-nonempty="">{}</script>
      </div>
    </div>
    <div data-display-if="input.plotType == '#39;hist#39;'">
      <div class="form-group shiny-input-container">
        <label class="control-label" for="breaks">Breaks</label>
        <div>
          <select id="breaks"><option value="Sturges" selected>Sturges</option>
<option value="Scott">Scott</option>
<option value="Freedman-Diaconis">Freedman-Diaconis</option>
<option value="custom">[Custom]</option></select>
          <script type="application/json" data-for="breaks" data-nonempty="">{}</script>
        </div>
      </div>
    <div data-display-if="input.breaks == '#39;custom#39;'">
      <div class="form-group shiny-input-container">
        <label class="control-label" for="breakCount">Break Count</label>
        <input class="js-range-slider" id="breakCount" data-min="1" data-max="1000" data-from="10
" data-step="1" data-grid="true" data-grid-num="9.99" data-grid-snap="false" data-prettyfy-separat
or="," data-keyboard="true" data-keyboard-step="0.1001001001001"/>
      </div>
    </div>
  </div>
</form>
</div>

```

## 3.6 Create a page with a fixed layout

# Create a page with a fixed layout

```
fixedPage(..., title = NULL, responsive = NULL, theme = NULL)
```

```
fixedRow(...)
```

## Arguments

...	Elements to include within the container
title	The browser window title (defaults to the host URL of the page)
responsive	This option is deprecated; it is no longer optional with Bootstrap 3.
theme	Alternative Bootstrap stylesheet (normally a css file within the www directory). For example, to use the theme located at <code>www/bootstrap.css</code> you would use <code>theme = "bootstrap.css"</code> .

## Value

A UI definition that can be passed to the `shinyUI` function.

## Description

Functions for creating fixed page layouts. A fixed page layout consists of rows which in turn include columns. Rows exist for the purpose of making sure their elements appear on the same line (if the browser has adequate width). Columns exist for the purpose of defining how much horizontal space within a 12-unit wide grid it's elements should occupy. Fixed pages limit their width to 940 pixels on a typical display, and 724px or 1170px on smaller and larger displays respectively.

## Details

To create a fixed page use the `fixedPage` function and include instances of `fixedRow` and `column` within it. Note that unlike `fluidPage`, fixed pages cannot make use of higher-level layout functions like `sidebarLayout`, rather, all layout must be done with `fixedRow` and `column`.

## Note

See the [Shiny Application Layout Guide](#) for additional details on laying out fixed pages.

## Examples

```
shinyUI(fixedPage(  
  title = "Hello, Shiny!",  
  fixedRow(  
    column(width = 4,
```

```
    "4"
  ),
  column(width = 3, offset = 2,
    "3 offset 2"
  )
)
))

<div class="container">
  <div class="row">
    <div class="col-sm-4">4</div>
    <div class="col-sm-3 col-sm-offset-2">3 offset 2</div>
  </div>
</div>
```

## See also

[column](#)

### 3.7 Create a page with fluid layout

# Create a page with fluid layout

```
fluidPage(..., title = NULL, responsive = NULL, theme = NULL)
```

```
fluidRow(...)
```

## Arguments

...	Elements to include within the page
title	The browser window title (defaults to the host URL of the page). Can also be set as a side effect of the <code>titlePanel</code> function.
responsive	This option is deprecated; it is no longer optional with Bootstrap 3.
theme	Alternative Bootstrap stylesheet (normally a css file within the www directory). For example, to use the theme located at <code>www/bootstrap.css</code> you would use <code>theme = "bootstrap.css"</code> .

## Value

A UI definition that can be passed to the `shinyUI` function.

## Description

Functions for creating fluid page layouts. A fluid page layout consists of rows which in turn include columns. Rows exist for the purpose of making sure their elements appear on the same line (if the browser has adequate width). Columns exist for the purpose of defining how much horizontal space within a 12-unit wide grid it's elements should occupy. Fluid pages scale their components in realtime to fill all available browser width.

## Details

To create a fluid page use the `fluidPage` function and include instances of `fluidRow` and `column` within it. As an alternative to low-level row and column functions you can also use higher-level layout functions like `sidebarLayout`.

## Note

See the [Shiny-Application-Layout-Guide](#) for additional details on laying out fluid pages.

## Examples

```
shinyUI(fluidPage(  
  
  # Application title  
  titlePanel("Hello Shiny!"),
```

```

sidebarLayout(

  # Sidebar with a slider input
  sidebarPanel(
    sliderInput("obs",
               "Number of observations:",
               min = 0,
               max = 1000,
               value = 500)
  ),

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

<div class="container-fluid">
  <h2>Hello Shiny!</h2>
  <div class="row">
    <div class="col-sm-4">
      <form class="well">
        <div class="form-group shiny-input-container">
          <label class="control-label" for="obs">Number of observations:</label>
          <input class="js-range-slider" id="obs" data-min="0" data-max="1000" data-from="500" data
- step="1" data-grid="true" data-grid-num="10" data-grid-snap="false" data-prettify-separator="," data
ata-keyboard="true" data-keyboard-step="0.1"/>
        </div>
      </form>
    </div>
    <div class="col-sm-8">
      <div id="distPlot" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>
    </div>
  </div>
</div>

shinyUI(fluidPage(
  title = "Hello Shiny!",
  fluidRow(
    column(width = 4,
           "4"
    ),
    column(width = 3, offset = 2,
           "3 offset 2"
    )
  )
))

<div class="container-fluid">
  <div class="row">
    <div class="col-sm-4">4</div>
    <div class="col-sm-3 col-sm-offset-2">3 offset 2</div>
  </div>
</div>

```

## See also

`column`, `sidebarLayout`

Shiny is an [RStudio](#) project. © 2014 RStudio, Inc.



## 3.8 Create a header panel

# Create a header panel

```
headerPanel(title, windowTitle = title)
```

## Arguments

`title` An application title to display

`windowTitle` The title that should be displayed by the browser window. Useful if `title` is not a string.

## Value

A `headerPanel` that can be passed to [pageWithSidebar](#)

## Description

Create a header panel containing an application title.

## Examples

```
headerPanel("Hello Shiny!")
```

```
<div class="col-sm-12">  
  <h1>Hello Shiny!</h1>  
</div>
```

## 3.9 Create a help text element

# Create a help text element

```
helpText(...)
```

## Arguments

... One or more help text strings (or other inline HTML elements)

## Value

A help text element that can be added to a UI definition.

## Description

Create help text which can be added to an input form to provide additional explanation or context.

## Examples

```
helpText("Note: while the data view will show only",  
         "the specified number of observations, the",  
         "summary will be based on the full dataset.")
```

```
<span class="help-block">  
  Note: while the data view will show only  
  the specified number of observations, the  
  summary will be based on the full dataset.  
</span>
```

### 3.10 Create an icon

# Create an icon

```
icon(name, class = NULL, lib = "font-awesome")
```

## Arguments

- name** Name of icon. Icons are drawn from the [Font Awesome](#) and [Glyphicons](#) libraries. Note that the "fa-" and "glyphicon-" prefixes should not be used in icon names (i.e. the "fa-calendar" icon should be referred to as "calendar")
- class** Additional classes to customize the style of the icon (see the [usage examples](#) for details on supported styles).
- lib** Icon library to use ("font-awesome" or "glyphicon")

## Value

An icon element

## Description

Create an icon for use within a page. Icons can appear on their own, inside of a button, or as an icon for a `tabpanel` within a `navbarPage`.

## Examples

```
icon("calendar")           # standard icon

<i class="fa fa-calendar"></i>

icon("calendar", "fa-3x")   # 3x normal size

<i class="fa fa-calendar fa-3x"></i>

icon("cog", lib = "glyphicon") # From glyphicon library

<i class="glyphicon glyphicon-cog"></i>

# add an icon to a submit button
submitButton("Update View", icon = icon("refresh"))

<div>
  <button type="submit" class="btn btn-primary">
    <i class="fa fa-refresh"></i>
```

```

    update view
  </button>
</div>

```

```

shinyUI(navbarPage("App Title",
  tabPanel("Plot", icon = icon("bar-chart-o")),
  tabPanel("Summary", icon = icon("list-alt")),
  tabPanel("Table", icon = icon("table")))
))

```

```

<nav class="navbar navbar-default navbar-static-top" role="navigation">
  <div class="container">
    <div class="navbar-header">
      <span class="navbar-brand">App Title</span>
    </div>
    <ul class="nav navbar-nav">
      <li class="active">
        <a href="#tab-1065-1" data-toggle="tab" data-value="Plot">
          <i class=" fa fa-bar-chart-o fa-fw"></i>
          Plot
        </a>
      </li>
      <li>
        <a href="#tab-1065-2" data-toggle="tab" data-value="Summary">
          <i class=" fa fa-list-alt fa-fw"></i>
          Summary
        </a>
      </li>
      <li>
        <a href="#tab-1065-3" data-toggle="tab" data-value="Table">
          <i class=" fa fa-table fa-fw"></i>
          Table
        </a>
      </li>
    </ul>
  </div>
</nav>
<div class="container-fluid">
  <div class="tab-content">
    <div class="tab-pane active" data-value="Plot" data-icon-class="fa fa-bar-chart-o" id="tab-1065-1"></div>
    <div class="tab-pane" data-value="Summary" data-icon-class="fa fa-list-alt" id="tab-1065-2"></div>
    <div class="tab-pane" data-value="Table" data-icon-class="fa fa-table" id="tab-1065-3"></div>
  </div>
</div>

```

## See also

For lists of available icons, see <http://fontawesome.io/icons/> and <http://getbootstrap.com/components/#glyphicons>.

### 3.11 Create a main panel

# Create a main panel

```
mainPanel(..., width = 8)
```

## Arguments

... Output elements to include in the main panel

width The width of the main panel. For fluid layouts this is out of 12 total units; for fixed layouts it is out of whatever the width of the main panel's parent column is.

## Value

A main panel that can be passed to `sidebarLayout`.

## Description

Create a main panel containing output elements that can in turn be passed to `sidebarLayout`.

## Examples

```
# Show the caption and plot of the requested variable against mpg
mainPanel(
  h3(textOutput("caption")),
  plotOutput("mpgPlot")
)
```

```
<div class="col-sm-8">
  <h3>
    <div id="caption" class="shiny-text-output"></div>
  </h3>
  <div id="mpgPlot" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>
</div>
```

### 3.12 Create a page with a top level

# Create a page with a top level navigation bar

```
navbarPage(title, ..., id = NULL, position = c("static-top", "fixed-top", "fixed-bottom"),
  header = NULL, footer = NULL, inverse = FALSE, collapsible = FALSE, collapsable,
  fluid = TRUE, responsive = NULL, theme = NULL, windowTitle = title)
```

```
navbarMenu(title, ..., icon = NULL)
```

## Arguments

title	The title to display in the navbar
...	<code>tabpanel</code> elements to include in the page
id	If provided, you can use <code>input\$ id</code> in your server logic to determine which of the current tabs is active. The value will correspond to the <code>value</code> argument that is passed to <code>tabpanel</code> .
position	Determines whether the navbar should be displayed at the top of the page with normal scrolling behavior ( <code>"static-top"</code> ), pinned at the top ( <code>"fixed-top"</code> ), or pinned at the bottom ( <code>"fixed-bottom"</code> ). Note that using <code>"fixed-top"</code> or <code>"fixed-bottom"</code> will cause the navbar to overlay your body content, unless you add padding, e.g.: <code>tags\$style(type="text/css", "body {padding-top: 70px;}")</code>
header	Tag or list of tags to display as a common header above all <code>tabPanels</code> .
footer	Tag or list of tags to display as a common footer below all <code>tabPanels</code>
inverse	<code>TRUE</code> to use a dark background and light text for the navigation bar
collapsible	<code>TRUE</code> to automatically collapse the navigation elements into a menu when the width of the browser is less than 940 pixels (useful for viewing on smaller touchscreen device)
collapsable	Deprecated; use <code>collapsible</code> instead.
fluid	<code>TRUE</code> to use a fluid layout. <code>FALSE</code> to use a fixed layout.
responsive	This option is deprecated; it is no longer optional with Bootstrap 3.
theme	Alternative Bootstrap stylesheet (normally a <code>css</code> file within the <code>www</code> directory). For example, to use the theme located at <code>www/bootstrap.css</code> you would use <code>theme = "bootstrap.css"</code> .
windowTitle	The title that should be displayed by the browser window. Useful if <code>title</code> is not a string.
icon	Optional icon to appear on a <code>navbarMenu</code> tab.

## Value

A UI definition that can be passed to the `shinyUI` function.

## Description

Create a page that contains a top level navigation bar that can be used to toggle a set of `tabPanel` elements.

## Details

The `navbarMenu` function can be used to create an embedded menu within the navbar that in turns includes additional `tabPanels` (see example below).

## Examples

```
shinyUI(navbarPage("App Title",
  tabPanel("Plot"),
  tabPanel("Summary"),
  tabPanel("Table")
))
```

```
<nav class="navbar navbar-default navbar-static-top" role="navigation">
  <div class="container">
    <div class="navbar-header">
      <span class="navbar-brand">App Title</span>
    </div>
    <ul class="nav navbar-nav">
      <li class="active">
        <a href="#tab-3367-1" data-toggle="tab" data-value="Plot">Plot</a>
      </li>
      <li>
        <a href="#tab-3367-2" data-toggle="tab" data-value="Summary">Summary</a>
      </li>
      <li>
        <a href="#tab-3367-3" data-toggle="tab" data-value="Table">Table</a>
      </li>
    </ul>
  </div>
</nav>
<div class="container-fluid">
  <div class="tab-content">
    <div class="tab-pane active" data-value="Plot" id="tab-3367-1"></div>
    <div class="tab-pane" data-value="Summary" id="tab-3367-2"></div>
    <div class="tab-pane" data-value="Table" id="tab-3367-3"></div>
  </div>
</div>
```

```
shinyUI(navbarPage("App Title",
  tabPanel("Plot"),
  navbarMenu("More",
    tabPanel("Summary"),
    tabPanel("Table")
  )
))
```

```
<nav class="navbar navbar-default navbar-static-top" role="navigation">
  <div class="container">
    <div class="navbar-header">
      <span class="navbar-brand">App Title</span>
```

```
</div>
<ul class="nav navbar-nav">
  <li class="active">
    <a href="#tab-1130-1" data-toggle="tab" data-value="Plot">Plot</a>
  </li>
  <li class="dropdown">
    <a href="#" class="dropdown-toggle" data-toggle="dropdown">
      More
      <b class="caret"></b>
    </a>
    <ul class="dropdown-menu">
      <li>
        <a href="#tab-2344-1" data-toggle="tab" data-value="Summary">Summary</a>
      </li>
      <li>
        <a href="#tab-2344-2" data-toggle="tab" data-value="Table">Table</a>
      </li>
    </ul>
  </li>
</ul>
</div>
</nav>
<div class="container-fluid">
  <div class="tab-content">
    <div class="tab-pane active" data-value="Plot" id="tab-1130-1"></div>
    <div class="tab-pane" data-value="Summary" id="tab-2344-1"></div>
    <div class="tab-pane" data-value="Table" id="tab-2344-2"></div>
  </div>
</div>
```

## See also

[tabPanel](#) , [tabsetPanel](#)



### 3.13 Create a navigation list panel

# Create a navigation list panel

```
navlistPanel(..., id = NULL, selected = NULL, well = TRUE, fluid = TRUE, widths = c(4, 8))
```

## Arguments

- ... `tabPanel` elements to include in the navlist
- id If provided, you can use `input$id` in your server logic to determine which of the current navlist items is active. The value will correspond to the `value` argument that is passed to `tabPanel`.
- selected The `value` (or, if none was supplied, the `title`) of the navigation item that should be selected by default. If `NULL`, the first navigation will be selected.
- well `TRUE` to place a well (gray rounded rectangle) around the navigation list.
- fluid `TRUE` to use fluid layout; `FALSE` to use fixed layout.
- widths Column widths of the navigation list and tabset content areas respectively.

## Description

Create a navigation list panel that provides a list of links on the left which navigate to a set of `tabPanels` displayed to the right.

## Details

You can include headers within the `navlistPanel` by including plain text elements in the list. Versions of Shiny before 0.11 supported separators with "-----", but as of 0.11, separators were no longer supported. This is because version 0.11 switched to Bootstrap 3, which doesn't support separators.

## Examples

```
shinyUI(fluidPage(  
  
  titlePanel("Application Title"),  
  
  navlistPanel(  
    "Header",  
    tabPanel("First"),  
    tabPanel("Second"),  
    tabPanel("Third")  
  )  
))  
  
<div class="container-fluid">
```

```
<h2>Application title</h2>
<div class="row">
  <div class="col-sm-4 well">
    <ul class="nav nav-pills nav-stacked">
      <li class="navbar-brand">Header</li>
      <li class="active">
        <a href="#tab-3608-1" data-toggle="tab" data-value="First">First</a>
      </li>
      <li>
        <a href="#tab-3608-2" data-toggle="tab" data-value="Second">Second</a>
      </li>
      <li>
        <a href="#tab-3608-3" data-toggle="tab" data-value="Third">Third</a>
      </li>
    </ul>
  </div>
  <div class="col-sm-8">
    <div class="tab-content">
      <div class="tab-pane active" data-value="First" id="tab-3608-1"></div>
      <div class="tab-pane" data-value="Second" id="tab-3608-2"></div>
      <div class="tab-pane" data-value="Third" id="tab-3608-3"></div>
    </div>
  </div>
</div>
</div>
```

### 3.14 Create a page with a sidebar

# Create a page with a sidebar

```
pageWithSidebar(headerPanel, sidebarPanel, mainPanel)
```

## Arguments

`headerPanel` The `headerPanel` with the application title

`sidebarPanel` The `sidebarPanel` containing input controls

`mainPanel` The `mainPanel` containing outputs

## Value

A UI definition that can be passed to the `shinyUI` function

## Description

Create a Shiny UI that contains a header with the application title, a sidebar for input controls, and a main area for output.

## Note

This function is deprecated. You should use `fluidPage` along with `sidebarLayout` to implement a page with a sidebar.

## Examples

```
# Define UI
shinyUI(pageWithSidebar(

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

  # Sidebar with a slider input
  sidebarPanel(
    sliderInput("obs",
               "Number of observations:",
               min = 0,
               max = 1000,
               value = 500)
  ),

  # Show a plot of the generated distribution
  mainPanel(
```

```
plotOutput("distPlot")
)
))

<div class="container-fluid">
  <div class="row">
    <div class="col-sm-12">
      <h1>Hello Shiny!</h1>
    </div>
  </div>
  <div class="row">
    <div class="col-sm-4">
      <form class="well">
        <div class="form-group shiny-input-container">
          <label class="control-label" for="obs">Number of observations:</label>
          <input class="js-range-slider" id="obs" data-min="0" data-max="1000" data-from="500" data-
- step="1" data-grid="true" data-grid-num="10" data-grid-snap="false" data-prettify-separator="," data-
ata-keyboard="true" data-keyboard-step="0.1"/>
        </div>
      </form>
    </div>
    <div class="col-sm-8">
      <div id="distPlot" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>
    </div>
  </div>
</div>
```

### 3.15 Layout a sidebar and main area

# Layout a sidebar and main area

```
sidebarLayout(sidebarPanel, mainPanel, position = c("left", "right"), fluid = TRUE)
```

## Arguments

`sidebarPanel` The `sidebarPanel` containing input controls

`mainPanel` The `mainPanel` containing outputs

`position` The position of the sidebar relative to the main area ("left" or "right")

`fluid` `TRUE` to use fluid layout; `FALSE` to use fixed layout.

## Description

Create a layout with a sidebar and main area. The sidebar is displayed with a distinct background color and typically contains input controls. The main area occupies 2/3 of the horizontal width and typically contains outputs.

## Examples

```
# Define UI
shinyUI(fluidPage(

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

  sidebarLayout(

    # Sidebar with a slider input
    sidebarPanel(
      sliderInput("obs",
                  "Number of observations:",
                  min = 0,
                  max = 1000,
                  value = 500)
    ),

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

<div class="container-fluid">
```

```
<n2>HELLO Shiny!</n2>
<div class="row">
  <div class="col-sm-4">
    <form class="well">
      <div class="form-group shiny-input-container">
        <label class="control-label" for="obs">Number of observations:</label>
        <input class="js-range-slider" id="obs" data-min="0" data-max="1000" data-from="500" data
-step="1" data-grid="true" data-grid-num="10" data-grid-snap="false" data-prettyfy-separator="," d
ata-keyboard="true" data-keyboard-step="0.1"/>
      </div>
    </form>
  </div>
  <div class="col-sm-8">
    <div id="distPlot" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>
  </div>
</div>
```

### 3.16 Create a sidebar panel

# Create a sidebar panel

```
sidebarPanel(..., width = 4)
```

## Arguments

... UI elements to include on the sidebar

width The width of the sidebar. For fluid layouts this is out of 12 total units; for fixed layouts it is out of whatever the width of the sidebar's parent column is.

## Value

A sidebar that can be passed to `sidebarLayout`

## Description

Create a sidebar panel containing input controls that can in turn be passed to `sidebarLayout`.

## Examples

```
# Sidebar with controls to select a dataset and specify
# the number of observations to view
```

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

  numericInput("obs", "Observations:", 10)
)
```

```
<div class="col-sm-4">
  <form class="well">
    <div class="form-group shiny-input-container">
      <label class="control-label" for="dataset">Choose a dataset:</label>
      <div>
        <select id="dataset"><option value="rock" selected>rock</option>
<option value="pressure">pressure</option>
<option value="cars">cars</option></select>
        <script type="application/json" data-for="dataset" data-nonempty="">{}</script>
      </div>
    </div>
    <div class="form-group shiny-input-container">
      <label for="obs">Observations:</label>
      <input id="obs" type="number" class="form-control" value="10"/>
    </div>
```

```
</form>
```

```
</div>
```

Shiny is an RStudio project. © 2014 RStudio, Inc.



### 3.17 Create a tab panel

# Create a tab panel

```
tabPanel(title, ..., value = title, icon = NULL)
```

## Arguments

- `title`    Display title for tab
- `...`     UI elements to include within the tab
- `value`    The value that should be sent when `tabsetPanel` reports that this tab is selected. If omitted and `tabsetPanel` has an `id`, then the title will be used..
- `icon`     Optional icon to appear on the tab. This attribute is only valid when using a `tabPanel` within a `navbarPage`.

## Value

A tab that can be passed to `tabsetPanel`

## Description

Create a tab panel that can be included within a `tabsetPanel`.

## Examples

```
# Show a tabset that includes a plot, summary, and
# table view of the generated distribution
mainPanel(
  tabsetPanel(
    tabPanel("Plot", plotOutput("plot")),
    tabPanel("Summary", verbatimTextOutput("summary")),
    tabPanel("Table", tableOutput("table"))
  )
)

<div class="col-sm-8">
  <div class="tabbable tabs-above">
    <ul class="nav nav-tabs">
      <li class="active">
        <a href="#tab-5787-1" data-toggle="tab" data-value="Plot">Plot</a>
      </li>
      <li>
        <a href="#tab-5787-2" data-toggle="tab" data-value="Summary">Summary</a>
      </li>
    </ul>
  </div>
</div>
```

```
<a href="#tab-5787-3" data-toggle="tab" data-value="Table">Table</a>
</li>
</ul>
<div class="tab-content">
  <div class="tab-pane active" data-value="Plot" id="tab-5787-1">
    <div id="plot" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>
  </div>
  <div class="tab-pane" data-value="Summary" id="tab-5787-2">
    <pre id="summary" class="shiny-text-output"></pre>
  </div>
  <div class="tab-pane" data-value="Table" id="tab-5787-3">
    <div id="table" class="shiny-html-output"></div>
  </div>
</div>
</div>
</div>
```

## See also

[tabsetPanel](#)

### 3.18 Create a tabset panel

# Create a tabset panel

```
tabsetPanel(..., id = NULL, selected = NULL, type = c("tabs", "pills"),
  position = c("above", "below", "left", "right"))
```

## Arguments

- ... `tabPanel` elements to include in the tabset
- `id` If provided, you can use `input$ id` in your server logic to determine which of the current tabs is active. The value will correspond to the `value` argument that is passed to `tabPanel` .
- `selected` The `value` (or, if none was supplied, the `title` ) of the tab that should be selected by default. If `NULL` , the first tab will be selected.
- `type` Use "tabs" for the standard look; Use "pills" for a more plain look where tabs are selected using a background fill color.
- `position` The position of the tabs relative to the content. Valid values are "above", "below", "left", and "right" (defaults to "above"). Note that the `position` argument is not valid when `type` is "pill".

## Value

A tabset that can be passed to `mainPanel`

## Description

Create a tabset that contains `tabPanel` elements. Tabsets are useful for dividing output into multiple independently viewable sections.

## Examples

```
# Show a tabset that includes a plot, summary, and
# table view of the generated distribution
mainPanel(
  tabsetPanel(
    tabPanel("Plot", plotOutput("plot")),
    tabPanel("Summary", verbatimTextOutput("summary")),
    tabPanel("Table", tableOutput("table"))
  )
)
```

```
<div class="col-sm-8">
  <div class="tabbable tabs-above">
    <ul class="nav nav-tabs">
      <li class="active">
```

```
    <a href="#tab-3579-1" data-toggle="tab" data-value="Plot">Plot</a>
  </li>
  <li>
    <a href="#tab-3579-2" data-toggle="tab" data-value="Summary">Summary</a>
  </li>
  <li>
    <a href="#tab-3579-3" data-toggle="tab" data-value="Table">Table</a>
  </li>
</ul>
<div class="tab-content">
  <div class="tab-pane active" data-value="Plot" id="tab-3579-1">
    <div id="plot" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>
  </div>
  <div class="tab-pane" data-value="Summary" id="tab-3579-2">
    <pre id="summary" class="shiny-text-output"></pre>
  </div>
  <div class="tab-pane" data-value="Table" id="tab-3579-3">
    <div id="table" class="shiny-html-output"></div>
  </div>
</div>
</div>
</div>
```

## See also

[tabPanel](#) , [updateTabsetPanel](#)

### 3.19 Create a panel containing an

# Create a panel containing an application title.

```
titlePanel(title, windowTitle = title)
```

## Arguments

`title` An application title to display

`windowTitle` The title that should be displayed by the browser window.

## Description

Create a panel containing an application title.

## Details

Calling this function has the side effect of including a `title` tag within the head. You can also specify a page title explicitly using the `title` parameter of the top-level page function.

## Examples

```
titlePanel("Hello Shiny!")
```

```
<h2>Hello Shiny!</h2>
```

## 3.20 Input panel

# Input panel

`inputPanel (...)`

## Arguments

... Input controls or other HTML elements.

## Description

A `flowLayout` with a grey border and light grey background, suitable for wrapping inputs.

## 3.21 Flow layout

# Flow layout

```
flowLayout(..., cellArgs = list())
```

## Arguments

... Unnamed arguments will become child elements of the layout. Named arguments will become HTML attributes on the outermost tag.

cellArgs Any additional attributes that should be used for each cell of the layout.

## Description

Lays out elements in a left-to-right, top-to-bottom arrangement. The elements on a given row will be top-aligned with each other. This layout will not work well with elements that have a percentage-based width (e.g. `plotOutput` at its default setting of `width = "100%"`).

## Examples

```
flowLayout(
  numericInput("rows", "How many rows?", 5),
  selectInput("letter", "Which letter?", LETTERS),
  sliderInput("value", "What value?", 0, 100, 50)
)
```

```
<div class="shiny-flow-layout">
  <div>
    <div class="form-group shiny-input-container">
      <label for="rows">How many rows?</label>
      <input id="rows" type="number" class="form-control" value="5"/>
    </div>
  </div>
  <div>
    <div class="form-group shiny-input-container">
      <label class="control-label" for="letter">Which letter?</label>
      <div>
        <select id="letter"><option value="A" selected>A</option>
<option value="B">B</option>
<option value="C">C</option>
<option value="D">D</option>
<option value="E">E</option>
<option value="F">F</option>
<option value="G">G</option>
<option value="H">H</option>
<option value="I">I</option>
```

```
<option value="J">J</option>
<option value="K">K</option>
<option value="L">L</option>
<option value="M">M</option>
<option value="N">N</option>
<option value="O">O</option>
<option value="P">P</option>
<option value="Q">Q</option>
<option value="R">R</option>
<option value="S">S</option>
<option value="T">T</option>
<option value="U">U</option>
<option value="V">V</option>
<option value="W">W</option>
<option value="X">X</option>
<option value="Y">Y</option>
<option value="Z">Z</option></select>
  <script type="application/json" data-for="letter" data-nonempty="">{}</script>
</div>
</div>
</div>
<div>
  <div class="form-group shiny-input-container">
    <label class="control-label" for="value">What value?</label>
    <input class="js-range-slider" id="value" data-min="0" data-max="100" data-from="50" data-ste
p="1" data-grid="true" data-grid-num="10" data-grid-snap="false" data-prettyfy-separator="," data-
keyboard="true" data-keyboard-step="1"/>
  </div>
</div>
</div>
```

## See also

[vertical Layout](#)



## 3.22 Split layout

# Split layout

```
splitLayout(..., cellWidths = NULL, cellArgs = list())
```

## Arguments

- ... Unnamed arguments will become child elements of the layout. Named arguments will become HTML attributes on the outermost tag.
- cellWidths Character or numeric vector indicating the widths of the individual cells. Recycling will be used if needed. Character values will be interpreted as CSS lengths (see `validateCssUnit`), numeric values as pixels.
- cellArgs Any additional attributes that should be used for each cell of the layout.

## Description

Lays out elements horizontally, dividing the available horizontal space into equal parts (by default).

## Examples

```
# Equal sizing
```

```
splitLayout(  
  plotOutput("plot1"),  
  plotOutput("plot2")  
)
```

```
<div class="shiny-split-layout">  
  <div style="width: 50.000%;">  
    <div id="plot1" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>  
  </div>  
  <div style="width: 50.000%;">  
    <div id="plot2" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>  
  </div>  
</div>
```

```
# Custom widths
```

```
splitLayout(cellWidths = c("25%", "75%"),  
  plotOutput("plot1"),  
  plotOutput("plot2")  
)
```

```
<div class="shiny-split-layout">  
  <div style="width: 25%;">
```

*Split layout*

```

  <div id="plot1" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>
</div>
<div style="width: 75%;">
  <div id="plot2" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>
</div>
</div>

```

```

# All cells at 300 pixels wide, with cell padding
# and a border around everything

```

```

splitLayout(
  style = "border: 1px solid silver;",
  cellWidths = 300,
  cellArgs = list(style = "padding: 6px"),
  plotOutput("plot1"),
  plotOutput("plot2"),
  plotOutput("plot3")
)

<div class="shiny-split-layout" style="border: 1px solid silver;">
  <div style="width: 300px; padding: 6px">
    <div id="plot1" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>
  </div>
  <div style="width: 300px; padding: 6px">
    <div id="plot2" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>
  </div>
  <div style="width: 300px; padding: 6px">
    <div id="plot3" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>
  </div>
</div>

```

### 3.23 Lay out UI elements vertically

# Lay out UI elements vertically

```
verticalLayout(..., fluid = TRUE)
```

## Arguments

... Elements to include within the container

fluid `TRUE` to use fluid layout; `FALSE` to use fixed layout.

## Description

Create a container that includes one or more rows of content (each element passed to the container will appear on it's own line in the UI)

## Examples

```
shinyUI(fluidPage(  
  verticalLayout(  
    a(href="http://example.com/link1", "Link One"),  
    a(href="http://example.com/link2", "Link Two"),  
    a(href="http://example.com/link3", "Link Three")  
  )  
))
```

```
<div class="container-fluid">  
  <div class="row">  
    <div class="col-sm-12">  
      <a href="http://example.com/link1">Link One</a>  
    </div>  
  </div>  
  <div class="row">  
    <div class="col-sm-12">  
      <a href="http://example.com/link2">Link Two</a>  
    </div>  
  </div>  
  <div class="row">  
    <div class="col-sm-12">  
      <a href="http://example.com/link3">Link Three</a>  
    </div>  
  </div>  
</div>
```

## See also

## 3.24 Create a well panel

# Create a well panel

`wellPanel (...)`

## Arguments

... UI elements to include inside the panel.

## Value

The newly created panel.

## Description

Creates a panel with a slightly inset border and grey background. Equivalent to Bootstrap's `well` CSS class.

### 3.25 Load the MathJax library and typeset

# Load the MathJax library and typeset math expressions

```
withMathJax(...)
```

## Arguments

... any HTML elements to apply MathJax to

## Description

This function adds MathJax to the page and typeset the math expressions (if found) in the content . . . . It only needs to be called once in an app unless the content is rendered *after* the page is loaded, e.g. via `renderUI` , in which case we have to call it explicitly every time we write math expressions to the output.

## Examples

```
withMathJax(helpText("Some math here  $\alpha + \beta$ "))
```

```
<span class="help-block">Some math here  $\alpha + \beta$ </span>  
<script>MathJax.Hub.Queue(["Typeset", MathJax.Hub]);</script>
```

```
# now we can just write "static" content without withMathJax()  
div("more math here  $\sqrt{2}$ ")
```

```
<div>more math here  $\sqrt{2}$ </div>
```

## 3.26 Action button/link

# Action button/link

```
actionButton(inputId, label, icon = NULL, ...)
```

```
actionLink(inputId, label, icon = NULL, ...)
```

## Arguments

`inputId` Specifies the input slot that will be used to access the value.

`label` The contents of the button or link--usually a text label, but you could also use any other HTML, like an image.

`icon` An optional `icon` to appear on the button.

`...` Named attributes to be applied to the button or link.

## Description

Creates an action button or link whose value is initially zero, and increments by one each time it is pressed.

## Examples

```
## <strong>Not run</strong>:  
# # In server.R  
# output$distPlot <- renderPlot({  
#   # Take a dependency on input$goButton  
#   input$goButton  
#  
#   # Use isolate() to avoid dependency on input$obs  
#   dist <- isolate(rnorm(input$obs))  
#   hist(dist)  
# })  
#  
# # In ui.R  
# actionButton("goButton", "Go!")  
# ## <strong>End(Not run)</strong>
```

## See also

`observeEvent` and `eventReactive` Other input.elements: `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `fileInput`; `numericInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `submitButton`; `textInput`

## 3.27 Checkbox Group Input Control

# Checkbox Group Input Control

```
checkboxGroupInput(inputId, label, choices, selected = NULL, inline = FALSE)
```

## Arguments

- inputId** The `input` slot that will be used to access the value.
- label** Display label for the control, or `NULL` for no label.
- choices** List of values to show checkboxes for. If elements of the list are named then that name rather than the value is displayed to the user.
- selected** The values that should be initially selected, if any.
- inline** If `TRUE`, render the choices inline (i.e. horizontally)

## Value

A list of HTML elements that can be added to a UI definition.

## Description

Create a group of checkboxes that can be used to toggle multiple choices independently. The server will receive the input as a character vector of the selected values.

## Examples

```
checkboxGroupInput("variable", "Variable:",  
                c("Cylinders" = "cyl",  
                  "Transmission" = "am",  
                  "Gears" = "gear"))
```

```
<div id="variable" class="form-group shiny-input-checkboxgroup shiny-input-container">  
  <label class="control-label" for="variable">Variable:</label>  
  <div class="shiny-options-group">  
    <div class="checkbox">  
      <label>  
        <input type="checkbox" name="variable" value="cyl"/>  
        <span>Cylinders</span>  
      </label>  
    </div>  
    <div class="checkbox">  
      <label>  
        <input type="checkbox" name="variable" value="am"/>  
        <span>Transmission</span>  
      </label>  
    </div>  
  </div>  
</div>
```

```
</label>
</div>
<div class="checkbox">
  <label>
    <input type="checkbox" name="variable" value="gear"/>
    <span>Gears</span>
  </label>
</div>
</div>
</div>
```

## See also

[checkboxInput](#) , [updateCheckboxGroupInput](#) Other input.elements: [actionButton](#) , [actionLink](#) ; [animationOptions](#) , [sliderInput](#) ; [checkboxInput](#) ; [dateInput](#) ; [dateRangeInput](#) ; [fileInput](#) ; [numericInput](#) ; [passwordInput](#) ; [radioButtons](#) ; [selectInput](#) , [selectizeInput](#) ; [submitButton](#) ; [textInput](#)



## 3.28 Checkbox Input Control

# Checkbox Input Control

```
checkboxInput(inputId, label, value = FALSE)
```

## Arguments

`inputId` The `input` slot that will be used to access the value.

`label` Display label for the control, or `NULL` for no label.

`value` Initial value (`TRUE` or `FALSE`).

## Value

A checkbox control that can be added to a UI definition.

## Description

Create a checkbox that can be used to specify logical values.

## Examples

```
checkboxInput("outliers", "Show outliers", FALSE)
```

```
<div class="form-group shiny-input-container">  
  <div class="checkbox">  
    <label>  
      <input id="outliers" type="checkbox"/>  
      <span>Show outliers</span>  
    </label>  
  </div>  
</div>
```

## See also

`checkboxGroupInput` , `updateCheckboxInput` Other input elements: `actionButton` , `actionLink` ;  
`animationOptions` , `sliderInput` ; `checkboxGroupInput` ; `dateInput` ; `dateRangeInput` ; `fileInput` ;  
`numericInput` ; `passwordInput` ; `radioButtons` ; `selectInput` , `selectizeInput` ; `submitButton` ;  
`textInput`

## 3.29 Create date input

# Create date input

```
dateInput(inputId, label, value = NULL, min = NULL, max = NULL, format = "yyyy-mm-dd",  
  startview = "month", weekstart = 0, language = "en")
```

## Arguments

<code>inputId</code>	The <code>input</code> slot that will be used to access the value.
<code>label</code>	Display label for the control, or <code>NULL</code> for no label.
<code>value</code>	The starting date. Either a <code>Date</code> object, or a string in <code>yyyy-mm-dd</code> format. If <code>NULL</code> (the default), will use the current date in the client's time zone.
<code>min</code>	The minimum allowed date. Either a <code>Date</code> object, or a string in <code>yyyy-mm-dd</code> format.
<code>max</code>	The maximum allowed date. Either a <code>Date</code> object, or a string in <code>yyyy-mm-dd</code> format.
<code>format</code>	The format of the date to display in the browser. Defaults to <code>"yyyy-mm-dd"</code> .
<code>startview</code>	The date range shown when the input object is first clicked. Can be <code>"month"</code> (the default), <code>"year"</code> , or <code>"decade"</code> .
<code>weekstart</code>	Which day is the start of the week. Should be an integer from 0 (Sunday) to 6 (Saturday).
<code>language</code>	The language used for month and day names. Default is <code>"en"</code> . Other valid values include <code>"bg"</code> , <code>"ca"</code> , <code>"cs"</code> , <code>"da"</code> , <code>"de"</code> , <code>"el"</code> , <code>"es"</code> , <code>"fi"</code> , <code>"fr"</code> , <code>"he"</code> , <code>"hr"</code> , <code>"hu"</code> , <code>"id"</code> , <code>"is"</code> , <code>"it"</code> , <code>"ja"</code> , <code>"kr"</code> , <code>"lt"</code> , <code>"lv"</code> , <code>"ms"</code> , <code>"nb"</code> , <code>"nl"</code> , <code>"pl"</code> , <code>"pt"</code> , <code>"pt-BR"</code> , <code>"ro"</code> , <code>"rs"</code> , <code>"rs-latin"</code> , <code>"ru"</code> , <code>"sk"</code> , <code>"sl"</code> , <code>"sv"</code> , <code>"sw"</code> , <code>"th"</code> , <code>"tr"</code> , <code>"uk"</code> , <code>"zh-CN"</code> , and <code>"zh-TW"</code> .

## Description

Creates a text input which, when clicked on, brings up a calendar that the user can click on to select dates.

## Details

The date `format` string specifies how the date will be displayed in the browser. It allows the following values:

- `yy` Year without century (12)
- `yyyy` Year with century (2012)
- `mm` Month number, with leading zero (01-12)
- `m` Month number, without leading zero (01-12)
- `M` Abbreviated month name
- `MM` Full month name
- `dd` Day of month with leading zero
- `d` Day of month without leading zero
- `D` Abbreviated weekday name
- `DD` Full weekday name

## Examples

```
dateInput("date", "Date:", value = "2012-02-29")
```

```
<div id="date" class="shiny-date-input form-group shiny-input-container">
  <label class="control-label" for="date">Date:</label>
  <input type="text" class="form-control datepicker" data-date-language="en" data-date-weekstart="0"
" data-date-format="yyyy-mm-dd" data-date-start-view="month" data-initial-date="2012-02-29"/>
</div>
```

```
# Default value is the date in client's time zone
```

```
dateInput("date", "Date:")
```

```
<div id="date" class="shiny-date-input form-group shiny-input-container">
  <label class="control-label" for="date">Date:</label>
  <input type="text" class="form-control datepicker" data-date-language="en" data-date-weekstart="0"
" data-date-format="yyyy-mm-dd" data-date-start-view="month"/>
</div>
```

```
# value is always yyyy-mm-dd, even if the display format is different
```

```
dateInput("date", "Date:", value = "2012-02-29", format = "mm/dd/yy")
```

```
<div id="date" class="shiny-date-input form-group shiny-input-container">
  <label class="control-label" for="date">Date:</label>
  <input type="text" class="form-control datepicker" data-date-language="en" data-date-weekstart="0"
" data-date-format="mm/dd/yy" data-date-start-view="month" data-initial-date="2012-02-29"/>
</div>
```

```
# Pass in a Date object
```

```
dateInput("date", "Date:", value = Sys.Date()-10)
```

```
<div id="date" class="shiny-date-input form-group shiny-input-container">
  <label class="control-label" for="date">Date:</label>
  <input type="text" class="form-control datepicker" data-date-language="en" data-date-weekstart="0"
" data-date-format="yyyy-mm-dd" data-date-start-view="month" data-initial-date="2015-06-02"/>
</div>
```

```
# Use different language and different first day of week
```

```
dateInput("date", "Date:",
  language = "de",
  weekstart = 1)
```

```
<div id="date" class="shiny-date-input form-group shiny-input-container">
  <label class="control-label" for="date">Date:</label>
  <input type="text" class="form-control datepicker" data-date-language="de" data-date-weekstart="1"
" data-date-format="yyyy-mm-dd" data-date-start-view="month"/>
</div>
```

```
# Start with decade view instead of default month view
```

```
dateInput("date", "Date:",
  startview = "decade")
```

```
<div id="date" class="shiny-date-input form-group shiny-input-container">
  <label class="control-label" for="date">Date:</label>
  <input type="text" class="form-control datepicker" data-date-language="en" data-date-weekstart="0"
  " data-date-format="yyyy-mm-dd" data-date-start-view="decade"/>
</div>
```

## See also

`dateRangeInput` , `updateDateInput` Other input elements: `actionButton` , `actionLink` ; `animationOptions` , `sliderInput` ; `checkboxGroupInput` ; `checkboxInput` ; `dateRangeInput` ; `fileInput` ; `numericInput` ; `passwordInput` ; `radioButtons` ; `selectInput` , `selectizeInput` ; `submitButton` ; `textInput`

### 3.30 Create date range input

# Create date range input

```
dateRangeInput(inputId, label, start = NULL, end = NULL, min = NULL, max = NULL,  
  format = "yyyy-mm-dd", startview = "month", weekstart = 0, language = "en",  
  separator = " to ")
```

## Arguments

<code>inputId</code>	The <code>input</code> slot that will be used to access the value.
<code>label</code>	Display label for the control, or <code>NULL</code> for no label.
<code>start</code>	The initial start date. Either a <code>Date</code> object, or a string in <code>yyyy-mm-dd</code> format. If <code>NULL</code> (the default), will use the current date in the client's time zone.
<code>end</code>	The initial end date. Either a <code>Date</code> object, or a string in <code>yyyy-mm-dd</code> format. If <code>NULL</code> (the default), will use the current date in the client's time zone.
<code>min</code>	The minimum allowed date. Either a <code>Date</code> object, or a string in <code>yyyy-mm-dd</code> format.
<code>max</code>	The maximum allowed date. Either a <code>Date</code> object, or a string in <code>yyyy-mm-dd</code> format.
<code>format</code>	The format of the date to display in the browser. Defaults to <code>"yyyy-mm-dd"</code> .
<code>startview</code>	The date range shown when the input object is first clicked. Can be <code>"month"</code> (the default), <code>"year"</code> , or <code>"decade"</code> .
<code>weekstart</code>	Which day is the start of the week. Should be an integer from 0 (Sunday) to 6 (Saturday).
<code>language</code>	The language used for month and day names. Default is <code>"en"</code> . Other valid values include <code>"bg"</code> , <code>"ca"</code> , <code>"cs"</code> , <code>"da"</code> , <code>"de"</code> , <code>"el"</code> , <code>"es"</code> , <code>"fi"</code> , <code>"fr"</code> , <code>"he"</code> , <code>"hr"</code> , <code>"hu"</code> , <code>"id"</code> , <code>"is"</code> , <code>"it"</code> , <code>"ja"</code> , <code>"kr"</code> , <code>"lt"</code> , <code>"lv"</code> , <code>"ms"</code> , <code>"nb"</code> , <code>"nl"</code> , <code>"pl"</code> , <code>"pt"</code> , <code>"pt-BR"</code> , <code>"ro"</code> , <code>"rs"</code> , <code>"rs-latin"</code> , <code>"ru"</code> , <code>"sk"</code> , <code>"sl"</code> , <code>"sv"</code> , <code>"sw"</code> , <code>"th"</code> , <code>"tr"</code> , <code>"uk"</code> , <code>"zh-CN"</code> , and <code>"zh-TW"</code> .
<code>separator</code>	String to display between the start and end input boxes.

## Description

Creates a pair of text inputs which, when clicked on, bring up calendars that the user can click on to select dates.

## Details

The date `format` string specifies how the date will be displayed in the browser. It allows the following values:

- `yy` Year without century (12)
- `yyyy` Year with century (2012)
- `mm` Month number, with leading zero (01-12)
- `m` Month number, without leading zero (01-12)
- `M` Abbreviated month name
- `MM` Full month name
- `dd` Day of month with leading zero

- d Day of month without leading zero
- D Abbreviated weekday name
- DD Full weekday name

## Examples

```
dateRangeInput("daterange", "Date range:",
              start = "2001-01-01",
              end   = "2010-12-31")
```

```
<div id="daterange" class="shiny-date-range-input form-group shiny-input-container">
  <label class="control-label" for="daterange">Date range:</label>
  <div class="input-daterange input-group">
    <input class="input-sm form-control" type="text" data-date-language="en" data-date-weekstart="0"
    " data-date-format="yyyy-mm-dd" data-date-start-view="month" data-initial-date="2001-01-01"/>
    <span class="input-group-addon"> to </span>
    <input class="input-sm form-control" type="text" data-date-language="en" data-date-weekstart="0"
    " data-date-format="yyyy-mm-dd" data-date-start-view="month" data-initial-date="2010-12-31"/>
  </div>
</div>
```

```
# Default start and end is the current date in the client's time zone
dateRangeInput("daterange", "Date range:")
```

```
<div id="daterange" class="shiny-date-range-input form-group shiny-input-container">
  <label class="control-label" for="daterange">Date range:</label>
  <div class="input-daterange input-group">
    <input class="input-sm form-control" type="text" data-date-language="en" data-date-weekstart="0"
    " data-date-format="yyyy-mm-dd" data-date-start-view="month"/>
    <span class="input-group-addon"> to </span>
    <input class="input-sm form-control" type="text" data-date-language="en" data-date-weekstart="0"
    " data-date-format="yyyy-mm-dd" data-date-start-view="month"/>
  </div>
</div>
```

```
# start and end are always specified in yyyy-mm-dd, even if the display
# format is different
```

```
dateRangeInput("daterange", "Date range:",
              start = "2001-01-01",
              end   = "2010-12-31",
              min   = "2001-01-01",
              max   = "2012-12-21",
              format = "mm/dd/yy",
              separator = " - ")
```

```
<div id="daterange" class="shiny-date-range-input form-group shiny-input-container">
  <label class="control-label" for="daterange">Date range:</label>
  <div class="input-daterange input-group">
    <input class="input-sm form-control" type="text" data-date-language="en" data-date-weekstart="0"
    " data-date-format="mm/dd/yy" data-date-start-view="month" data-min-date="2001-01-01" data-max-dat
    e="2012-12-21" data-initial-date="2001-01-01"/>
    <span class="input-group-addon"> - </span>
    <input class="input-sm form-control" type="text" data-date-language="en" data-date-weekstart="0"
    " data-date-format="mm/dd/vv" data-date-start-view="month" data-min-date="2001-01-01" data-max-dat
```

```
e="2012-12-21" data-initial-date="2010-12-31"/>
</div>
</div>
```

#### # Pass in Date objects

```
dateRangeInput("daterange", "Date range:",
  start = Sys.Date()-10,
  end = Sys.Date()+10)
```

```
<div id="daterange" class="shiny-date-range-input form-group shiny-input-container">
  <label class="control-label" for="daterange">Date range:</label>
  <div class="input-daterange input-group">
    <input class="input-sm form-control" type="text" data-date-language="en" data-date-weekstart="0"
" data-date-format="yyyy-mm-dd" data-date-start-view="month" data-initial-date="2015-06-02"/>
    <span class="input-group-addon"> to </span>
    <input class="input-sm form-control" type="text" data-date-language="en" data-date-weekstart="0"
" data-date-format="yyyy-mm-dd" data-date-start-view="month" data-initial-date="2015-06-22"/>
  </div>
</div>
```

#### # Use different language and different first day of week

```
dateRangeInput("daterange", "Date range:",
  language = "de",
  weekstart = 1)
```

```
<div id="daterange" class="shiny-date-range-input form-group shiny-input-container">
  <label class="control-label" for="daterange">Date range:</label>
  <div class="input-daterange input-group">
    <input class="input-sm form-control" type="text" data-date-language="de" data-date-weekstart="1"
" data-date-format="yyyy-mm-dd" data-date-start-view="month"/>
    <span class="input-group-addon"> to </span>
    <input class="input-sm form-control" type="text" data-date-language="de" data-date-weekstart="1"
" data-date-format="yyyy-mm-dd" data-date-start-view="month"/>
  </div>
</div>
```

#### # Start with decade view instead of default month view

```
dateRangeInput("daterange", "Date range:",
  startview = "decade")
```

```
<div id="daterange" class="shiny-date-range-input form-group shiny-input-container">
  <label class="control-label" for="daterange">Date range:</label>
  <div class="input-daterange input-group">
    <input class="input-sm form-control" type="text" data-date-language="en" data-date-weekstart="0"
" data-date-format="yyyy-mm-dd" data-date-start-view="decade"/>
    <span class="input-group-addon"> to </span>
    <input class="input-sm form-control" type="text" data-date-language="en" data-date-weekstart="0"
" data-date-format="yyyy-mm-dd" data-date-start-view="decade"/>
  </div>
</div>
```

## See also

[dateInput](#) , [updateDateRangeInput](#) Other input.elements: [actionButton](#) , [actionLink](#) ; [animationOptions](#) ,

`sliderInput ; checkboxGroupInput ; checkboxInput ; dateInput ; fileInput ; numericInput ;  
passwordInput ; radioButtons ; selectInput , selectizeInput ; submitButton ; textInput`



### 3.31 File Upload Control

# File Upload Control

```
fileInput(inputId, label, multiple = FALSE, accept = NULL)
```

## Arguments

`inputId` The `input` slot that will be used to access the value.

`label` Display label for the control, or `NULL` for no label.

`multiple` Whether the user should be allowed to select and upload multiple files at once. Does not work on older browsers, including Internet Explorer 9 and earlier.

`accept` A character vector of MIME types; gives the browser a hint of what kind of files the server is expecting.

## Description

Create a file upload control that can be used to upload one or more files.

## Details

Whenever a file upload completes, the corresponding input variable is set to a dataframe. This dataframe contains one row for each selected file, and the following columns:

### `name`

The filename provided by the web browser. This is not the path to read to get at the actual data that was uploaded (see `datapath` column).

### `size`

The size of the uploaded data, in bytes.

### `type`

The MIME type reported by the browser (for example, `text/plain`), or empty string if the browser didn't know.

### `datapath`

The path to a temp file that contains the data that was uploaded. This file may be deleted if the user performs another upload operation.

## See also

Other input.elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `numericInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `submitButton`; `textInput`

### 3.32 Create a numeric input control

# Create a numeric input control

```
numericInput(inputId, label, value, min = NA, max = NA, step = NA)
```

## Arguments

`inputId` The `input` slot that will be used to access the value.

`label` Display label for the control, or `NULL` for no label.

`value` Initial value.

`min` Minimum allowed value

`max` Maximum allowed value

`step` Interval to use when stepping between min and max

## Value

A numeric input control that can be added to a UI definition.

## Description

Create an input control for entry of numeric values

## Examples

```
numericInput("obs", "Observations:", 10,  
            min = 1, max = 100)
```

```
<div class="form-group shiny-input-container">  
  <label for="obs">Observations:</label>  
  <input id="obs" type="number" class="form-control" value="10" min="1" max="100"/>  
</div>
```

## See also

`updateNumericInput` Other input.elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `fileInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `submitButton`; `textInput`

### 3.33 Create radio buttons

# Create radio buttons

```
radioButtons(inputId, label, choices, selected = NULL, inline = FALSE)
```

## Arguments

<code>inputId</code>	The <code>input</code> slot that will be used to access the value.
<code>label</code>	Display label for the control, or <code>NULL</code> for no label.
<code>choices</code>	List of values to select from (if elements of the list are named then that name rather than the value is displayed to the user)
<code>selected</code>	The initially selected value (if not specified then defaults to the first value)
<code>inline</code>	If <code>TRUE</code> , render the choices inline (i.e. horizontally)

## Value

A set of radio buttons that can be added to a UI definition.

## Description

Create a set of radio buttons used to select an item from a list.

## Examples

```
radioButtons("dist", "Distribution type:",  
             c("Normal" = "norm",  
               "Uniform" = "unif",  
               "Log-normal" = "lnorm",  
               "Exponential" = "exp"))
```

```
<div id="dist" class="form-group shiny-input-radiogroup shiny-input-container">  
  <label class="control-label" for="dist">Distribution type:</label>  
  <div class="shiny-options-group">  
    <div class="radio">  
      <label>  
        <input type="radio" name="dist" value="norm" checked="checked"/>  
        <span>Normal</span>  
      </label>  
    </div>  
    <div class="radio">  
      <label>  
        <input type="radio" name="dist" value="unif"/>  
        <span>Uniform</span>  
      </label>  
    </div>  
  </div>  
</div>
```

```
</label>
</div>
<div class="radio">
  <label>
    <input type="radio" name="dist" value="lnorm"/>
    <span>Log-normal</span>
  </label>
</div>
<div class="radio">
  <label>
    <input type="radio" name="dist" value="exp"/>
    <span>Exponential</span>
  </label>
</div>
</div>
</div>
```

## See also

[updateRadioButtons](#) Other input.elements: [actionButton](#) , [actionLink](#) ; [animationOptions](#) , [sliderInput](#) ; [checkboxGroupInput](#) ; [checkboxInput](#) ; [dateInput](#) ; [dateRangeInput](#) ; [fileInput](#) ; [numericInput](#) ; [passwordInput](#) ; [selectInput](#) , [selectizeInput](#) ; [submitButton](#) ; [textInput](#)

### 3.34 Create a select list input control

# Create a select list input control

```
selectInput(inputId, label, choices, selected = NULL, multiple = FALSE, selectize = TRUE,  
            width = NULL, size = NULL)
```

```
selectizeInput(inputId, ..., options = NULL, width = NULL)
```

## Arguments

<code>inputId</code>	The <code>input</code> slot that will be used to access the value.
<code>label</code>	Display label for the control, or <code>NULL</code> for no label.
<code>choices</code>	List of values to select from. If elements of the list are named then that name rather than the value is displayed to the user.
<code>selected</code>	The initially selected value (or multiple values if <code>multiple = TRUE</code> ). If not specified then defaults to the first value for single-select lists and no values for multiple select lists.
<code>multiple</code>	Is selection of multiple items allowed?
<code>selectize</code>	Whether to use selectize.js or not.
<code>width</code>	The width of the input, e.g. <code>'400px'</code> , or <code>'100%'</code> ; see <code>validateCssUnit</code> .
<code>size</code>	Number of items to show in the selection box; a larger number will result in a taller box. Not compatible with <code>selectize=TRUE</code> . Normally, when <code>multiple=FALSE</code> , a select input will be a drop-down list, but when <code>size</code> is set, it will be a box instead.
<code>...</code>	Arguments passed to <code>selectInput()</code> .
<code>options</code>	A list of options. See the documentation of selectize.js for possible options (character option values inside <code>I()</code> will be treated as literal JavaScript code; see <code>renderDataTable()</code> for details).

## Value

A select list control that can be added to a UI definition.

## Description

Create a select list that can be used to choose a single or multiple items from a list of values.

## Details

By default, `selectInput()` and `selectizeInput()` use the JavaScript library `selectize.js` (<https://github.com/brianreavis/selectize.js>) to instead of the basic select input element. To use the standard HTML select input element, use `selectInput()` with `selectize=FALSE`.

## Note

The `selectize` input created from `selectizeInput()` allows deletion of the selected option even in a single select input, which will return an empty string as its value. This is the default behavior of `selectize.js`. However, the `selectize` input created from `selectInput(..., selectize = TRUE)` will ignore the empty string value when it is a single choice input and the empty string is not in the `choices` argument. This is to keep compatibility with `selectInput(..., selectize = FALSE)`.

## Examples

```
selectInput("variable", "Variable:",
  c("Cylinders" = "cyl",
    "Transmission" = "am",
    "Gears" = "gear"))
```

```
<div class="form-group shiny-input-container">
  <label class="control-label" for="variable">Variable:</label>
  <div>
    <select id="variable"><option value="cyl" selected>Cylinders</option>
    <option value="am">Transmission</option>
    <option value="gear">Gears</option></select>
    <script type="application/json" data-for="variable" data-nonempty="">{}</script>
  </div>
</div>
```

## See also

`updateSelectInput` Other input elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `fileInput`; `numericInput`; `passwordInput`; `radioButtons`; `submitButton`; `textInput`

## 3.35 Slider Input Widget

# Slider Input Widget

```
sliderInput(inputId, label, min, max, value, step = NULL, round = FALSE, format = NULL,
  locale = NULL, ticks = TRUE, animate = FALSE, width = NULL, sep = ",", pre = NULL,
  post = NULL)
```

```
animationOptions(interval = 1000, loop = FALSE, playButton = NULL, pauseButton = NULL)
```

## Arguments

<code>inputId</code>	The <code>input</code> slot that will be used to access the value.
<code>label</code>	Display label for the control, or <code>NULL</code> for no label.
<code>min</code>	The minimum value (inclusive) that can be selected.
<code>max</code>	The maximum value (inclusive) that can be selected.
<code>value</code>	The initial value of the slider. A numeric vector of length one will create a regular slider; a numeric vector of length two will create a double-ended range slider. A warning will be issued if the value doesn't fit between <code>min</code> and <code>max</code> .
<code>step</code>	Specifies the interval between each selectable value on the slider (if <code>NULL</code> , a heuristic is used to determine the step size).
<code>round</code>	<code>TRUE</code> to round all values to the nearest integer; <code>FALSE</code> if no rounding is desired; or an integer to round to that number of digits (for example, 1 will round to the nearest 10, and -2 will round to the nearest .01). Any rounding will be applied after snapping to the nearest step.
<code>format</code>	Deprecated.
<code>locale</code>	Deprecated.
<code>ticks</code>	<code>FALSE</code> to hide tick marks, <code>TRUE</code> to show them according to some simple heuristics.
<code>animate</code>	<code>TRUE</code> to show simple animation controls with default settings; <code>FALSE</code> not to; or a custom settings list, such as those created using <code>animationOptions</code> .
<code>width</code>	The width of the input, e.g. <code>'400px'</code> , or <code>'100%'</code> ; see <code>validateCssUnit</code> .
<code>sep</code>	Separator between thousands places in numbers.
<code>pre</code>	A prefix string to put in front of the value.
<code>post</code>	A suffix string to put after the value.
<code>interval</code>	The interval, in milliseconds, between each animation step.
<code>loop</code>	<code>TRUE</code> to automatically restart the animation when it reaches the end.
<code>playButton</code>	Specifies the appearance of the play button. Valid values are a one-element character vector (for a simple text label), an HTML tag or list of tags (using <code>tag</code> and friends), or raw HTML (using <code>HTML</code> ).

`pauseButton` Similar to `playButton`, but for the pause button.

## Description

Constructs a slider widget to select a numeric value from a range.

## See also

`updateSliderInput` Other input elements: `actionButton`, `actionLink`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `fileInput`; `numericInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `submitButton`; `textInput`



### 3.36 Create a submit button

# Create a submit button

```
submitButton(text = "Apply Changes", icon = NULL)
```

## Arguments

`text` Button caption

`icon` Optional `icon` to appear on the button

## Value

A submit button that can be added to a UI definition.

## Description

Create a submit button for an input form. Forms that include a submit button do not automatically update their outputs when inputs change, rather they wait until the user explicitly clicks the submit button.

## Examples

```
submitButton("Update View")
```

```
<div>  
  <button type="submit" class="btn btn-primary">Update View</button>  
</div>
```

```
submitButton("Update View", icon("refresh"))
```

```
<div>  
  <button type="submit" class="btn btn-primary">  
    <i class="fa fa-refresh"></i>  
    Update View  
  </button>  
</div>
```

## See also

Other input.elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `fileInput`; `numericInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `textInput`

### 3.37 Create a text input control

# Create a text input control

```
textInput(inputId, label, value = "")
```

## Arguments

`inputId` The `input` slot that will be used to access the value.

`label` Display label for the control, or `NULL` for no label.

`value` Initial value.

## Value

A text input control that can be added to a UI definition.

## Description

Create an input control for entry of unstructured text values

## Examples

```
textInput("caption", "Caption:", "Data Summary")
```

```
<div class="form-group shiny-input-container">
  <label for="caption">Caption:</label>
  <input id="caption" type="text" class="form-control" value="Data Summary"/>
</div>
```

## See also

`updateTextInput` Other input elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `fileInput`; `numericInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `submitButton`

### 3.38 Create a password input control

# Create a password input control

```
passwordInput(inputId, label, value = "")
```

## Arguments

`inputId` The `input` slot that will be used to access the value.

`label` Display label for the control, or `NULL` for no label.

`value` Initial value.

## Value

A text input control that can be added to a UI definition.

## Description

Create an password control for entry of passwords.

## Examples

```
passwordInput("password", "Password:")
```

```
<div class="form-group shiny-input-container">  
  <label for="password">Password:</label>  
  <input id="password" type="password" class="form-control" value=""/>  
</div>
```

## See also

`updateTextInput` Other input elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `fileInput`; `numericInput`; `radioButtons`; `selectInput`, `selectizeInput`; `submitButton`; `textInput`

### 3.39 Change the value of a checkbox

# Change the value of a checkbox group input on the client

```
updateCheckboxGroupInput(session, inputId, label = NULL, choices = NULL, selected = NULL,
  inline = FALSE)
```

## Arguments

- session The `session` object passed to function given to `shinyServer`.
- inputId The id of the input object.
- label The label to set for the input object.
- choices List of values to show checkboxes for. If elements of the list are named then that name rather than the value is displayed to the user.
- selected The values that should be initially selected, if any.
- inline If `TRUE`, render the choices inline (i.e. horizontally)

## Description

Change the value of a checkbox group input on the client

## Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, `numericInput()` and `updateNumericInput()` take a similar set of arguments.

Any arguments with `NULL` values will be ignored; they will not result in any changes to the input object on the client.

## Examples

```
## <strong>Not run</strong>:
# shinyServer(function(input, output, session) {
#
#   observe({
#     # We'll use the input$controller variable multiple times, so save it as x
#     # for convenience.
#     x <- input$controller
#
#     # Create a list of new options, where the name of the items is something
```

```
# # like 'option label x 1', and the values are 'option-x-1'.
#   cb_options <- list()
#   cb_options[[sprintf("option label %d 1", x)]] <- sprintf("option-%d-1", x)
#   cb_options[[sprintf("option label %d 2", x)]] <- sprintf("option-%d-2", x)
#
#   # Change values for input$inCheckboxGroup
#   updateCheckboxGroupInput(session, "inCheckboxGroup", choices = cb_options)
#
#   # Can also set the label and select items
#   updateCheckboxGroupInput(session, "inCheckboxGroup2",
#     label = paste("checkboxgroup label", x),
#     choices = cb_options,
#     selected = sprintf("option-%d-2", x)
#   )
# })
# })
# ## <strong>End(Not run)</strong>
```

## See also

[checkboxGroupInput](#)

### 3.40 Change the value of a checkbox

# Change the value of a checkbox input on the client

```
updateCheckboxInput(session, inputId, label = NULL, value = NULL)
```

## Arguments

`session` The `session` object passed to function given to `shinyServer`.

`inputId` The id of the input object.

`label` The label to set for the input object.

`value` The value to set for the input object.

## Description

Change the value of a checkbox input on the client

## Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, `numericInput()` and `updateNumericInput()` take a similar set of arguments.

Any arguments with NULL values will be ignored; they will not result in any changes to the input object on the client.

## Examples

```
## <strong>Not run</strong>:  
# shinyServer(function(input, output, session) {  
#  
#   observe({  
#     # TRUE if input$controller is even, FALSE otherwise.  
#     x_even <- input$controller %% 2 == 0  
#  
#     updateCheckboxInput(session, "inCheckbox", value = x_even)  
#   })  
# })  
# ## <strong>End(Not run)</strong>
```

## See also

`checkboxInput`

Shiny is an [RStudio](#) project. © 2014 RStudio, Inc.

### 3.41 Change the value of a date input on

# Change the value of a date input on the client

```
updateDateInput(session, inputId, label = NULL, value = NULL, min = NULL, max = NULL)
```

## Arguments

`session` The `session` object passed to function given to `shinyServer`.

`inputId` The id of the input object.

`label` The label to set for the input object.

`value` The desired date value. Either a `Date` object, or a string in `yyyy-mm-dd` format.

`min` The minimum allowed date. Either a `Date` object, or a string in `yyyy-mm-dd` format.

`max` The maximum allowed date. Either a `Date` object, or a string in `yyyy-mm-dd` format.

## Description

Change the value of a date input on the client

## Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, `numericInput()` and `updateNumericInput()` take a similar set of arguments.

Any arguments with `NULL` values will be ignored; they will not result in any changes to the input object on the client.

## Examples

```
## <strong>Not run</strong>:
# shinyServer(function(input, output, session) {
#
#   observe({
#     # We'll use the input$controller variable multiple times, so save it as x
#     # for convenience.
#     x <- input$controller
#
#     updateDateInput(session, "inDate",
#       label = paste("Date label", x),
#       value = paste("2013-04-", x, sep=""),
#
```



*Change the value of a date input on*

```
#      min  = paste("2013-04-", x-1, sep=""),
#      max  = paste("2013-04-", x+1, sep="")
#    )
#  })
# })
# ## End(Not run)</strong>
```

## See also

[dateInput](#)

### 3.42 Change the start and end values of a

# Change the start and end values of a date range input on the client

```
updateDateRangeInput(session, inputId, label = NULL, start = NULL, end = NULL, min = NULL,
  max = NULL)
```

## Arguments

- `session` The `session` object passed to function given to `shinyServer`.
- `inputId` The id of the input object.
- `label` The label to set for the input object.
- `start` The start date. Either a `Date` object, or a string in `yyyy-mm-dd` format.
- `end` The end date. Either a `Date` object, or a string in `yyyy-mm-dd` format.
- `min` The minimum allowed date. Either a `Date` object, or a string in `yyyy-mm-dd` format.
- `max` The maximum allowed date. Either a `Date` object, or a string in `yyyy-mm-dd` format.

## Description

Change the start and end values of a date range input on the client

## Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, `numericInput()` and `updateNumericInput()` take a similar set of arguments.

Any arguments with `NULL` values will be ignored; they will not result in any changes to the input object on the client.

## Examples

```
## <strong>Not run</strong>:
# shinyServer(function(input, output, session) {
#
#   observe({
#     # We'll use the input$controller variable multiple times, so save it as x
#     # for convenience.
#     x <- input$controller
#
#   })
# }
```

*Change the start and end values of a*

```
#   updateDateRangeInput(session, "inDateRange",
#     label = paste("Date range label", x),
#     start = paste("2013-01-", x, sep=""),
#     end = paste("2013-12-", x, sep=""))
#   })
# })
# ## <strong>End(Not run)</strong>
```

## See also

[dateRangeInput](#)

### 3.43 Change the value of a number input

# Change the value of a number input on the client

```
updateNumericInput(session, inputId, label = NULL, value = NULL, min = NULL, max = NULL,
  step = NULL)
```

## Arguments

`session` The `session` object passed to function given to `shinyServer`.

`inputId` The id of the input object.

`label` The label to set for the input object.

`value` The value to set for the input object.

`min` Minimum value.

`max` Maximum value.

`step` Step size.

## Description

Change the value of a number input on the client

## Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, `numericInput()` and `updateNumericInput()` take a similar set of arguments.

Any arguments with NULL values will be ignored; they will not result in any changes to the input object on the client.

## Examples

```
## <strong>Not run</strong>:
# shinyServer(function(input, output, session) {
#
#   observe({
#     # We'll use the input$controller variable multiple times, so save it as x
#     # for convenience.
#     x <- input$controller
#   })
# }
```

*Change the value of a number input*

```
#   updateNumericInput(session, "inNumber", value = x)
#
#   updateNumericInput(session, "inNumber2",
#     label = paste("Number label ", x),
#     value = x, min = x-10, max = x+10, step = 5)
# })
# })
# ## <strong>End(Not run)</strong>
```

## See also

[numericInput](#)

### 3.44 Change the value of a radio input on

# Change the value of a radio input on the client

```
updateRadioButtons(session, inputId, label = NULL, choices = NULL, selected = NULL,
  inline = FALSE)
```

## Arguments

- `session` The `session` object passed to function given to `shinyServer`.
- `inputId` The id of the input object.
- `label` The label to set for the input object.
- `choices` List of values to select from (if elements of the list are named then that name rather than the value is displayed to the user)
- `selected` The initially selected value (if not specified then defaults to the first value)
- `inline` If `TRUE`, render the choices inline (i.e. horizontally)

## Description

Change the value of a radio input on the client

## Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, `numericInput()` and `updateNumericInput()` take a similar set of arguments.

Any arguments with `NULL` values will be ignored; they will not result in any changes to the input object on the client.

## Examples

```
## <strong>Not run</strong>:
# shinyServer(function(input, output, session) {
#
#   observe({
#     # We'll use the input$controller variable multiple times, so save it as x
#     # for convenience.
#     x <- input$controller
#
#     r_options <- list()
```

*Change the value of a radio input on*

```
#   r_options[[sprintf("option label %d 1", x)]] <- sprintf("option-%d-1", x)
#   r_options[[sprintf("option label %d 2", x)]] <- sprintf("option-%d-2", x)
#
#   # Change values for input$inRadio
#   updateRadioButtons(session, "inRadio", choices = r_options)
#
#   # Can also set the label and select an item
#   updateRadioButtons(session, "inRadio2",
#     label = paste("Radio label", x),
#     choices = r_options,
#     selected = sprintf("option-%d-2", x)
#   )
# })
# })
# ## <strong>End(Not run)</strong>
```

## See also

[radioButtons](#)

### 3.45 Change the value of a select input on

# Change the value of a select input on the client

```
updateSelectInput(session, inputId, label = NULL, choices = NULL, selected = NULL)
```

```
updateSelectizeInput(session, inputId, label = NULL, choices = NULL, selected = NULL,  
options = list(), server = FALSE)
```

## Arguments

- session** The `session` object passed to function given to `shinyServer`.
- inputId** The id of the input object.
- label** The label to set for the input object.
- choices** List of values to select from. If elements of the list are named then that name rather than the value is displayed to the user.
- selected** The initially selected value (or multiple values if `multiple = TRUE`). If not specified then defaults to the first value for single-select lists and no values for multiple select lists.
- options** A list of options. See the documentation of `selectize.js` for possible options (character option values inside `I()` will be treated as literal JavaScript code; see `renderDataTable()` for details).
- server** whether to store `choices` on the server side, and load the select options dynamically on searching, instead of writing all `choices` into the page at once (i.e., only use the client-side version of `selectize.js`)

## Description

Change the value of a select input on the client

## Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, `numericInput()` and `updateNumericInput()` take a similar set of arguments.

Any arguments with NULL values will be ignored; they will not result in any changes to the input object on the client.

## Examples

```
## <strong>Not run</strong>:
```



```
# shinyServer(function(input, output, session) {  
#  
#   observe({  
#     # We'll use the input$controller variable multiple times, so save it as x  
#     # for convenience.  
#     x <- input$controller  
#  
#     # Create a list of new options, where the name of the items is something  
#     # like 'option label x 1', and the values are 'option-x-1'.  
#     s_options <- list()  
#     s_options[[sprintf("option label %d 1", x)]] <- sprintf("option-%d-1", x)  
#     s_options[[sprintf("option label %d 2", x)]] <- sprintf("option-%d-2", x)  
#  
#     # Change values for input$select  
#     updateSelectInput(session, "inSelect", choices = s_options)  
#  
#     # Can also set the label and select an item (or more than one if it's a  
#     # multi-select)  
#     updateSelectInput(session, "inSelect2",  
#       label = paste("Select label", x),  
#       choices = s_options,  
#       selected = sprintf("option-%d-2", x)  
#     )  
#   })  
# })  
# ## <strong>End(Not run)</strong>
```

## See also

[selectInput](#)

### 3.46 Change the value of a slider input on

# Change the value of a slider input on the client

```
updateSliderInput(session, inputId, label = NULL, value = NULL, min = NULL, max = NULL,
  step = NULL)
```

## Arguments

`session` The `session` object passed to function given to `shinyServer`.

`inputId` The id of the input object.

`label` The label to set for the input object.

`value` The value to set for the input object.

`min` Minimum value.

`max` Maximum value.

`step` Step size.

## Description

Change the value of a slider input on the client

## Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, `numericInput()` and `updateNumericInput()` take a similar set of arguments.

Any arguments with NULL values will be ignored; they will not result in any changes to the input object on the client.

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  shinyApp(
    ui = fluidPage(
      sidebarLayout(
        sidebarPanel(
          p("The first slider controls the second"),
          slider2Input("control", "Controller:", min=0, max=20, value=10,
```

```
      step=1),
      slider2Input("receive", "Receiver:", min=0, max=20, value=10,
                  step=1)
    ),
    mainPanel()
  )
),
server = function(input, output, session) {
  observe({
    val <- input$control
    # Control the value, min, max, and step.
    # Step size is 2 when input value is even; 1 when value is odd.
    updateSliderInput(session, "receive", value = val,
                      min = floor(val/2), max = val+4, step = (val+1)%2 + 1)
  })
}
)
```

## See also

[sliderInput](#)

### 3.47 Change the selected tab on the client

# Change the selected tab on the client

```
updateTabsetPanel (session, inputId, selected = NULL)
```

## Arguments

`session` The `session` object passed to function given to `shinyServer` .

`inputId` The id of the `tabsetPanel` , `navlistPanel` , or `navbarPage` object.

`selected` The name of the tab to make active.

## Description

Change the selected tab on the client

## Examples

```
## <strong>Not run</strong>:  
# shinyServer(function(input, output, session) {  
#  
#   observe({  
#     # TRUE if input$controller is even, FALSE otherwise.  
#     x_even <- input$controller %% 2 == 0  
#  
#     # Change the selected tab.  
#     # Note that the tabset container must have been created with an 'id' argument  
#     if (x_even) {  
#       updateTabsetPanel(session, "inTabset", selected = "panel2")  
#     } else {  
#       updateTabsetPanel(session, "inTabset", selected = "panel1")  
#     }  
#   })  
# })  
# ## <strong>End(Not run)</strong>
```

## See also

[tabsetPanel](#) , [navlistPanel](#) , [navbarPage](#)

### 3.48 Change the value of a text input on

# Change the value of a text input on the client

```
updateTextInput(session, inputId, label = NULL, value = NULL)
```

## Arguments

`session` The `session` object passed to function given to `shinyServer`.

`inputId` The id of the input object.

`label` The label to set for the input object.

`value` The value to set for the input object.

## Description

Change the value of a text input on the client

## Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, `numericInput()` and `updateNumericInput()` take a similar set of arguments.

Any arguments with NULL values will be ignored; they will not result in any changes to the input object on the client.

## Examples

```
## <strong>Not run</strong>:  
# shinyServer(function(input, output, session) {  
#  
#   observe({  
#     # We'll use the input$controller variable multiple times, so save it as x  
#     # for convenience.  
#     x <- input$controller  
#  
#     # This will change the value of input$inText, based on x  
#     updateTextInput(session, "inText", value = paste("New text", x))  
#  
#     # Can also set the label, this time for input$inText2  
#     updateTextInput(session, "inText2",  
#       label = paste("New label", x),
```

*Change the value of a text input on*

```
#       value = paste("New text", x))  
#   })  
# })  
# ## <strong>End(Not run)</strong>
```

## See also

`textInput`

### 3.49 Create an HTML output element

# Create an HTML output element

```
htmlOutput(outputId, inline = FALSE, container = if (inline) span else div, ...)
```

```
uiOutput(outputId, inline = FALSE, container = if (inline) span else div, ...)
```

## Arguments

<code>outputId</code>	output variable to read the value from
<code>inline</code>	use an inline ( <code>span()</code> ) or block container ( <code>div()</code> ) for the output
<code>container</code>	a function to generate an HTML element to contain the text
<code>...</code>	Other arguments to pass to the container tag function. This is useful for providing additional classes for the tag.

## Value

An HTML output element that can be included in a panel

## Description

Render a reactive output variable as HTML within an application page. The text will be included within an HTML `div` tag, and is presumed to contain HTML content which should not be escaped.

## Details

`uiOutput` is intended to be used with `renderUI` on the server side. It is currently just an alias for `htmlOutput`.

## Examples

```
htmlOutput("summary")
```

```
<div id="summary" class="shiny-html-output"></div>
```

```
# Using a custom container and class
```

```
tags$ul(  
  htmlOutput("summary", container = tags$li, class = "custom-li-output")  
)
```

```
<ul>
```

```
  <li class="shiny-html-output custom-li-output" id="summary"></li>
```

```
</ul>
```

## 3.50 Create an plot or image output

# Create an plot or image output element

```
imageOutput(outputId, width = "100%", height = "400px", click = NULL, dblclick = NULL,
  hover = NULL, hoverDelay = NULL, hoverDelayType = NULL, brush = NULL, clickId = NULL,
  hoverId = NULL, inline = FALSE)
```

```
plotOutput(outputId, width = "100%", height = "400px", click = NULL, dblclick = NULL,
  hover = NULL, hoverDelay = NULL, hoverDelayType = NULL, brush = NULL, clickId = NULL,
  hoverId = NULL, inline = FALSE)
```

## Arguments

outputId	output variable to read the plot/image from.
width,height	Image width/height. Must be a valid CSS unit (like "100%", "400px", "auto") or a number, which will be coerced to a string and have "px" appended. These two arguments are ignored when <code>inline = TRUE</code> , in which case the width/height of a plot must be specified in <code>renderPlot()</code> . Note that, for height, using "auto" or "100%" generally will not work as expected, because of how height is computed with HTML/CSS.
click	This can be <code>NULL</code> (the default), a string, or an object created by the <code>clickOpts</code> function. If you use a value like "plot_click" (or equivalently, <code>clickOpts(id="plot_click")</code> ), the plot will send coordinates to the server whenever it is clicked, and the value will be accessible via <code>input\$plot_click</code> . The value will be a named list with <code>x</code> and <code>y</code> elements indicating the mouse position.
dblclick	This is just like the <code>click</code> argument, but for double-click events.
hover	Similar to the <code>click</code> argument, this can be <code>NULL</code> (the default), a string, or an object created by the <code>hoverOpts</code> function. If you use a value like "plot_hover" (or equivalently, <code>hoverOpts(id="plot_hover")</code> ), the plot will send coordinates to the server pauses on the plot, and the value will be accessible via <code>input\$plot_hover</code> . The value will be a named list with <code>x</code> and <code>y</code> elements indicating the mouse position. To control the hover time or hover delay type, you must use <code>hoverOpts</code> .
hoverDelay	Deprecated; use <code>hover</code> instead. Also see the <code>hoverOpts</code> function.
hoverDelayType	Deprecated; use <code>hover</code> instead. Also see the <code>hoverOpts</code> function.
brush	Similar to the <code>click</code> argument, this can be <code>NULL</code> (the default), a string, or an object created by the <code>brushOpts</code> function. If you use a value like "plot_brush" (or equivalently, <code>brushOpts(id="plot_brush")</code> ), the plot will allow the user to "brush" in the plotting area, and will send information about the brushed area to the server, and the value will be accessible via <code>input\$plot_brush</code> . Brushing means that the user will be able to draw a rectangle in the plotting area and drag it around. The value will be a named list with <code>xmin</code> ,



	<code>xmax</code> , <code>ymi n</code> , and <code>ymax</code> elements indicating the brush area. To control the brush behavior, use <code>brush0pts</code> .
<code>clickId</code>	Deprecated; use <code>click</code> instead. Also see the <code>click0pts</code> function.
<code>hoverId</code>	Deprecated; use <code>hover</code> instead. Also see the <code>hover0pts</code> function.
<code>inline</code>	use an inline ( <code>span()</code> ) or block container ( <code>div()</code> ) for the output

## Value

A plot or image output element that can be included in a panel.

## Description

Render a `renderPlot` or `renderImage` within an application page.

## Note

The arguments `clickId` and `hoverId` only work for R base graphics (see the `graphics` package). They do not work for `grid`-based graphics, such as `ggplot2`, `lattice`, and so on.

## Interactive plots

Plots and images in Shiny support mouse-based interaction, via clicking, double-clicking, hovering, and brushing. When these interaction events occur, the mouse coordinates will be sent to the server as `input$` variables, as specified by `click` , `dblclick` , `hover` , or `brush` .

For `plotOutput` , the coordinates will be sent scaled to the data space, if possible. (At the moment, plots generated by base graphics support this scaling, although plots generated by `grid` or `ggplot2` do not.) If scaling is not possible, the raw pixel coordinates will be sent. For `imageOutput` , the coordinates will be sent in raw pixel coordinates.

## Examples

```
# Only run these examples in interactive R sessions
if (interactive()) {

# A basic shiny app with a plotOutput
shinyApp(
  ui = fluidPage(
    sidebarLayout(
      sidebarPanel(
        actionButton("newplot", "New plot")
      ),
      mainPanel(
        plotOutput("plot")
      )
    )
  ),
  server = function(input, output) {
    output$plot <- renderPlot({
      input$newplot
      # Add a little noise to the cars data
      cars2 <- cars + rnorm(nrow(cars))
      plot(cars2)
    })
  }
}
```

```

)

# A demonstration of clicking, hovering, and brushing
shinyApp(
  ui = basicPage(
    fluidRow(
      column(width = 4,
        plotOutput("plot", height=300,
          click = "plot_click", # Equiv, to click=clickOpts(id="plot_click")
          hover = hoverOpts(id = "plot_hover", delayType = "throttle"),
          brush = brushOpts(id = "plot_brush")
        ),
        h4("Clicked points"),
        tableOutput("plot_clickedpoints"),
        h4("Brushed points"),
        tableOutput("plot_brushedpoints")
      ),
      column(width = 4,
        verbatimTextOutput("plot_clickinfo"),
        verbatimTextOutput("plot_hoverinfo")
      ),
      column(width = 4,
        wellPanel(actionButton("newplot", "New plot")),
        verbatimTextOutput("plot_brushinfo")
      )
    )
  ),
  server = function(input, output, session) {
    data <- reactive({
      input$newplot
      # Add a little noise to the cars data so the points move
      cars + rnorm(nrow(cars))
    })
    output$plot <- renderPlot({
      d <- data()
      plot(d$speed, d$dist)
    })
    output$plot_clickinfo <- renderPrint({
      cat("Click:\n")
      str(input$plot_click)
    })
    output$plot_hoverinfo <- renderPrint({
      cat("Hover (throttled):\n")
      str(input$plot_hover)
    })
    output$plot_brushinfo <- renderPrint({
      cat("Brush (debounced):\n")
      str(input$plot_brush)
    })
    output$plot_clickedpoints <- renderTable({
      # For base graphics, we need to specify columns, though for ggplot2,
      # it's usually not necessary.
      res <- nearPoints(data(), input$plot_click, "speed", "dist")
      if (nrow(res) == 0)
        return()
      res
    })
  }
)

```

```

  })
  output$plot_brushedpoints <- renderTable({
    res <- brushedPoints(data(), input$plot_brush, "speed", "dist")
    if (nrow(res) == 0)
      return()
    res
  })
}
)

```

```

# Demo of clicking, hovering, brushing with imageOutput
# Note that coordinates are in pixels
shinyApp(
  ui = basicPage(
    fluidRow(
      column(width = 4,
        imageOutput("image", height=300,
          click = "image_click",
          hover = hoverOpts(
            id = "image_hover",
            delay = 500,
            delayType = "throttle"
          ),
          brush = brushOpts(id = "image_brush")
        ),
      ),
      column(width = 4,
        verbatimTextOutput("image_clickinfo"),
        verbatimTextOutput("image_hoverinfo")
      ),
      column(width = 4,
        wellPanel(actionButton("newimage", "New image")),
        verbatimTextOutput("image_brushinfo")
      )
    )
  ),
  server = function(input, output, session) {
    output$image <- renderImage({
      input$newimage

      # Get width and height of image output
      width <- session$clientData$output_image_width
      height <- session$clientData$output_image_height

      # Write to a temporary PNG file
      outfile <- tempfile(fileext = ".png")

      png(outfile, width=width, height=height)
      plot(rnorm(200), rnorm(200))
      dev.off()

      # Return a list containing information about the image
      list(
        src = outfile,
        contentType = "image/png",
        width = width.

```

```
      height = height,
      alt = "This is alternate text"
    )
  })
  output$image_clickinfo <- renderPrint({
    cat("Click:\n")
    str(input$image_click)
  })
  output$image_hoverinfo <- renderPrint({
    cat("Hover (throttled):\n")
    str(input$image_hover)
  })
  output$image_brushinfo <- renderPrint({
    cat("Brush (debounced):\n")
    str(input$image_brush)
  })
}
)
```

## See also

For the corresponding server-side functions, see [renderPlot](#) and [renderImage](#).

### 3.51 Set options for an output object.

# Set options for an output object.

```
outputOptions(x, name, ...)
```

## Arguments

- `x` A shinyoutput object (typically `output` ).
- `name` The name of an output observer in the shinyoutput object.
- `...` Options to set for the output observer.

## Description

These are the available options for an output object:

- `suspendWhenHidden`. When `TRUE` (the default), the output object will be suspended (not execute) when it is hidden on the web page. When `FALSE`, the output object will not suspend when hidden, and if it was already hidden and suspended, then it will resume immediately.
- `priority`. The priority level of the output object. Queued outputs with higher priority values will execute before those with lower values.

## Examples

```
## <strong>Not run</strong>:  
# # Get the list of options for all observers within output  
# outputOptions(output)  
#  
# # Disable suspend for output$myplot  
# outputOptions(output, "myplot", suspendWhenHidden = FALSE)  
#  
# # Change priority for output$myplot  
# outputOptions(output, "myplot", priority = 10)  
#  
# # Get the list of options for output$myplot  
# outputOptions(output, "myplot")  
# ## <strong>End(Not run)</strong>
```

## 3.52 Create a table output element

# Create a table output element

```
tableOutput(outputId)
```

```
dataTableOutput(outputId)
```

## Arguments

`outputId` output variable to read the table from

## Value

A table output element that can be included in a panel

## Description

Render a `renderTable` or `renderDataTable` within an application page. `renderTable` uses a standard HTML table, while `renderDataTable` uses the DataTables Javascript library to create an interactive table with more features.

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  # table example
  shinyApp(
    ui = fluidPage(
      fluidRow(
        column(12,
          tableOutput('table')
        )
      )
    ),
    server = function(input, output) {
      output$table <- renderTable(iris)
    }
  )
}
```

```
# DataTables example
shinyApp(
  ui = fluidPage(
    fluidRow(
      column(12,
```

```
      dataTableOutput('table')
    )
  )
),
server = function(input, output) {
  output$table <- renderDataTable(iris)
}
)
}
```

## See also

[renderTable](#) , [renderDataTable](#) .

### 3.53 Create a text output element

# Create a text output element

```
textOutput(outputId, container = if (inline) span else div, inline = FALSE)
```

## Arguments

`outputId` output variable to read the value from  
`container` a function to generate an HTML element to contain the text  
`inline` use an inline ( `span()` ) or block container ( `div()` ) for the output

## Value

A text output element that can be included in a panel

## Description

Render a reactive output variable as text within an application page. The text will be included within an HTML `div` tag by default.

## Details

Text is HTML-escaped prior to rendering. This element is often used to display `renderText` output variables.

## Examples

```
h3(textOutput("caption"))
```

```
<h3>  
  <div id="caption" class="shiny-text-output"></div>  
</h3>
```



### 3.54 Create a verbatim text output element

# Create a verbatim text output element

```
verbatimTextOutput(outputId)
```

## Arguments

`outputId` output variable to read the value from

## Value

A verbatim text output element that can be included in a panel

## Description

Render a reactive output variable as verbatim text within an application page. The text will be included within an HTML `pre` tag.

## Details

Text is HTML-escaped prior to rendering. This element is often used with the `renderPrint` function to preserve fixed-width formatting of printed objects.

## Examples

```
mainPanel(  
  h4("Summary"),  
  verbatimTextOutput("summary"),  
  
  h4("Observations"),  
  tableOutput("view")  
)
```

```
<div class="col-sm-8">  
  <h4>Summary</h4>  
  <pre id="summary" class="shiny-text-output"></pre>  
  <h4>Observations</h4>  
  <div id="view" class="shiny-html-output"></div>  
</div>
```

### 3.55 Create a download button or link

# Create a download button or link

```
downloadButton(outputId, label = "Download", class = NULL)
```

```
downloadLink(outputId, label = "Download", class = NULL)
```

## Arguments

`outputId` The name of the output slot that the `downloadHandler` is assigned to.

`label` The label that should appear on the button.

`class` Additional CSS classes to apply to the tag, if any.

## Description

Use these functions to create a download button or link; when clicked, it will initiate a browser download. The filename and contents are specified by the corresponding `downloadHandler` defined in the server function.

## Examples

```
## <strong>Not run</strong>:  
## In server.R:  
# output$downloadData <- downloadHandler(  
#   filename = function() {  
#     paste('data-', Sys.Date(), '.csv', sep='')  
#   },  
#   content = function(con) {  
#     write.csv(data, con)  
#   }  
# )  
## In ui.R:  
# downloadLink('downloadData', 'Download')  
## <strong>End(Not run)</strong>
```

## See also

`downloadHandler`

## 3.56 Reporting progress (object-oriented)

# Reporting progress (object-oriented API)

## Arguments

<code>session</code>	The Shiny session object, as provided by <code>shinyServer</code> to the server function.
<code>min</code>	The value that represents the starting point of the progress bar. Must be less than <code>max</code> .
<code>max</code>	The value that represents the end of the progress bar. Must be greater than <code>min</code> .
<code>message</code>	A single-element character vector; the message to be displayed to the user, or <code>NULL</code> to hide the current message (if any).
<code>detail</code>	A single-element character vector; the detail message to be displayed to the user, or <code>NULL</code> to hide the current detail message (if any). The detail message will be shown with a de-emphasized appearance relative to <code>message</code> .
<code>value</code>	A numeric value at which to set the progress bar, relative to <code>min</code> and <code>max</code> . <code>NULL</code> hides the progress bar, if it is currently visible.
<code>amount</code>	Single-element numeric vector; the value at which to set the progress bar, relative to <code>min</code> and <code>max</code> . <code>NULL</code> hides the progress bar, if it is currently visible.
<code>amount</code>	For the <code>inc()</code> method, a numeric value to increment the progress bar.

## Description

Reports progress to the user during long-running operations.

## Details

This package exposes two distinct programming APIs for working with progress. `withProgress` and `setProgress` together provide a simple function-based interface, while the `Progress` reference class provides an object-oriented API.

Instantiating a `Progress` object causes a progress panel to be created, and it will be displayed the first time the `set` method is called. Calling `close` will cause the progress panel to be removed.

### Methods

`initialize(session, min = 0, max = 1)`

Creates a new progress panel (but does not display it).

`set(value = NULL, message = NULL, detail = NULL)`

Updates the progress panel. When called the first time, the progress panel is displayed.

`inc(amount = 0.1, message = NULL, detail = NULL)`

Like `set`, this updates the progress panel. The difference is that `inc` increases the progress bar by `amount`,

```
close()
```

Removes the progress panel. Future calls to `set` and `close` will be ignored.

## Examples

```
## <strong>Not run</strong>:  
# # server.R  
# shinyServer(function(input, output, session) {  
#   output$plot <- renderPlot({  
#     progress <- shiny::Progress$new(session, min=1, max=15)  
#     on.exit(progress$close())  
#  
#     progress$set(message = 'Calculation in progress',  
#                 detail = 'This may take a while...')  
#  
#     for (i in 1:15) {  
#       progress$set(value = i)  
#       Sys.sleep(0.5)  
#     }  
#     plot(cars)  
#   })  
# })  
## <strong>End(Not run)</strong>
```

## See also

[withProgress](#)

## 3.57 Reporting progress (functional API)

# Reporting progress (functional API)

```
withProgress(expr, min = 0, max = 1, value = min + (max - min) * 0.1, message = NULL,
  detail = NULL, session = getDefaultReactiveDomain(), env = parent.frame(),
  quoted = FALSE)
```

```
setProgress(value = NULL, message = NULL, detail = NULL,
  session = getDefaultReactiveDomain())
```

```
incProgress(amount = 0.1, message = NULL, detail = NULL,
  session = getDefaultReactiveDomain())
```

## Arguments

<code>expr</code>	The work to be done. This expression should contain calls to <code>setProgress</code> .
<code>min</code>	The value that represents the starting point of the progress bar. Must be less than <code>max</code> . Default is 0.
<code>max</code>	The value that represents the end of the progress bar. Must be greater than <code>min</code> . Default is 1.
<code>value</code>	Single-element numeric vector; the value at which to set the progress bar, relative to <code>min</code> and <code>max</code> . <code>NULL</code> hides the progress bar, if it is currently visible.
<code>message</code>	A single-element character vector; the message to be displayed to the user, or <code>NULL</code> to hide the current message (if any).
<code>detail</code>	A single-element character vector; the detail message to be displayed to the user, or <code>NULL</code> to hide the current detail message (if any). The detail message will be shown with a de-emphasized appearance relative to <code>message</code> .
<code>session</code>	The Shiny session object, as provided by <code>shinyServer</code> to the server function. The default is to automatically find the session by using the current reactive domain.
<code>env</code>	The environment in which <code>expr</code> should be evaluated.
<code>quoted</code>	Whether <code>expr</code> is a quoted expression (this is not common).
<code>amount</code>	For <code>incProgress</code> , the amount to increment the status bar. Default is 0.1.

## Description

Reports progress to the user during long-running operations.

## Details

This package exposes two distinct programming APIs for working with progress. Using `withProgress` with `incProgress` or `setProgress` provide a simple function-based interface, while the [Progress](#) reference class provides an object-oriented API.

Use `withProgress` to wrap the scope of your work; doing so will cause a new progress panel to be created, and it will be displayed the first time `incProgress` or `setProgress` are called. When `withProgress` exits, the corresponding progress panel will be removed.

The `incProgress` function increments the status bar by a specified amount, whereas the `setProgress` function sets it to a specific value, and can also set the text displayed.

Generally, `withProgress` / `incProgress` / `setProgress` should be sufficient; the exception is if the work to be done is asynchronous (this is not common) or otherwise cannot be encapsulated by a single scope. In that case, you can use the `Progress` reference class.

## Examples

```
## <strong>Not run</strong>:
# # server.R
# shinyServer(function(input, output) {
#   output$plot <- renderPlot({
#     withProgress(message = 'Calculation in progress',
#                 detail = 'This may take a while...', value = 0, {
#       for (i in 1:15) {
#         incProgress(1/15)
#         Sys.sleep(0.25)
#       }
#     })
#   plot(cars)
# })
# ## <strong>End(Not run)</strong>
```

## See also

[Progress](#)

## 3.58 HTML Builder Functions

# HTML Builder Functions

`tags`

`p(...)`

`h1(...)`

`h2(...)`

`h3(...)`

`h4(...)`

`h5(...)`

`h6(...)`

`a(...)`

`br(...)`

`div(...)`

`span(...)`

`pre(...)`

`code(...)`

`img(...)`

`strong(...)`

`em(...)`

`hr(...)`

## Arguments

... Attributes and children of the element. Named arguments become attributes, and positional arguments become children. Valid children are tags, single-character character vectors (which become text nodes), and raw HTML (see [HTML](#)). You can also pass lists that contain tags, text nodes, and HTML.

# Description

Simple functions for constructing HTML documents.

## Details

The `tags` environment contains convenience functions for all valid HTML5 tags. To generate tags that are not part of the HTML5 specification, you can use the `tag()` function.

Dedicated functions are available for the most common HTML tags that do not conflict with common R functions.

The result from these functions is a tag object, which can be converted using `as.character()`.

## Examples

```
doc <- tags$html(  
  tags$head(  
    tags$title('My first page')  
  ),  
  tags$body(  
    h1('My first heading'),  
    p('My first paragraph, with some ',  
      strong('bold'),  
      ' text. '),  
    div(id='myDiv', class='simpleDiv',  
        'Here is a div with some attributes. ')  
  )  
)  
cat(as.character(doc))  
  
<html>  
  <body>  
    <h1>My first heading</h1>  
    <p>  
      My first paragraph, with some  
      <strong>bold</strong>  
      text.  
    </p>  
    <div id="myDiv" class="simpleDiv">Here is a div with some attributes.</div>  
  </body>  
</html>
```



### 3.59 Mark Characters as HTML

# Mark Characters as HTML

`HTML(text, ...)`

## Arguments

`text` The text value to mark with HTML

`...` Any additional values to be converted to character and concatenated together

## Value

The same value, but marked as HTML.

## Description

Marks the given text as HTML, which means the `tag` functions will know not to perform HTML escaping on it.

## Examples

```
e1 <- div(HTML("I like <u>turtles</u>"))  
cat(as.character(e1))
```

```
<div>I like <u>turtles</u></div>
```

### 3.60 Include Content From a File

# Include Content From a File

`includeHTML(path)`

`includeText(path)`

`includeMarkdown(path)`

`includeCSS(path, ...)`

`includeScript(path, ...)`

## Arguments

`path` The path of the file to be included. It is highly recommended to use a relative path (the base path being the Shiny application directory), not an absolute path.

`...` Any additional attributes to be applied to the generated tag.

## Description

Load HTML, text, or rendered Markdown from a file and turn into HTML.

## Details

These functions provide a convenient way to include an extensive amount of HTML, textual, Markdown, CSS, or JavaScript content, rather than using a large literal R string.

## Note

`includeText` escapes its contents, but does no other processing. This means that hard breaks and multiple spaces will be rendered as they usually are in HTML: as a single space character. If you are looking for preformatted text, wrap the call with `pre`, or consider using `includeMarkdown` instead.

The `includeMarkdown` function requires the `markdown` package.

### 3.61 Include content only once

# Include content only once

```
singleton(x, value = TRUE)
```

```
is.singleton(x)
```

## Arguments

`x` A `tag`, text, `HTML`, or list.

`value` Whether the object should be a singleton.

## Description

Use `singleton` to wrap contents (tag, text, HTML, or lists) that should be included in the generated document only once, yet may appear in the document-generating code more than once. Only the first appearance of the content (in document order) will be used.

## 3.62 HTML Tag Object

# HTML Tag Object

```
tagList(...)
```

```
tagAppendAttributes(tag, ...)
```

```
tagAppendChild(tag, child)
```

```
tagAppendChildren(tag, ..., list = NULL)
```

```
tagSetChildren(tag, ..., list = NULL)
```

```
tag(`_tag_name`, varArgs)
```

## Arguments

`_tag_name` HTML tag name

`varArgs` List of attributes and children of the element. Named list items become attributes, and unnamed list items become children. Valid children are tags, single-character character vectors (which become text nodes), and raw HTML (see [HTML](#)). You can also pass lists that contain tags, text nodes, and HTML.

`tag` A tag to append child elements to.

`child` A child element to append to a parent tag.

`...` Unnamed items that comprise this list of tags.

`list` An optional list of elements. Can be used with or instead of the `...` items.

## Value

An HTML tag object that can be rendered as HTML using `as.character()`.

## Description

`tag()` creates an HTML tag definition. Note that all of the valid HTML5 tags are already defined in the `tags` environment so these functions should only be used to generate additional tags. `tagAppendChild()` and `tagList()` are for supporting package authors who wish to create their own sets of tags; see the contents of `bootstrap.R` for examples.

## Examples

```
tagList(tags$h1("Title"),
        tags$h2("Header text"),
        tags$p("Text here"))
```

```
<h1>Title</h1>
<h2>Header text</h2>
<p>Text here</p>
```

```
# Can also convert a regular list to a tagList (internal data structure isn't
# exactly the same, but when rendered to HTML, the output is the same).
```

```
x <- list(tags$h1("Title"),
         tags$h2("Header text"),
         tags$p("Text here"))
tagList(x)
```

```
<h1>Title</h1>
<h2>Header text</h2>
<p>Text here</p>
```

### 3.63 Validate proper CSS formatting of a

# Validate proper CSS formatting of a unit

```
validateCssUnit(x)
```

## Arguments

`x` The unit to validate. Will be treated as a number of pixels if a unit is not specified.

## Value

A properly formatted CSS unit of length, if possible. Otherwise, will throw an error.

## Description

Checks that the argument is valid for use as a CSS unit of length.

## Details

NULL and NA are returned unchanged.

Single element numeric vectors are returned as a character vector with the number plus a suffix of "px" .

Single element character vectors must be "auto" or "inherit" , or a number. If the number has a suffix, it must be valid: px , % , em , pt , in , cm , mm , ex , or pc . If the number has no suffix, the suffix "px" is appended.

Any other value will cause an error to be thrown.

## Examples

```
validateCssUnit("10%")
```

```
[1] "10%"
```

```
validateCssUnit(400) #treated as '400px'
```

```
[1] "400px"
```

### 3.64 Evaluate an expression using tags

# Evaluate an expression using tags

```
withTags(code)
```

## Arguments

`code` A set of tags.

## Description

This function makes it simpler to write HTML-generating code. Instead of needing to specify `tags` each time a tag function is used, as in `tags$div()` and `tags$p()`, code inside `withTags` is evaluated with `tags` searched first, so you can simply use `div()` and `p()`.

## Details

If your code uses an object which happens to have the same name as an HTML tag function, such as `source()` or `summary()`, it will call the tag function. To call the intended (non-tags function), specify the namespace, as in `base::source()` or `base::summary()`.

## Examples

```
# Using tags$ each time
tags$div(class = "myclass",
  tags$h3("header"),
  tags$p("text")
)
```

```
<div class="myclass">
  <h3>header</h3>
  <p>text</p>
</div>
```

```
# Equivalent to above, but using withTags
withTags(
  div(class = "myclass",
    h3("header"),
    p("text")
  )
)
```

```
<div class="myclass">
  <h3>header</h3>
```

```
<p>text</p>
```

```
</div>
```

Shiny is an [RStudio](#) project. © 2014 RStudio, Inc.



## 3.65 Plot Output

# Plot Output

```
renderPlot(expr, width = "auto", height = "auto", res = 72, ..., env = parent.frame(),  
           quoted = FALSE, func = NULL)
```

## Arguments

<code>expr</code>	An expression that generates a plot.
<code>width,height</code>	The width/height of the rendered plot, in pixels; or <code>'auto'</code> to use the <code>offsetWidth / offsetHeight</code> of the HTML element that is bound to this plot. You can also pass in a function that returns the width/height in pixels or <code>'auto'</code> ; in the body of the function you may reference reactive values and functions. When rendering an inline plot, you must provide numeric values (in pixels) to both <code>width</code> and <code>height</code> .
<code>res</code>	Resolution of resulting plot, in pixels per inch. This value is passed to <code>png</code> . Note that this affects the resolution of PNG rendering in R; it won't change the actual ppi of the browser.
<code>...</code>	Arguments to be passed through to <code>png</code> . These can be used to set the width, height, background color, etc.
<code>env</code>	The environment in which to evaluate <code>expr</code> .
<code>quoted</code>	Is <code>expr</code> a quoted expression (with <code>quote()</code> )? This is useful if you want to save an expression in a variable.
<code>func</code>	A function that generates a plot (deprecated; use <code>expr</code> instead).

## Description

Renders a reactive plot that is suitable for assigning to an `output` slot.

## Details

The corresponding HTML output tag should be `div` or `img` and have the CSS class name `shiny-plot-output`.

## See also

For the corresponding client-side output function, and example usage, see `plotOutput`. For more details on how the plots are generated, and how to control the output, see `plotPNG`.

## 3.66 Text Output

# Text Output

```
renderText(expr, env = parent.frame(), quoted = FALSE, func = NULL)
```

## Arguments

- `expr` An expression that returns an R object that can be used as an argument to `cat` .
- `env` The environment in which to evaluate `expr` .
- `quoted` Is `expr` a quoted expression (with `quote()` )? This is useful if you want to save an expression in a variable.
- `func` A function that returns an R object that can be used as an argument to `cat` .(deprecated; use `expr` instead).

## Description

Makes a reactive version of the given function that also uses `cat` to turn its result into a single-element character vector.

## Details

The corresponding HTML output tag can be anything (though `pre` is recommended if you need a monospace font and whitespace preserved) and should have the CSS class name `shiny-text-output` .

The result of executing `func` will be passed to `cat` , inside a `capture.output` call.

## Examples

```
isolate({

# renderPrint captures any print output, converts it to a string, and
# returns it
visFun <- renderPrint({ "foo" })
visFun()
# '[1] "foo"'

invisFun <- renderPrint({ invisible("foo") })
invisFun()
# ''

multiprintFun <- renderPrint({
  print("foo");
  "bar"
})
```

```
multiprintFun()
# '[1] "foo"\n[1] "bar"'

nullFun <- renderPrint({ NULL })
nullFun()
# 'NULL'

invisNullFun <- renderPrint({ invisible(NULL) })
invisNullFun()
# ''

vecFun <- renderPrint({ 1:5 })
vecFun()
# '[1] 1 2 3 4 5'

# Contrast with renderText, which takes the value returned from the function
# and uses cat() to convert it to a string
visFun <- renderText({ "foo" })
visFun()
# 'foo'

invisFun <- renderText({ invisible("foo") })
invisFun()
# 'foo'

multiprintFun <- renderText({
  print("foo");
  "bar"
})
multiprintFun()
# 'bar'

nullFun <- renderText({ NULL })
nullFun()
# ''

invisNullFun <- renderText({ invisible(NULL) })
invisNullFun()
# ''

vecFun <- renderText({ 1:5 })
vecFun()
# '1 2 3 4 5'

})

[1] "foo"

[1] "1 2 3 4 5"
```

## See also

[renderPrint](#) for capturing the print output of a function, rather than the returned text value.

## 3.67 Printable Output

# Printable Output

```
renderPrint(expr, env = parent.frame(), quoted = FALSE, func = NULL,
            width = getOption("width"))
```

## Arguments

`expr` An expression that may print output and/or return a printable R object.

`env` The environment in which to evaluate `expr`.

`quoted` Is `expr` a quoted expression (with `quote()`)? This

`func` A function that may print output and/or return a printable R object (deprecated; use `expr` instead).

`width` The value for `options('width')`.

## Description

Makes a reactive version of the given function that captures any printed output, and also captures its printable result (unless `invisible()`), into a string. The resulting function is suitable for assigning to an `output` slot.

## Details

The corresponding HTML output tag can be anything (though `pre` is recommended if you need a monospace font and whitespace preserved) and should have the CSS class name `shiny-text-output`.

The result of executing `func` will be printed inside a `capture.output` call.

Note that unlike most other Shiny output functions, if the given function returns `NULL` then `NULL` will actually be visible in the output. To display nothing, make your function return `invisible()`.

## Examples

```
isolate({

# renderPrint captures any print output, converts it to a string, and
# returns it
visFun <- renderPrint({ "foo" })
visFun()
# '[1] "foo"'

invisFun <- renderPrint({ invisible("foo") })
invisFun()
# ''

multiprintFun <- renderPrint({
```

```
  print("foo");
  "bar"
})
multiprintFun()
# '[1] "foo"\n[1] "bar"'

nullFun <- renderPrint({ NULL })
nullFun()
# 'NULL'

invisNullFun <- renderPrint({ invisible(NULL) })
invisNullFun()
# ''

vecFun <- renderPrint({ 1:5 })
vecFun()
# '[1] 1 2 3 4 5'

# Contrast with renderText, which takes the value returned from the function
# and uses cat() to convert it to a string
visFun <- renderText({ "foo" })
visFun()
# 'foo'

invisFun <- renderText({ invisible("foo") })
invisFun()
# 'foo'

multiprintFun <- renderText({
  print("foo");
  "bar"
})
multiprintFun()
# 'bar'

nullFun <- renderText({ NULL })
nullFun()
# ''

invisNullFun <- renderText({ invisible(NULL) })
invisNullFun()
# ''

vecFun <- renderText({ 1:5 })
vecFun()
# '1 2 3 4 5'

})

[1] "foo"

[1] "1 2 3 4 5"
```

## See also

[renderText](#) for displaying the value returned from a function, instead of the printed output.

### 3.68 Table output with the JavaScript

# Table output with the JavaScript library DataTables

```
renderDataTable(expr, options = NULL, searchDelay = 500, callback = "function(oTable) {}",  
  escape = TRUE, env = parent.frame(), quoted = FALSE)
```

## Arguments

<code>expr</code>	An expression that returns a data frame or a matrix.
<code>options</code>	A list of initialization options to be passed to DataTables, or a function to return such a list.
<code>searchDelay</code>	The delay for searching, in milliseconds (to avoid too frequent search requests).
<code>callback</code>	A JavaScript function to be applied to the DataTable object. This is useful for DataTables plug-ins, which often require the DataTable instance to be available ( <a href="http://datatables.net/extensions/">http://datatables.net/extensions/</a> ).
<code>escape</code>	Whether to escape HTML entities in the table: <code>TRUE</code> means to escape the whole table, and <code>FALSE</code> means not to escape it. Alternatively, you can specify numeric column indices or column names to indicate which columns to escape, e.g. <code>1:5</code> (the first 5 columns), <code>c(1, 3, 4)</code> , or <code>c(-1, -3)</code> (all columns except the first and third), or <code>c('Species', 'Sepal.Length')</code> .
<code>env</code>	The environment in which to evaluate <code>expr</code> .
<code>quoted</code>	Is <code>expr</code> a quoted expression (with <code>quote()</code> )? This is useful if you want to save an expression in a variable.

## Description

Makes a reactive version of the given function that returns a data frame (or matrix), which will be rendered with the DataTables library. Paging, searching, filtering, and sorting can be done on the R side using Shiny as the server infrastructure.

## Details

For the `options` argument, the character elements that have the class `"AsIs"` (usually returned from `I()`) will be evaluated in JavaScript. This is useful when the type of the option value is not supported in JSON, e.g., a JavaScript function, which can be obtained by evaluating a character string. Note this only applies to the root-level elements of the options list, and the `I()` notation does not work for lower-level elements in the list.

## Note

This function only provides the server-side version of DataTables (using R to process the data object on the server side). There is a separate package `DT` (<https://github.com/rstudio/DT>) that allows you to create both server-side and client-side DataTables, and supports additional DataTables features. Consider using `DT::renderDataTable()` and

`DT::dataTableOutput()` (see <http://rstudio.github.io/DI/shiny.html> for more information).

## References

<http://datatables.net>

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  # pass a callback function to DataTables using I()
  shinyApp(
    ui = fluidPage(
      fluidRow(
        column(12,
          dataTableOutput('table')
        )
      )
    ),
    server = function(input, output) {
      output$table <- renderDataTable(iris,
        options = list(
          pageLength = 5,
          initComplete = I("function(settings, json) {alert('Done.')}")
        )
      )
    }
  )
}
```

## 3.69 Image file output

# Image file output

```
renderImage(expr, env = parent.frame(), quoted = FALSE, deleteFile = TRUE)
```

## Arguments

- `expr` An expression that returns a list.
- `env` The environment in which to evaluate `expr`.
- `quoted` Is `expr` a quoted expression (with `quote()`)? This is useful if you want to save an expression in a variable.
- `deleteFile` Should the file in `func()$src` be deleted after it is sent to the client browser? Generally speaking, if the image is a temp file generated within `func`, then this should be `TRUE`; if the image is not a temp file, this should be `FALSE`.

## Description

Renders a reactive image that is suitable for assigning to an `output` slot.

## Details

The expression `expr` must return a list containing the attributes for the `img` object on the client web page. For the image to display, properly, the list must have at least one entry, `src`, which is the path to the image file. It may also be useful to have a `contentType` entry specifying the MIME type of the image. If one is not provided, `renderImage` will try to autodetect the type, based on the file extension.

Other elements such as `width`, `height`, `class`, and `alt`, can also be added to the list, and they will be used as attributes in the `img` object.

The corresponding HTML output tag should be `div` or `img` and have the CSS class name `shiny-image-output`.

## Examples

```
## <strong>Not run</strong>:  
#  
# shinyServer(function(input, output, clientData) {  
#  
#   # A plot of fixed size  
#   output$plot1 <- renderImage({  
#     # A temp file to save the output. It will be deleted after renderImage  
#     # sends it, because deleteFile=TRUE.  
#     outfile <- tempfile(fileext='.png')  
#  
#     # Generate a png
```



*Image file output*

```

#   png(outfile, width=400, height=400)
#   hist(rnorm(input$n))
#   dev.off()
#
#   # Return a list
#   list(src = outfile,
#         alt = "This is alternate text")
# }, deleteFile = TRUE)
#
#   # A dynamically-sized plot
#   output$plot2 <- renderImage({
#     # Read plot2's width and height. These are reactive values, so this
#     # expression will re-run whenever these values change.
#     width <- clientData$output_plot2_width
#     height <- clientData$output_plot2_height
#
#     # A temp file to save the output.
#     outfile <- tempfile(fileext='.png')
#
#     png(outfile, width=width, height=height)
#     hist(rnorm(input$obs))
#     dev.off()
#
#     # Return a list containing the filename
#     list(src = outfile,
#          width = width,
#          height = height,
#          alt = "This is alternate text")
#   }, deleteFile = TRUE)
#
#   # Send a pre-rendered image, and don't delete the image after sending it
#   output$plot3 <- renderImage({
#     # When input$n is 1, filename is ./images/image1.jpeg
#     filename <- normalizePath(file.path('./images',
#                                         paste('image', input$n, '.jpeg', sep='')))
#
#     # Return a list containing the filename
#     list(src = filename)
#   }, deleteFile = FALSE)
# })
#
# ## <strong>End(Not run)</strong>

```

## See also

For more details on how the images are generated, and how to control the output, see [plotPNG](#).

## 3.70 Table Output

# Table Output

```
renderTable(expr, ..., env = parent.frame(), quoted = FALSE, func = NULL)
```

## Arguments

- `expr` An expression that returns an R object that can be used with `xtable`.
- `...` Arguments to be passed through to `xtable` and `print.xtable`.
- `env` The environment in which to evaluate `expr`.
- `quoted` Is `expr` a quoted expression (with `quote()`)? This is useful if you want to save an expression in a variable.
- `func` A function that returns an R object that can be used with `xtable` (deprecated; use `expr` instead).

## Description

Creates a reactive table that is suitable for assigning to an `output` slot.

## Details

The corresponding HTML output tag should be `div` and have the CSS class name `shiny-html-output`.

## 3.71 UI Output

# UI Output

```
renderUI(expr, env = parent.frame(), quoted = FALSE, func = NULL)
```

## Arguments

- `expr` An expression that returns a Shiny tag object, [HTML](#), or a list of such objects.
- `env` The environment in which to evaluate `expr`.
- `quoted` Is `expr` a quoted expression (with `quote()`)? This is useful if you want to save an expression in a variable.
- `func` A function that returns a Shiny tag object, [HTML](#), or a list of such objects (deprecated; use `expr` instead).

## Description

Experimental feature. Makes a reactive version of a function that generates HTML using the Shiny UI library.

## Details

The corresponding HTML output tag should be `div` and have the CSS class name `shiny-html-output` (or use [uiOutput](#)).

## Examples

```
## <strong>Not run</strong>:  
# output$moreControls <- renderUI({  
#   list(  
#  
#   )  
# })  
# ## <strong>End(Not run)</strong>
```

## See also

[conditionalPanel](#)

## 3.72 File Downloads

# File Downloads

```
downloadHandler(filename, content, contentType = NA)
```

## Arguments

- filename** A string of the filename, including extension, that the user's web browser should default to when downloading the file; or a function that returns such a string. (Reactive values and functions may be used from this function.)
- content** A function that takes a single argument `file` that is a file path (string) of a nonexistent temp file, and writes the content to that file path. (Reactive values and functions may be used from this function.)
- contentType** A string of the download's [content type](#), for example `"text/csv"` or `"image/png"`. If `NULL` or `NA`, the content type will be guessed based on the filename extension, or `application/octet-stream` if the extension is unknown.

## Description

Allows content from the Shiny application to be made available to the user as file downloads (for example, downloading the currently visible data as a CSV file). Both filename and contents can be calculated dynamically at the time the user initiates the download. Assign the return value to a slot on `output` in your server function, and in the UI use [downloadButton](#) or [downloadLink](#) to make the download available.

## Examples

```
## <strong>Not run</strong>:  
## In server.R:  
# output$downloadData <- downloadHandler(  
#   filename = function() {  
#     paste('data-', Sys.Date(), '.csv', sep='')  
#   },  
#   content = function(file) {  
#     write.csv(data, file)  
#   }  
# )  
## In ui.R:  
# downloadLink('downloadData', 'Download')  
## <strong>End(Not run)</strong>
```

### 3.73 Plot output (deprecated)

# Plot output (deprecated)

```
reactivePlot(func, width = "auto", height = "auto", ...)
```

## Arguments

`func` A function.

`width` Width.

`height` Height.

`...` Other arguments to pass on.

## Description

See `renderPlot` .

### 3.74 Print output (deprecated)

# Print output (deprecated)

`reactivePrint(func)`

## Arguments

`func` A function.

## Description

See `renderPrint` .

### 3.75 Table output (deprecated)

# Table output (deprecated)

`reactiveTable(func, ...)`

## Arguments

`func` A function.

`...` Other arguments to pass on.

## Description

See [renderTable](#).

## 3.76 Text output (deprecated)

# Text output (deprecated)

`reactiveText(func)`

## Arguments

`func` A function.

## Description

See `renderText` .



## 3.77 UI output (deprecated)

# UI output (deprecated)

`reactiveUI` (func)

## Arguments

`func` A function.

## Description

See `renderUI` .

### 3.78 Scheduled Invalidation

# Scheduled Invalidation

```
invalidateLater(millis, session)
```

## Arguments

- millis**     Approximate milliseconds to wait before invalidating the current reactive context.
- session**    A session object. This is needed to cancel any scheduled invalidations after a user has ended the session. If `NULL`, then this invalidation will not be tied to any session, and so it will still occur.

## Description

Schedules the current reactive context to be invalidated in the given number of milliseconds.

## Details

If this is placed within an observer or reactive expression, that object will be invalidated (and re-execute) after the interval has passed. The re-execution will reset the invalidation flag, so in a typical use case, the object will keep re-executing and waiting for the specified interval. It's possible to stop this cycle by adding conditional logic that prevents the `invalidateLater` from being run.

## Examples

```
## <strong>Not run</strong>:
# shinyServer(function(input, output, session) {
#
#   observe({
#     # Re-execute this reactive expression after 1000 milliseconds
#     invalidateLater(1000, session)
#
#     # Do something each time this is invalidated.
#     # The isolate() makes this observer _not_ get invalidated and re-executed
#     # when input$n changes.
#     print(paste("The value of input$n is", isolate(input$n)))
#   })
#
#   # Generate a new histogram at timed intervals, but not when
#   # input$n changes.
#   output$plot <- renderPlot({
#     # Re-execute this reactive expression after 2000 milliseconds
#     invalidateLater(2000, session)
#     hist(isolate(input$n))
#   })
# }
```

```
# })  
# ## <strong>End(Not run)</strong>
```

## See also

[reactiveTimer](#) is a slightly less safe alternative.

### 3.79 Checks whether an object is a

# Checks whether an object is a reactivevalues object

`is.reactivevalues(x)`

## Arguments

`x` The object to test.

## Description

Checks whether its argument is a reactivevalues object.

## See also

[reactiveValues](#).

### 3.80 Create a non-reactive scope for an

# Create a non-reactive scope for an expression

```
isolate(expr)
```

## Arguments

`expr` An expression that can access reactive values or expressions.

## Description

Executes the given expression in a scope where reactive values or expression can be read, but they cannot cause the reactive scope of the caller to be re-evaluated when they change.

## Details

Ordinarily, the simple act of reading a reactive value causes a relationship to be established between the caller and the reactive value, where a change to the reactive value will cause the caller to re-execute. (The same applies for the act of getting a reactive expression's value.) The `isolate` function lets you read a reactive value or expression without establishing this relationship.

The expression given to `isolate()` is evaluated in the calling environment. This means that if you assign a variable inside the `isolate()`, its value will be visible outside of the `isolate()`. If you want to avoid this, you can use `local()` inside the `isolate()`.

This function can also be useful for calling reactive expression at the console, which can be useful for debugging. To do so, simply wrap the calls to the reactive expression with `isolate()`.

## Examples

```
## <strong>Not run</strong>:  
# observe({  
#   input$saveButton # Do take a dependency on input$saveButton  
#  
#   # isolate a simple expression  
#   data <- get(isolate(input$dataset)) # No dependency on input$dataset  
#   writeToDatabase(data)  
# })  
#  
# observe({  
#   input$saveButton # Do take a dependency on input$saveButton  
#  
#   # isolate a whole block
```

```
# data <- isolate({
#   a <- input$valueA # No dependency on input$valueA or input$valueB
#   b <- input$valueB
#   c(a=a, b=b)
# })
# writeToDatabase(data)
# })
#
# observe({
#   x <- 1
#   # x outside of isolate() is affected
#   isolate(x <- 2)
#   print(x) # 2
#
#   y <- 1
#   # Use local() to avoid affecting calling environment
#   isolate(local(y <- 2))
#   print(y) # 1
# })
#
# ## <strong>End(Not run)</strong>

# Can also use isolate to call reactive expressions from the R console
values <- reactiveValues(A=1)
fun <- reactive({ as.character(values$A) })
isolate(fun())

[1] "1"

# "1"

# isolate also works if the reactive expression accesses values from the
# input object, like input$x
```

### 3.81 Make a reactive variable

# Make a reactive variable

```
makeReactiveBinding(symbol, env = parent.frame())
```

## Arguments

`symbol` A character string indicating the name of the variable that should be made reactive

`env` The environment that will contain the reactive variable

## Value

None.

## Description

Turns a normal variable into a reactive variable, that is, one that has reactive semantics when assigned or read in the usual ways. The variable may already exist; if so, its value will be used as the initial value of the reactive variable (or `NULL` if the variable did not exist).

## Examples

```
## <strong>Not run</strong>:  
# a <- 10  
# makeReactiveBinding("a")  
# b <- reactive(a * -1)  
# observe(print(b()))  
# a <- 20  
# ## <strong>End(Not run)</strong>
```

## 3.82 Create a reactive observer

# Create a reactive observer

```
observe(x, env = parent.frame(), quoted = FALSE, label = NULL, suspended = FALSE,  
       priority = 0, domain = getDefaultReactiveDomain(), autoDestroy = TRUE)
```

## Arguments

<code>x</code>	An expression (quoted or unquoted). Any return value will be ignored.
<code>env</code>	The parent environment for the reactive expression. By default, this is the calling environment, the same as when defining an ordinary non-reactive expression.
<code>quoted</code>	Is the expression quoted? By default, this is <code>FALSE</code> . This is useful when you want to use an expression that is stored in a variable; to do so, it must be quoted with <code>quote()</code> .
<code>label</code>	A label for the observer, useful for debugging.
<code>suspended</code>	If <code>TRUE</code> , start the observer in a suspended state. If <code>FALSE</code> (the default), start in a non-suspended state.
<code>priority</code>	An integer or numeric that controls the priority with which this observer should be executed. An observer with a given priority level will always execute sooner than all observers with a lower priority level. Positive, negative, and zero values are allowed.
<code>domain</code>	See <a href="#">domains</a> .
<code>autoDestroy</code>	If <code>TRUE</code> (the default), the observer will be automatically destroyed when its domain (if any) ends.

## Value

An observer reference class object. This object has the following methods:

`suspend()`

Causes this observer to stop scheduling flushes (re-executions) in response to invalidations. If the observer was invalidated prior to this call but it has not re-executed yet then that re-execution will still occur, because the flush is already scheduled.

`resume()`

Causes this observer to start re-executing in response to invalidations. If the observer was invalidated while suspended, then it will schedule itself for re-execution.

`destroy()`

Stops the observer from executing ever again, even if it is currently scheduled for re-execution.

`setPriority(priority = 0)`

Change this observer's priority. Note that if the observer is currently invalidated, then the change in priority will not take effect until the next invalidation--unless the observer is also currently suspended, in which case the priority change will be effective upon resume.

`setAutoDestroy(autoDestroy)`



Sets whether this observer should be automatically destroyed when its domain (if any) ends. If `autoDestroy` is `TRUE` and the domain already ended, then `destroy()` is called immediately."

`onInvalidDate(callback)`

Register a callback function to run when this observer is invalidated. No arguments will be provided to the callback function when it is invoked.

## Description

Creates an observer from the given expression.

## Details

An observer is like a reactive expression in that it can read reactive values and call reactive expressions, and will automatically re-execute when those dependencies change. But unlike reactive expressions, it doesn't yield a result and can't be used as an input to other reactive expressions. Thus, observers are only useful for their side effects (for example, performing I/O).

Another contrast between reactive expressions and observers is their execution strategy. Reactive expressions use lazy evaluation; that is, when their dependencies change, they don't re-execute right away but rather wait until they are called by someone else. Indeed, if they are not called then they will never re-execute. In contrast, observers use eager evaluation; as soon as their dependencies change, they schedule themselves to re-execute.

Starting with Shiny 0.10.0, observers are automatically destroyed by default when the `domain` that owns them ends (e.g. when a Shiny session ends).

## Examples

```
values <- reactiveValues(A=1)

obsB <- observe({
  print(values$A + 1)
})

# Can use quoted expressions
obsC <- observe(quote({ print(values$A + 2) })), quoted = TRUE)

# To store expressions for later conversion to observe, use quote()
expr_q <- quote({ print(values$A + 3) })
obsD <- observe(expr_q, quoted = TRUE)

# In a normal Shiny app, the web client will trigger flush events. If you
# are at the console, you can force a flush with flushReact()
shiny:::flushReact()

[1] 2
[1] 3
[1] 4
```

## 3.83 Event handler

# Event handler

```
observeEvent(eventExpr, handlerExpr, event.env = parent.frame(), event.quoted = FALSE,
  handler.env = parent.frame(), handler.quoted = FALSE, label = NULL, suspended = FALSE,
  priority = 0, domain = getDefaultReactiveDomain(), autoDestroy = TRUE,
  ignoreNULL = TRUE)
```

```
eventReactive(eventExpr, valueExpr, event.env = parent.frame(), event.quoted = FALSE,
  value.env = parent.frame(), value.quoted = FALSE, label = NULL,
  domain = getDefaultReactiveDomain(), ignoreNULL = TRUE)
```

## Arguments

eventExpr	A (quoted or unquoted) expression that represents the event; this can be a simple reactive value like <code>input\$click</code> , a call to a reactive expression like <code>dataset()</code> , or even a complex expression inside curly braces
handlerExpr	The expression to call whenever <code>eventExpr</code> is invalidated. This should be a side-effect-producing action (the return value will be ignored). It will be executed within an <code>isolate</code> scope.
event.env	The parent environment for <code>eventExpr</code> . By default, this is the calling environment.
event.quoted	Is the <code>eventExpr</code> expression quoted? By default, this is <code>FALSE</code> . This is useful when you want to use an expression that is stored in a variable; to do so, it must be quoted with <code>quote()</code> .
handler.env	The parent environment for <code>handlerExpr</code> . By default, this is the calling environment.
handler.quoted	Is the <code>handlerExpr</code> expression quoted? By default, this is <code>FALSE</code> . This is useful when you want to use an expression that is stored in a variable; to do so, it must be quoted with <code>quote()</code> .
label	A label for the observer or reactive, useful for debugging.
suspended	If <code>TRUE</code> , start the observer in a suspended state. If <code>FALSE</code> (the default), start in a non-suspended state.
priority	An integer or numeric that controls the priority with which this observer should be executed. An observer with a given priority level will always execute sooner than all observers with a lower priority level. Positive, negative, and zero values are allowed.
domain	See <a href="#">domains</a> .
autoDestroy	If <code>TRUE</code> (the default), the observer will be automatically destroyed when its domain (if any) ends.
ignoreNULL	Whether the action should be triggered (or value calculated, in the case of <code>eventReactive</code> ) when the input is <code>NULL</code> . See <a href="#">Details</a> .
valueExpr	The expression that produces the return value of the <code>eventReactive</code> . It will be executed within an <code>isolate</code> scope.

<code>value.env</code>	The parent environment for <code>valueExpr</code> . By default, this is the calling environment.
<code>value.quoted</code>	Is the <code>valueExpr</code> expression quoted? By default, this is <code>FALSE</code> . This is useful when you want to use an expression that is stored in a variable; to do so, it must be quoted with <code>quote()</code> .

## Value

`observeEvent` returns an observer reference class object (see [observe](#)). `eventReactive` returns a reactive expression object (see [reactive](#)).

## Description

Respond to "event-like" reactive inputs, values, and expressions.

## Details

Shiny's reactive programming framework is primarily designed for calculated values (reactive expressions) and side-effect-causing actions (observers) that respond to *any* of their inputs changing. That's often what is desired in Shiny apps, but not always: sometimes you want to wait for a specific action to be taken from the user, like clicking an [actionButton](#), before calculating an expression or taking an action. A reactive value or expression that is used to trigger other calculations in this way is called an *event*.

These situations demand a more imperative, "event handling" style of programming that is possible--but not particularly intuitive--using the reactive programming primitives [observe](#) and [isolate](#). `observeEvent` and `eventReactive` provide straightforward APIs for event handling that wrap `observe` and `isolate`.

Use `observeEvent` whenever you want to *perform an action* in response to an event. (Note that "recalculate a value" does not generally count as performing an action--see `eventReactive` for that.) The first argument is the event you want to respond to, and the second argument is a function that should be called whenever the event occurs.

Use `eventReactive` to create a *calculated value* that only updates in response to an event. This is just like a normal [reactive expression](#) except it ignores all the usual invalidations that come from its reactive dependencies; it only invalidates in response to the given event.

Both `observeEvent` and `eventReactive` take an `ignoreNULL` parameter that affects behavior when the `eventExpr` evaluates to `NULL` (or in the special case of an [actionButton](#), `0`). In these cases, if `ignoreNULL` is `TRUE`, then an `observeEvent` will not execute and an `eventReactive` will raise a silent [validation](#) error. This is useful behavior if you don't want to do the action or calculation when your app first starts, but wait for the user to initiate the action first (like a "Submit" button); whereas `ignoreNULL=FALSE` is desirable if you want to initially perform the action/calculation and just let the user re-initiate it (like a "Recalculate" button).

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  ui <- fluidPage(
    column(4,
      numericInput("x", "Value", 5),
      br(),
      actionButton("button", "Show")
    ),
    column(8, tableOutput("table"))
  )
  server <- function(input, output) {
    # Take an action every time button is pressed:
```

```
# here, we just print a message to the console
observeEvent(input$button, {
  cat("Showing", input$x, "rows\n")
})
# Take a reactive dependency on input$button, but
# not on any of the stuff inside the function
df <- eventReactive(input$button, {
  head(cars, input$x)
})
output$table <- renderTable({
  df()
})
}
shinyApp(ui=ui, server=server)
}
```

## See also

[actionButton](#)

### 3.84 Create a reactive expression

# Create a reactive expression

```
reactive(x, env = parent.frame(), quoted = FALSE, label = NULL,  
         domain = getDefaultReactiveDomain())
```

```
is.reactive(x)
```

## Arguments

- x** For `reactive`, an expression (quoted or unquoted). For `is.reactive`, an object to test.
- env** The parent environment for the reactive expression. By default, this is the calling environment, the same as when defining an ordinary non-reactive expression.
- quoted** Is the expression quoted? By default, this is `FALSE`. This is useful when you want to use an expression that is stored in a variable; to do so, it must be quoted with `quote()`.
- label** A label for the reactive expression, useful for debugging.
- domain** See [domains](#).

## Value

a function, wrapped in a S3 class "reactive"

## Description

Wraps a normal expression to create a reactive expression. Conceptually, a reactive expression is a expression whose result will change over time.

## Details

Reactive expressions are expressions that can read reactive values and call other reactive expressions. Whenever a reactive value changes, any reactive expressions that depended on it are marked as "invalidated" and will automatically re-execute if necessary. If a reactive expression is marked as invalidated, any other reactive expressions that recently called it are also marked as invalidated. In this way, invalidations ripple through the expressions that depend on each other.

See the [Shiny tutorial](#) for more information about reactive expressions.

## Examples

```
values <- reactiveValues(A=1)
```

```
reactiveB <- reactive({  
  values$A + 1
```

```
})

# Can use quoted expressions
reactiveC <- reactive(quote({ values$A + 2 }), quoted = TRUE)

# To store expressions for later conversion to reactive, use quote()
expr_q <- quote({ values$A + 3 })
reactiveD <- reactive(expr_q, quoted = TRUE)

# View the values from the R console with isolate()
isolate(reactiveB())

[1] 2

isolate(reactiveC())

[1] 3

isolate(reactiveD())

[1] 4
```

### 3.85 Reactive file reader

# Reactive file reader

```
reactiveFileReader(intervalMillis, session, filePath, readFunc, ...)
```

## Arguments

<code>intervalMillis</code>	Approximate number of milliseconds to wait between checks of the file's last modified time. This can be a numeric value, or a function that returns a numeric value.
<code>session</code>	The user session to associate this file reader with, or <code>NULL</code> if none. If non-null, the reader will automatically stop when the session ends.
<code>filePath</code>	The file path to poll against and to pass to <code>readFunc</code> . This can either be a single-element character vector, or a function that returns one.
<code>readFunc</code>	The function to use to read the file; must expect the first argument to be the file path to read. The return value of this function is used as the value of the reactive file reader.
<code>...</code>	Any additional arguments to pass to <code>readFunc</code> whenever it is invoked.

## Value

A reactive expression that returns the contents of the file, and automatically invalidates when the file changes on disk (as determined by last modified time).

## Description

Given a file path and read function, returns a reactive data source for the contents of the file.

## Details

`reactiveFileReader` works by periodically checking the file's last modified time; if it has changed, then the file is re-read and any reactive dependents are invalidated.

The `intervalMillis`, `filePath`, and `readFunc` functions will each be executed in a reactive context; therefore, they may read reactive values and reactive expressions.

## Examples

```
## <strong>Not run</strong>:  
# # Per-session reactive file reader  
# shinyServer(function(input, output, session) {  
#   fileData <- reactiveFileReader(1000, session, 'data.csv', read.csv)  
#  
#   output$data <- renderTable({  
#     fileData()  
#   })  
# }
```

```
#   })  
# }  
#  
# # Cross-session reactive file reader. In this example, all sessions share  
# # the same reader, so read.csv only gets executed once no matter how many  
# # user sessions are connected.  
# fileData <- reactiveFileReader(1000, session, 'data.csv', read.csv)  
# shinyServer(function(input, output, session) {  
#   output$data <- renderTable({  
#     fileData()  
#   })  
# }  
# ## <strong>End(Not run)</strong>
```

## See also

[reactivePoll](#)



## 3.86 Reactive polling

# Reactive polling

```
reactivePoll(intervalMillis, session, checkFunc, valueFunc)
```

## Arguments

<code>intervalMillis</code>	Approximate number of milliseconds to wait between calls to <code>checkFunc</code> . This can be either a numeric value, or a function that returns a numeric value.
<code>session</code>	The user session to associate this file reader with, or <code>NULL</code> if none. If non-null, the reader will automatically stop when the session ends.
<code>checkFunc</code>	A relatively cheap function whose values over time will be tested for equality; inequality indicates that the underlying value has changed and needs to be invalidated and re-read using <code>valueFunc</code> . See Details.
<code>valueFunc</code>	A function that calculates the underlying value. See Details.

## Value

A reactive expression that returns the result of `valueFunc`, and invalidates when `checkFunc` changes.

## Description

Used to create a reactive data source, which works by periodically polling a non-reactive data source.

## Details

`reactivePoll` works by pairing a relatively cheap "check" function with a more expensive value retrieval function. The check function will be executed periodically and should always return a consistent value until the data changes. When the check function returns a different value, then the value retrieval function will be used to re-populate the data.

Note that the check function doesn't return `TRUE` or `FALSE` to indicate whether the underlying data has changed. Rather, the check function indicates change by returning a different value from the previous time it was called.

For example, `reactivePoll` is used to implement `reactiveFileReader` by pairing a check function that simply returns the last modified timestamp of a file, and a value retrieval function that actually reads the contents of the file.

As another example, one might read a relational database table reactively by using a check function that does `SELECT MAX(timestamp) FROM table` and a value retrieval function that does `SELECT * FROM table`.

The `intervalMillis`, `checkFunc`, and `valueFunc` functions will be executed in a reactive context; therefore, they may read reactive values and reactive expressions.

## Examples

```
## <strong>NOT run</strong>:  
# # Assume the existence of readTimestamp and readValue functions  
# shinyServer(function(input, output, session) {  
#   data <- reactivePoll(1000, session, readTimestamp, readValue)  
#   output$dataTable <- renderTable({  
#     data()  
#   })  
# })  
# ## <strong>End(Not run)</strong>
```

## See also

[reactiveFileReader](#)

## 3.87 Timer

# Timer

```
reactiveTimer(intervalMs = 1000, session)
```

## Arguments

`intervalMs` How often to fire, in milliseconds

`session` A session object. This is needed to cancel any scheduled invalidations after a user has ended the session. If `NULL`, then this invalidation will not be tied to any session, and so it will still occur.

## Value

A no-parameter function that can be called from a reactive context, in order to cause that context to be invalidated the next time the timer interval elapses. Calling the returned function also happens to yield the current time (as in `Sys.time`).

## Description

Creates a reactive timer with the given interval. A reactive timer is like a reactive value, except reactive values are triggered when they are set, while reactive timers are triggered simply by the passage of time.

## Details

[Reactive expressions](#) and observers that want to be invalidated by the timer need to call the timer function that `reactiveTimer` returns, even if the current time value is not actually needed.

See [invalidateLater](#) as a safer and simpler alternative.

## Examples

```
## <strong>Not run</strong>:  
# shinyServer(function(input, output, session) {  
#  
# # Anything that calls autoInvalidate will automatically invalidate  
# # every 2 seconds.  
# autoInvalidate <- reactiveTimer(2000, session)  
#  
# observe({  
# # Invalidate and re-execute this reactive expression every time the  
# # timer fires.  
# autoInvalidate()  
#  
# # Do something each time this is invalidated.  
# # The isolate() makes this observer _not_ get invalidated and re-executed
```

*Timer*

```
# # when input$n changes.
# print(paste("The value of input$n is", isolate(input$n)))
# })
#
# # Generate a new histogram each time the timer fires, but not when
# # input$n changes.
# output$plot <- renderPlot({
#   autoInvalidate()
#   hist(isolate(input$n))
# })
# })
# ## <strong>End(Not run)</strong>
```

## See also

[invalidateLater](#)

## 3.88 Create an object for storing reactive

# Create an object for storing reactive values

```
reactiveValues(...)
```

## Arguments

... Objects that will be added to the reactivevalues object. All of these objects must be named.

## Description

This function returns an object for storing reactive values. It is similar to a list, but with special capabilities for reactive programming. When you read a value from it, the calling reactive expression takes a reactive dependency on that value, and when you write to it, it notifies any reactive functions that depend on that value. Note that values taken from the reactiveValues object are reactive, but the reactiveValues object itself is not.

## Examples

```
# Create the object with no values
values <- reactiveValues()

# Assign values to 'a' and 'b'
values$a <- 3
values[['b']] <- 4

## Not run:
# # From within a reactive context, you can access values with:
# values$a
# values[['a']]
# ## End(Not run)

# If not in a reactive context (e.g., at the console), you can use isolate()
# to retrieve the value:
isolate(values$a)

[1] 3

isolate(values[['a']])

[1] 3
```

Create an object for storing reactive

```
# Set values upon creation  
values <- reactiveValues(a = 1, b = 2)  
isolate(values$a)
```

```
[1] 1
```

## See also

`isolate` and `is.reactivevalues`.

### 3.89 Convert a reactivevalues object to a

# Convert a reactivevalues object to a list

```
reactiveValuesToList(x, all.names = FALSE)
```

## Arguments

`x` A reactivevalues object.

`all.names` If `TRUE`, include objects with a leading dot. If `FALSE` (the default) don't include those objects.

## Description

This function does something similar to what you might `as.list` to do. The difference is that the calling context will take dependencies on every object in the reactivevalues object. To avoid taking dependencies on all the objects, you can wrap the call with `isolate()`.

## Examples

```
values <- reactiveValues(a = 1)
## <strong>Not run</strong>:
# reactiveValuesToList(values)
# ## <strong>End(Not run)</strong>

# To get the objects without taking dependencies on them, use isolate().
# isolate() can also be used when calling from outside a reactive context (e.g.
# at the console)
isolate(reactiveValuesToList(values))

$a
[1] 1
```

### 3.90 Reactive domains

# Reactive domains

`getDefaultReactiveDomain()`

`withReactiveDomain(domain, expr)`

`onReactiveDomainEnded(domain, callback, failIfNull = FALSE)`

## Arguments

`domain` A valid domain object (for example, a Shiny session), or `NULL`

`expr` An expression to evaluate under `domain`

`callback` A callback function to be invoked

`failIfNull` If `TRUE` then an error is given if the `domain` is `NULL`

## Description

Reactive domains are a mechanism for establishing ownership over reactive primitives (like reactive expressions and observers), even if the set of reactive primitives is dynamically created. This is useful for lifetime management (i.e. destroying observers when the Shiny session that created them ends) and error handling.

## Details

At any given time, there can be either a single "default" reactive domain object, or none (i.e. the reactive domain object is `NULL`). You can access the current default reactive domain by calling `getDefaultReactiveDomain`.

Unless you specify otherwise, newly created observers and reactive expressions will be assigned to the current default domain (if any). You can override this assignment by providing an explicit `domain` argument to `reactive` or `observe`.

For advanced usage, it's possible to override the default domain using `withReactiveDomain`. The `domain` argument will be made the default domain while `expr` is evaluated.

Implementers of new reactive primitives can use `onReactiveDomainEnded` as a convenience function for registering callbacks. If the reactive domain is `NULL` and `failIfNull` is `FALSE`, then the callback will never be invoked.



## 3.91 Reactive Log Visualizer

# Reactive Log Visualizer

`showReactLog()`

## Description

Provides an interactive browser-based tool for visualizing reactive dependencies and execution in your application.

## Details

To use the reactive log visualizer, start with a fresh R session and run the command `options(shiny.reactLog=TRUE)` ; then launch your application in the usual way (e.g. using `runApp()`). At any time you can hit Ctrl+F3 (or for Mac users, Command+F3) in your web browser to launch the reactive log visualization.

The reactive log visualization only includes reactive activity up until the time the report was loaded. If you want to see more recent activity, refresh the browser.

Note that Shiny does not distinguish between reactive dependencies that "belong" to one Shiny user session versus another, so the visualization will include all reactive activity that has taken place in the process, not just for a particular application or session.

As an alternative to pressing Ctrl/Command+F3--for example, if you are using reactivities outside of the context of a Shiny application--you can run the `showReactLog` function, which will generate the reactive log visualization as a static HTML file and launch it in your default browser. In this case, refreshing your browser will not load new activity into the report; you will need to call `showReactLog()` explicitly.

For security and performance reasons, do not enable `shiny.reactLog` in production environments. When the option is enabled, it's possible for any user of your app to see at least some of the source code of your reactive expressions and observers.

## 3.92 Create a Shiny UI handler

# Create a Shiny UI handler

`shinyUI(ui)`

## Arguments

`ui` A user interface definition

## Value

The user interface definition, without modifications or side effects.

## Description

Historically this function was used in `ui.R` files to register a user interface with Shiny. It is no longer required as of Shiny 0.10; simply ensure that the last expression to be returned from `ui.R` is a user interface. This function is kept for backwards compatibility with older applications. It returns the value that is passed to it.

### 3.93 Define Server Functionality

# Define Server Functionality

`shinyServer(func)`

## Arguments

`func` The server function for this application. See the details section for more information.

## Description

Defines the server-side logic of the Shiny application. This generally involves creating functions that map user inputs to various kinds of output. In older versions of Shiny, it was necessary to call `shinyServer()` in the `server.R` file, but this is no longer required as of Shiny 0.10. Now the `server.R` file may simply return the appropriate server function (as the last expression in the code), without calling `shinyServer()`.

## Details

Call `shinyServer` from your application's `server.R` file, passing in a "server function" that provides the server-side logic of your application.

The server function will be called when each client (web browser) first loads the Shiny application's page. It must take an `input` and an `output` parameter. Any return value will be ignored. It also takes an optional `session` parameter, which is used when greater control is needed.

See the [tutorial](#) for more on how to write a server function.

## Examples

```
## <strong>Not run</strong>:
# # A very simple Shiny app that takes a message from the user
# # and outputs an uppercase version of it.
# shinyServer(function(input, output, session) {
#   output$uppercase <- renderText({
#     toupper(input$message)
#   })
# })
#
#
# # It is also possible for a server.R file to simply return the function,
# # without calling shinyServer().
# # For example, the server.R file could contain just the following:
# function(input, output, session) {
#   output$uppercase <- renderText({
#     toupper(input$message)
#   })
# }
```

```
# }  
# ## <strong>End(Not run)</strong>
```

Shiny is an RStudio project. © 2014 RStudio, Inc.

## 3.94 Run Shiny Application

# Run Shiny Application

```
runApp(appDir = getwd(), port = getOption("shiny.port"),
  launch.browser = getOption("shiny.launch.browser", interactive()),
  host = getOption("shiny.host", "127.0.0.1"), workerId = "", quiet = FALSE,
  display.mode = c("auto", "normal", "showcase"))
```

## Arguments

appDir	The directory of the application. Should contain <code>server.R</code> , plus, either <code>ui.R</code> or a <code>www</code> directory that contains the file <code>index.html</code> . Alternately, instead of <code>server.R</code> and <code>ui.R</code> , the directory may contain just <code>app.R</code> . Defaults to the working directory. Instead of a directory, this could be a list with <code>ui</code> and <code>server</code> components, or a Shiny app object created by <code>shinyApp</code> .
port	The TCP port that the application should listen on. If the <code>port</code> is not specified, and the <code>shiny.port</code> option is set (with <code>options(shiny.port = XX)</code> ), then that port will be used. Otherwise, use a random port.
launch.browser	If true, the system's default web browser will be launched automatically after the app is started. Defaults to true in interactive sessions only. This value of this parameter can also be a function to call with the application's URL.
host	The IPv4 address that the application should listen on. Defaults to the <code>shiny.host</code> option, if set, or <code>"127.0.0.1"</code> if not. See Details.
workerId	Can generally be ignored. Exists to help some editions of Shiny Server Pro route requests to the correct process.
quiet	Should Shiny status messages be shown? Defaults to FALSE.
display.mode	The mode in which to display the application. If set to the value <code>"showcase"</code> , shows application code and metadata from a <code>DESCRIPTION</code> file in the application directory alongside the application. If set to <code>"normal"</code> , displays the application normally. Defaults to <code>"auto"</code> , which displays the application in the mode given in its <code>DESCRIPTION</code> file, if any.

## Description

Runs a Shiny application. This function normally does not return; interrupt R to stop the application (usually by pressing Ctrl+C or Esc).

## Details

The `host` parameter was introduced in Shiny 0.9.0. Its default value of `"127.0.0.1"` means that, contrary to previous versions of Shiny, only the current machine can access locally hosted Shiny apps. To allow other clients to connect, use the value `"0.0.0.0"` instead (which was the value that was hard-coded into Shiny in 0.8.0 and earlier).

## Examples

```
## <strong>Not run</strong>:
# # Start app in the current working directory
# runApp()
#
# # Start app in a subdirectory called myapp
# runApp("myapp")
# ## <strong>End(Not run)</strong>

## Only run this example in interactive R sessions
if (interactive()) {
  # Apps can be run without a server.r and ui.r file
  runApp(list(
    ui = bootstrapPage(
      numericInput('n', 'Number of obs', 100),
      plotOutput('plot')
    ),
    server = function(input, output) {
      output$plot <- renderPlot({ hist(runif(input$n)) })
    }
  ))

  # Running a Shiny app object
  app <- shinyApp(
    ui = bootstrapPage(
      numericInput('n', 'Number of obs', 100),
      plotOutput('plot')
    ),
    server = function(input, output) {
      output$plot <- renderPlot({ hist(runif(input$n)) })
    }
  )
  runApp(app)
}
```

### 3.95 Run Shiny Example Applications

# Run Shiny Example Applications

```
runExample(example = NA, port = NULL, launch.browser = getOption("shiny.launch.browser",
  interactive()), host = getOption("shiny.host", "127.0.0.1"), display.mode = c("auto",
  "normal", "showcase"))
```

## Arguments

<code>example</code>	The name of the example to run, or <code>NA</code> (the default) to list the available examples.
<code>port</code>	The TCP port that the application should listen on. Defaults to choosing a random port.
<code>launch.browser</code>	If true, the system's default web browser will be launched automatically after the app is started. Defaults to true in interactive sessions only.
<code>host</code>	The IPv4 address that the application should listen on. Defaults to the <code>shiny.host</code> option, if set, or <code>"127.0.0.1"</code> if not.
<code>display.mode</code>	The mode in which to display the example. Defaults to <code>showcase</code> , but may be set to <code>normal</code> to see the example without code or commentary.

## Description

Launch Shiny example applications, and optionally, your system's web browser.

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  # List all available examples
  runExample()

  # Run one of the examples
  runExample("01_hello")

  # Print the directory containing the code for all examples
  system.file("examples", package="shiny")
}
```

### 3.96 Run a Shiny application from a URL

# Run a Shiny application from a URL

```
runUrl(url, filetype = NULL, subdir = NULL, destdir = NULL, ...)
```

```
runGist(gist, destdir = NULL, ...)
```

```
runGitHub(repo, username = getOption("github.user"), ref = "master", subdir = NULL,
  destdir = NULL, ...)
```

## Arguments

url	URL of the application.
filetype	The file type ( ".zip" , ".tar" , or ".tar.gz" ). Defaults to the file extension taken from the url.
subdir	A subdirectory in the repository that contains the app. By default, this function will run an app from the top level of the repo, but you can use a path such as ` "inst/shinyapp" ` .
destdir	Directory to store the downloaded application files. If <code>NULL</code> (the default), the application files will be stored in a temporary directory and removed when the app exits
...	Other arguments to be passed to <code>runApp()</code> , such as <code>port</code> and <code>launch.browser</code> .
gist	The identifier of the gist. For example, if the gist is <code>https://gist.github.com/jcheng5/3239667</code> , then <code>3239667</code> , <code>'3239667'</code> , and <code>'https://gist.github.com/jcheng5/3239667'</code> are all valid values.
repo	Name of the repository.
username	GitHub username. If <code>repo</code> is of the form <code>"username/repo"</code> , <code>username</code> will be taken from <code>repo</code> .
ref	Desired git reference. Could be a commit, tag, or branch name. Defaults to <code>"master"</code> .

## Description

`runUrl()` downloads and launches a Shiny application that is hosted at a downloadable URL. The Shiny application must be saved in a .zip, .tar, or .tar.gz file. The Shiny application files must be contained in the root directory or a subdirectory in the archive. For example, the files might be `myapp/server.r` and `myapp/ui.r` . The functions `runGitHub()` and `runGist()` are based on `runUrl()` , using URL's from GitHub (<https://github.com>) and GitHub gists (<https://gist.github.com>), respectively.

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  runUrl('https://github.com/rstudio/shiny_example/archive/master.tar.gz')

  # Can run an app from a subdirectory in the archive
  runUrl("https://github.com/rstudio/shiny_example/archive/master.zip",
```



```
  subdir = "inst/shinyapp/")
}
## Only run this example in interactive R sessions
if (interactive()) {
  runGist(3239667)
  runGist("https://gist.github.com/jcheng5/3239667")

  # Old URL format without username
  runGist("https://gist.github.com/3239667")
}
## Only run this example in interactive R sessions
if (interactive()) {
  runGitHub("shiny_example", "rstudio")
  # or runGitHub("rstudio/shiny_example")

  # Can run an app from a subdirectory in the repo
  runGitHub("shiny_example", "rstudio", subdir = "inst/shinyapp/")
}
```

### 3.97 Stop the currently running Shiny app

# Stop the currently running Shiny app

```
stopApp(returnValue = NULL)
```

## Arguments

`returnValue` The value that should be returned from `runApp`.

## Description

Stops the currently running Shiny app, returning control to the caller of `runApp`.

### 3.98 Create a web dependency

# Create a web dependency

```
createWebDependency(dependency)
```

## Arguments

`dependency` A single HTML dependency object, created using [htmlDependency](#). If the `src` value is named, then `href` and/or `file` names must be present.

## Value

A single HTML dependency object that has an `href`-named element in its `src`.

## Description

Ensure that a file-based HTML dependency (from the `htmltools` package) can be served over Shiny's HTTP server. This function works by using [addResourcePath](#) to map the HTML dependency's directory to a URL.

## 3.99 Resource Publishing

# Resource Publishing

```
addResourcePath(prefix, directoryPath)
```

## Arguments

- prefix** The URL prefix (without slashes). Valid characters are a-z, A-Z, 0-9, hyphen, period, and underscore; and must begin with a-z or A-Z. For example, a value of 'foo' means that any request paths that begin with '/foo' will be mapped to the given directory.
- directoryPath** The directory that contains the static resources to be served.

## Description

Adds a directory of static resources to Shiny's web server, with the given path prefix. Primarily intended for package authors to make supporting JavaScript/CSS files available to their components.

## Details

You can call `addResourcePath` multiple times for a given `prefix`; only the most recent value will be retained. If the normalized `directoryPath` is different than the directory that's currently mapped to the `prefix`, a warning will be issued.

## Examples

```
addResourcePath('datasets', system.file('data', package='datasets'))
```

## See also

[singleton](#)

### 3.100 Register an Input Handler

# Register an Input Handler

```
registerInputHandler(type, fun, force = FALSE)
```

## Arguments

- type** The type for which the handler should be added -- should be a single-element character vector.
- fun** The handler function. This is the function that will be used to parse the data delivered from the client before it is available in the `input` variable. The function will be called with the following three parameters:
1. The value of this input as provided by the client, deserialized using `jsonlite`.
  2. The `shinySession` in which the input exists.
  3. The name of the input.
- force** If `TRUE`, will overwrite any existing handler without warning. If `FALSE`, will throw an error if this class already has a handler defined.

## Description

Adds an input handler for data of this type. When called, Shiny will use the function provided to refine the data passed back from the client (after being deserialized by `jsonlite`) before making it available in the `input` variable of the `server.R` file.

## Details

This function will register the handler for the duration of the R process (unless Shiny is explicitly reloaded). For that reason, the `type` used should be very specific to this package to minimize the risk of colliding with another Shiny package which might use this data type name. We recommend the format of "packageName.widgetName".

Currently Shiny registers the following handlers: `shiny.matrix`, `shiny.number`, and `shiny.date`.

The `type` of a custom Shiny Input widget will be deduced using the `getType()` JavaScript function on the registered Shiny `inputBinding`.

## Examples

```
## <strong>Not run</strong>:  
# # Register an input handler which rounds a input number to the nearest integer  
# registerInputHandler("mypackage.validint", function(x, shinySession, name) {  
#   if (is.null(x)) return(NA)  
#   round(x)  
# })  
#  
# ## On the Javascript side, the associated input binding must have a corresponding getType method:  
# getType: function(e1) {
```

```
# return "mypackage.validate";  
# }  
#  
# ## <strong>End(Not run)</strong>
```

## See also

[removeInputHandler](#)

### 3.101 Deregister an Input Handler

# Deregister an Input Handler

`removeInputHandler(type)`

## Arguments

`type` The type for which handlers should be removed.

## Value

The handler previously associated with this `type`, if one existed. Otherwise, `NULL`.

## Description

Removes an Input Handler. Rather than using the previously specified handler for data of this type, the default jsonlite serialization will be used.

## See also

[registerInputHandler](#)

### 3.102 Mark a function as a render function

# Mark a function as a render function

```
markRenderFunction(uiFunc, renderFunc)
```

## Arguments

`uiFunc` A function that renders Shiny UI. Must take a single argument: an output ID.

`renderFunc` A function that is suitable for assigning to a Shiny output slot.

## Value

The `renderFunc` function, with annotations.

## Description

Should be called by implementers of `renderXXX` functions in order to mark their return values as Shiny render functions, and to provide a hint to Shiny regarding what UI function is most commonly used with this type of render function. This can be used in R Markdown documents to create complete output widgets out of just the render function.



## 3.103 Validate input values and other

# Validate input values and other conditions

```
validate(..., errorClass = character(0))
```

```
need(expr, message = paste(label, "must be provided"), label)
```

## Arguments

...	A list of tests. Each test should equal <code>NULL</code> for success, <code>FALSE</code> for silent failure, or a string for failure with an error message.
<code>errorClass</code>	A CSS class to apply. The actual CSS string will have <code>shiny-output-error-</code> prepended to this value.
<code>expr</code>	An expression to test. The condition will pass if the expression meets the conditions spelled out in <a href="#">Details</a> .
<code>message</code>	A message to convey to the user if the validation condition is not met. If no message is provided, one will be created using <code>label</code> . To fail with no message, use <code>FALSE</code> for the message.
<code>label</code>	A human-readable name for the field that may be missing. This parameter is not needed if <code>message</code> is provided, but must be provided otherwise.

## Description

For an output rendering function (e.g. `renderPlot()`), you may need to check that certain input values are available and valid before you can render the output. `validate` gives you a convenient mechanism for doing so.

## Details

The `validate` function takes any number of (unnamed) arguments, each of which represents a condition to test. If any of the conditions represent failure, then a special type of error is signaled which stops execution. If this error is not handled by application-specific code, it is displayed to the user by Shiny.

An easy way to provide arguments to `validate` is to use the `need` function, which takes an expression and a string; if the expression is considered a failure, then the string will be used as the error message. The `need` function considers its expression to be a failure if it is any of the following:

- `FALSE`
- `NULL`
- `""`
- An empty atomic vector
- An atomic vector that contains only missing values
- A logical vector that contains all `FALSE` or missing values

- An object of class "try-error"
- A value that represents an unclicked `actionButton`

If any of these values happen to be valid, you can explicitly turn them to logical values. For example, if you allow `NA` but not `NULL`, you can use the condition `!is.null(input$foo)`, because `!is.null(NA) == TRUE`.

If you need validation logic that differs significantly from `need`, you can create other validation test functions. A passing test should return `NULL`. A failing test should return an error message as a single-element character vector, or if the failure should happen silently, `FALSE`.

Because validation failure is signaled as an error, you can use `validate` in reactive expressions, and validation failures will automatically propagate to outputs that use the reactive expression. In other words, if reactive expression `a` needs `input$x`, and two outputs use `a` (and thus depend indirectly on `input$x`), it's not necessary for the outputs to validate `input$x` explicitly, as long as `a` does validate it.

## Examples

```
# in ui.R
fluidPage(
  checkboxGroupInput('in1', 'Check some letters', choices = head(LETTERS)),
  selectizeInput('in2', 'Select a state', choices = state.name),
  plotOutput('plot')
)

<div class="container-fluid">
  <div id="in1" class="form-group shiny-input-checkboxgroup shiny-input-container">
    <label class="control-label" for="in1">Check some letters</label>
    <div class="shiny-options-group">
      <div class="checkbox">
        <label>
          <input type="checkbox" name="in1" value="A"/>
          <span>A</span>
        </label>
      </div>
      <div class="checkbox">
        <label>
          <input type="checkbox" name="in1" value="B"/>
          <span>B</span>
        </label>
      </div>
      <div class="checkbox">
        <label>
          <input type="checkbox" name="in1" value="C"/>
          <span>C</span>
        </label>
      </div>
      <div class="checkbox">
        <label>
          <input type="checkbox" name="in1" value="D"/>
          <span>D</span>
        </label>
      </div>
      <div class="checkbox">
        <label>
          <input type="checkbox" name="in1" value="E"/>
          <span>E</span>
        </label>
      </div>
    </div>
  </div>
</div>
```

```

</div>
  <div class="checkbox">
    <label>
      <input type="checkbox" name="in1" value="F"/>
      <span>F</span>
    </label>
  </div>
</div>
</div>
<div class="form-group shiny-input-container">
  <label class="control-label" for="in2">Select a state</label>
  <div>
    <select id="in2" class="form-control"><option value="Alabama" selected>Alabama</option>
<option value="Alaska">Alaska</option>
<option value="Arizona">Arizona</option>
<option value="Arkansas">Arkansas</option>
<option value="California">California</option>
<option value="Colorado">Colorado</option>
<option value="Connecticut">Connecticut</option>
<option value="Delaware">Delaware</option>
<option value="Florida">Florida</option>
<option value="Georgia">Georgia</option>
<option value="Hawaii">Hawaii</option>
<option value="Idaho">Idaho</option>
<option value="Illinois">Illinois</option>
<option value="Indiana">Indiana</option>
<option value="Iowa">Iowa</option>
<option value="Kansas">Kansas</option>
<option value="Kentucky">Kentucky</option>
<option value="Louisiana">Louisiana</option>
<option value="Maine">Maine</option>
<option value="Maryland">Maryland</option>
<option value="Massachusetts">Massachusetts</option>
<option value="Michigan">Michigan</option>
<option value="Minnesota">Minnesota</option>
<option value="Mississippi">Mississippi</option>
<option value="Missouri">Missouri</option>
<option value="Montana">Montana</option>
<option value="Nebraska">Nebraska</option>
<option value="Nevada">Nevada</option>
<option value="New Hampshire">New Hampshire</option>
<option value="New Jersey">New Jersey</option>
<option value="New Mexico">New Mexico</option>
<option value="New York">New York</option>
<option value="North Carolina">North Carolina</option>
<option value="North Dakota">North Dakota</option>
<option value="Ohio">Ohio</option>
<option value="Oklahoma">Oklahoma</option>
<option value="Oregon">Oregon</option>
<option value="Pennsylvania">Pennsylvania</option>
<option value="Rhode Island">Rhode Island</option>
<option value="South Carolina">South Carolina</option>
<option value="South Dakota">South Dakota</option>
<option value="Tennessee">Tennessee</option>
<option value="Texas">Texas</option>
<option value="Utah">Utah</option>
<option value="Vermont">Vermont</option>

```

*Validate input values and other*

```

<option value="Virginia">Virginia</option>
<option value="Washington">Washington</option>
<option value="West Virginia">West Virginia</option>
<option value="Wisconsin">Wisconsin</option>
<option value="Wyoming">Wyoming</option></select>
  <script type="application/json" data-for="in2">{}</script>
</div>
</div>
<div id="plot" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>
</div>

```

```
# in server.R
```

```

function(input, output) {
  output$plot <- renderPlot({
    validate(
      need(input$in1, 'Check at least one letter!'),
      need(input$in2 != '', 'Please choose a state.')
    )
    plot(1:10, main = paste(c(input$in1, input$in2), collapse = ', '))
  })
}

```

```

function(input, output) {
  output$plot <- renderPlot({
    validate(
      need(input$in1, 'Check at least one letter!'),
      need(input$in2 != '', 'Please choose a state.')
    )
    plot(1:10, main = paste(c(input$in1, input$in2), collapse = ', '))
  })
}
<environment: 0x48c5130>

```

## 3.104 Session object

# Session object

## Value

`clientData`

A `reactiveValues` object that contains information about the client.

- `allowDataUriScheme` is a logical value that indicates whether the browser is able to handle URIs that use the `data:` scheme.
- `pixelRatio` reports the "device pixel ratio" from the web browser, or 1 if none is reported. The value is 2 for Apple Retina displays.
- `singletons` - for internal use
- `url_protocol`, `url_hostname`, `url_port`, `url_pathname`, `url_search`, and `url_hash_initial` can be used to get the components of the URL that was requested by the browser to load the Shiny app page. These values are from the browser's perspective, so neither HTTP proxies nor Shiny Server will affect these values. The `url_search` value may be used with `parseQueryString` to access query string parameters.

`clientData` also contains information about each output. `output_outputId_width` and `output_outputId_height` give the dimensions (using `offsetWidth` and `offsetHeight`) of the DOM element that is bound to `outputId`, and `output_outputId_hidden` is a logical that indicates whether the element is hidden. These values may be `NULL` if the output is not bound.

`input`

The session's `input` object (the same as is passed into the Shiny server function as an argument).

`isClosed()`

A function that returns `TRUE` if the client has disconnected.

`onEnded(callback)`

Synonym for `onSessionEnded`.

`onFlush(func, once=TRUE)`

Registers a function to be called before the next time (if `once=TRUE`) or every time (if `once=FALSE`) Shiny flushes the reactive system. Returns a function that can be called with no arguments to cancel the registration.

`onFlushed(func, once=TRUE)`

Registers a function to be called after the next time (if `once=TRUE`) or every time (if `once=FALSE`) Shiny flushes the reactive system. Returns a function that can be called with no arguments to cancel the registration.

`onSessionEnded(callback)`

Registers a function to be called after the client has disconnected. Returns a function that can be called with no arguments to cancel the registration.

`output`

The session's `output` object (the same as is passed into the Shiny server function as an argument).

`reactlog`

For internal use.

`registerDataObj(name, data, filterFunc)`

Publishes any R object as a URL endpoint that is unique to this session. `name` must be a single element character vector; it will be used to form part of the URL. `filterFunc` must be a function that takes two arguments: `data` (the value that was passed into `registerDataObj`) and `req` (an environment that implements the Rook specification for HTTP requests). `filterFunc` will be called with these values whenever an HTTP request is made to the URL endpoint. The return value of `filterFunc` should be a Rook-style response.

`request`

An environment that implements the Rook specification for HTTP requests. This is the request that was used to initiate the websocket connection (as opposed to the request that downloaded the web page for the app).

`sendCustomMessage(type, message)`

Sends a custom message to the web page. `type` must be a single-element character vector giving the type of message, while `message` can be any jsonlite-encodable value. Custom messages have no meaning to Shiny itself; they are used solely to convey information to custom JavaScript logic in the browser. You can do this by adding JavaScript code to the browser that calls `Shiny.addCustomMessageHandler(type, function(message){...})` as the page loads; the function you provide to `addCustomMessageHandler` will be invoked each time `sendCustomMessage` is called on the server.

`sendInputMessage(inputId, message)`

Sends a message to an input on the session's client web page; if the input is present and bound on the page at the time the message is received, then the input binding object's `receiveMessage(el, message)` method will be called. `sendInputMessage` should generally not be called directly from Shiny apps, but through friendlier wrapper functions like [updateTextInput](#).

## Description

Shiny server functions can optionally include `session` as a parameter (e.g.

`function(input, output, session)`). The session object is an environment that can be used to access information and functionality relating to the session. The following list describes the items available in the environment; they can be accessed using the `$` operator (for example, `session$clientData$url_search`).

### 3.105 Convert an expression to a function

# Convert an expression to a function

```
exprToFunction(expr, env = parent.frame(2), quoted = FALSE, caller_offset = 1)
```

## Arguments

`expr` A quoted or unquoted expression, or a function.

`env` The desired environment for the function. Defaults to the calling environment two steps back.

`quoted` Is the expression quoted?

`caller_offset` If specified, the offset in the callstack of the function to be treated as the caller.

## Description

This is to be called from another function, because it will attempt to get an unquoted expression from two calls back.

## Details

If `expr` is a quoted expression, then this just converts it to a function. If `expr` is a function, then this simply returns `expr` (and prints a deprecation message). If `expr` was a non-quoted expression from two calls back, then this will quote the original expression and convert it to a function.

## Examples

```
# Example of a new renderer, similar to renderText
# This is something that toolkit authors will do
renderTriple <- function(expr, env=parent.frame(), quoted=FALSE) {
  # Convert expr to a function
  func <- shiny::exprToFunction(expr, env, quoted)

  function() {
    value <- func()
    paste(rep(value, 3), collapse=", ")
  }
}

# Example of using the renderer.
# This is something that app authors will do.
values <- reactiveValues(A="text")

## <strong>Not run</strong>:
# # Create an output object
# output$tripleA <- renderTriple({
```

```
# values$A
# })
# ## <strong>End(Not run)</strong>

# At the R console, you can experiment with the renderer using isolate()
tripleA <- renderTriple({
  values$A
})

isolate(tripleA())

[1] "text, text, text"

# "text, text, text"
```



### 3.106 Install an expression as a function

# Install an expression as a function

```
installExprFunction(expr, name, eval.env = parent.frame(2), quoted = FALSE,  
  assign.env = parent.frame(1), label = as.character(sys.call(-1)[[1]]))
```

## Arguments

<code>expr</code>	A quoted or unquoted expression
<code>name</code>	The name the function should be given
<code>eval.env</code>	The desired environment for the function. Defaults to the calling environment two steps back.
<code>quoted</code>	Is the expression quoted?
<code>assign.env</code>	The environment in which the function should be assigned.
<code>label</code>	A label for the object to be shown in the debugger. Defaults to the name of the calling function.

## Description

Installs an expression in the given environment as a function, and registers debug hooks so that breakpoints may be set in the function.

## Details

This function can replace `exprToFunction` as follows: we may use `func <- exprToFunction(expr)` if we do not want the debug hooks, or `installExprFunction(expr, "func")` if we do. Both approaches create a function named `func` in the current environment.

## See also

Wraps `exprToFunction`; see that method's documentation for more documentation and examples.

## 3.107 Parse a GET query string from a URL

# Parse a GET query string from a URL

```
parseQueryString(str, nested = FALSE)
```

## Arguments

**str** The query string. It can have a leading "?" or not.

**nested** Whether to parse the query string of as a nested list when it contains pairs of square brackets []. For example, the query `a[i1][j1]=x&b[i1][j1]=y&b[i2][j1]=z` will be parsed as `list(a = list(i1 = list(j1 = 'x')), b = list(i1 = list(j1 = 'y'), i2 = list(j1 = 'z')))` when `nested = TRUE`, and `list(`a[i1][j1]` = 'x', `b[i1][j1]` = 'y', `b[i2][j1]` = 'z')` when `nested = FALSE`.

## Description

Returns a named list of key-value pairs.

## Examples

```
parseQueryString("?foo=1&bar=b%20a%20r")
```

```
$foo
[1] "1"
```

```
$bar
[1] "b a r"
```

```
## <strong>Not run</strong>:
# # Example of usage within a Shiny app
# shinyServer(function(input, output, clientData) {
#
#   output$queryText <- renderText({
#     query <- parseQueryString(clientData$url_search)
#
#     # Ways of accessing the values
#     if (as.numeric(query$foo) == 1) {
#       # Do something
#     }
#     if (query[["bar"]] == "targetstring") {
#       # Do something else
#     }
#
#     # Return a string with key-value pairs
#   })
# }
```

*Parse a GET query string from a URL*

```
#   paste(names(query), query, sep = "=", collapse=", ")
#   })
#   })
# ## <strong>End(Not run)</strong>
```

Shiny is an [RStudio](#) project. © 2014 RStudio, Inc.

### 3.108 Run a plotting function and save the

# Run a plotting function and save the output as a PNG

```
plotPNG(func, filename = tempfile(fileext = ".png"), width = 400, height = 400, res = 72,
...)
```

## Arguments

<code>func</code>	A function that generates a plot.
<code>filename</code>	The name of the output file. Defaults to a temp file with extension <code>.png</code> .
<code>width</code>	Width in pixels.
<code>height</code>	Height in pixels.
<code>res</code>	Resolution in pixels per inch. This value is passed to <code>png</code> . Note that this affects the resolution of PNG rendering in R; it won't change the actual ppi of the browser.
<code>...</code>	Arguments to be passed through to <code>png</code> . These can be used to set the width, height, background color, etc.

## Description

This function returns the name of the PNG file that it generates. In essence, it calls `png()`, then `func()`, then `dev.off()`. So `func` must be a function that will generate a plot when used this way.

## Details

For output, it will try to use the following devices, in this order: `quartz` (via `png`), then `CairoPNG`, and finally `png`. This is in order of quality of output. Notably, plain `png` output on Linux and Windows may not antialias some point shapes, resulting in poor quality output.

In some cases, `Cairo()` provides output that looks worse than `png()`. To disable Cairo output for an app, use `options(shiny.usecairo=FALSE)`.

### 3.109 Make a random number generator

# Make a random number generator repeatable

```
repeatable(rngfunc, seed = runif(1, 0, .Machine$integer.max))
```

## Arguments

`rngfunc` The function that is affected by the R session's seed.

`seed` The seed to set every time the resulting function is called.

## Value

A repeatable version of the function that was passed in.

## Description

Given a function that generates random data, returns a wrapped version of that function that always uses the same seed when called. The seed to use can be passed in explicitly if desired; otherwise, a random number is used.

## Note

When called, the returned function attempts to preserve the R session's current seed by snapshotting and restoring `.Random.seed`.

## Examples

```
rnormA <- repeatable(rnorm)
rnormB <- repeatable(rnorm)
rnormA(3) # [1] 1.8285879 -0.7468041 -0.4639111

[1] 1.5308819 0.9697510 0.5213101

rnormA(3) # [1] 1.8285879 -0.7468041 -0.4639111

[1] 1.5308819 0.9697510 0.5213101

rnormA(5) # [1] 1.8285879 -0.7468041 -0.4639111 -1.6510126 -1.4686924

[1] 1.5308819 0.9697510 0.5213101 1.0563204 0.1629233

rnormB(5) # [1] -0.7946034 0.2568374 -0.6567597 1.2451387 -0.8375699
```

### 3.110 Print message for deprecated

# Print message for deprecated functions in Shiny

```
shinyDeprecated(new = NULL, msg = NULL, old = as.character(sys.call(sys.parent()))[1L],  
  version = NULL)
```

## Arguments

- `new` Name of replacement function.
- `msg` Message to print. If used, this will override the default message.
- `old` Name of deprecated function.
- `version` The last version of Shiny before the item was deprecated.

## Description

To disable these messages, use `options(shiny.deprecation.messages=FALSE)`.

### 3.111 Collect information about the Shiny

# Collect information about the Shiny Server environment

`serverInfo()`

## Value

A list of the Shiny Server information.

## Description

This function returns the information about the current Shiny Server, such as its version, and whether it is the open source edition or professional edition. If the app is not served through the Shiny Server, this function just returns `list(shinyServer = FALSE)`.

## Details

This function will only return meaningful data when using Shiny Server version 1.2.2 or later.

## 3.112 Global options for Shiny

# Global options for Shiny

## Description

There are a number of global options that affect Shiny's behavior. These can be set with (for example) `options(shiny.trace=TRUE)` .

## Details

`shiny.launch.browser`

A boolean which controls the default behavior when an app is run. See [runApp](#) for more information.

`shiny.port`

A port number that Shiny will listen on. See [runApp](#) for more information.

`shiny.trace`

If `TRUE` , all of the messages sent between the R server and the web browser client will be printed on the console. This is useful for debugging.

`shiny.reactlog`

If `TRUE` , enable logging of reactive events, which can be viewed later with the [showReactLog](#) function. This incurs a substantial performance penalty and should not be used in production.

`shiny.usecairo`

This is used to disable graphical rendering by the Cairo package, if it is installed. See [plotPNG](#) for more information.

`shiny.maxRequestSize`

This is a number which specifies the maximum web request size, which serves as a size limit for file uploads. If unset, the maximum request size defaults to 5MB.

`shiny.suppressMissingContextError`

Normally, invoking a reactive outside of a reactive context (or `isolate()` ) results in an error. If this is `TRUE` , don't error in these cases. This should only be used for debugging or demonstrations of reactivity at the console.

`shiny.host`

The IP address that Shiny should listen on. See [runApp](#) for more information.

`shiny.json.digits`

The number of digits to use when converting numbers to JSON format to send to the client web browser.

`shiny.error`

This can be a function which is called when an error occurs. For example, `options(shiny.error=recover)` will result a the debugger prompt when an error occurs.

`shiny.observer.error`

This can be a function that is called by an observer when an unhandled error occurs in it or an upstream reactive. By default, these errors will result in a warning at the console, and the websocket connection will close.

`shiny.table.class`



CSS class names to use for tables.

`shiny.deprecation.messages`

This controls whether messages for deprecated functions in Shiny will be printed. See [shinyDeprecated](#) for more information.

### 3.113 Find rows of data that are selected by

# Find rows of data that are selected by a brush

```
brushedPoints(df, brush, xvar = NULL, yvar = NULL, panel var1 = NULL, panel var2 = NULL,
  allRows = FALSE)
```

## Arguments

<code>df</code>	A data frame from which to select rows.
<code>brush</code>	The data from a brush, such as <code>input\$plot_brush</code> .
<code>xvar,yvar</code>	A string with the name of the variable on the x or y axis. This must also be the name of a column in <code>df</code> . If absent, then this function will try to infer the variable from the brush (only works for ggplot2).
<code>panelvar1,panelvar2</code>	Each of these is a string with the name of a panel variable. For example, if with ggplot2, you facet on a variable called <code>cyl</code> , then you can use <code>"cyl"</code> here. However, specifying the panel variable should not be necessary with ggplot2; Shiny should be able to auto-detect the panel variable.
<code>allRows</code>	If <code>FALSE</code> (the default) return a data frame containing the selected rows. If <code>TRUE</code> , the input data frame will have a new column, <code>selected_</code> , which indicates whether the row was inside the brush ( <code>TRUE</code> ) or outside the brush ( <code>FALSE</code> ).

## Description

This function returns rows from a data frame which are under a brush used with `plotOutput`.

## Details

It is also possible for this function to return all rows from the input data frame, but with an additional column `selected_`, which indicates which rows of the input data frame are selected by the brush (`TRUE` for selected, `FALSE` for not-selected). This is enabled by setting `allRows=TRUE` option.

The `xvar`, `yvar`, `panel var1`, and `panel var2` arguments specify which columns in the data correspond to the x variable, y variable, and panel variables of the plot. For example, if your plot is `plot(x=cars$speed, y=cars$dist)`, and your brush is named `"cars_brush"`, then you would use `brushedPoints(cars, input$cars_brush, "speed", "dist")`.

For plots created with ggplot2, it should not be necessary to specify the column names; that information will already be contained in the brush, provided that variables are in the original data, and not computed. For example, with `ggplot(cars, aes(x=speed, y=dist)) + geom_point()`, you could use `brushedPoints(cars, input$cars_brush)`. If, however, you use a computed column, like `ggplot(cars, aes(x=speed/2, y=dist)) + geom_point()`, then it will not be able to automatically extract

column names and filter on them. If you want to use this function to filter data, it is recommended that you not use computed columns; instead, modify the data first, and then make the plot with "raw" columns in the modified data.

If a specified x or y column is a factor, then it will be coerced to an integer vector. If it is a character vector, then it will be coerced to a factor and then integer vector. This means that the brush will be considered to cover a given character/factor value when it covers the center value.

If the brush is operating in just the x or y directions (e.g., with `brushOpts(direction = "x")`), then this function will filter out points using just the x or y variable, whichever is appropriate.

## See also

[plotOutput](#) for example usage.

### 3.114 Create an object representing

# Create an object representing brushing options

```
brushOpts(id = NULL, fill = "#9cf", stroke = "#036", opacity = 0.25, delay = 300,  
  delayType = c("debounce", "throttle"), clip = TRUE, direction = c("xy", "x", "y"),  
  resetOnNew = FALSE)
```

## Arguments

id	Input value name. For example, if the value is "plot_brush", then the coordinates will be available as <code>input\$plot_brush</code> .
fill	Fill color of the brush.
stroke	Outline color of the brush.
opacity	Opacity of the brush
delay	How long to delay (in milliseconds) when debouncing or throttling, before sending the brush data to the server.
delayType	The type of algorithm for limiting the number of brush events. Use "throttle" to limit the number of brush events to one every <code>delay</code> milliseconds. Use "debounce" to suspend events while the cursor is moving, and wait until the cursor has been at rest for <code>delay</code> milliseconds before sending an event.
clip	Should the brush area be clipped to the plotting area? If FALSE, then the user will be able to brush outside the plotting area, as long as it is still inside the image.
direction	The direction for brushing. If "xy", the brush can be drawn and moved in both x and y directions. If "x", or "y", the brush will work horizontally or vertically.
resetOnNew	When a new image is sent to the browser (via <code>renderImage</code> ), should the brush be reset? The default, FALSE, is useful if you want to update the plot while keeping the brush. Using TRUE is useful if you want to clear the brush whenever the plot is updated.

## Description

This generates an object representing brushing options, to be passed as the `brush` argument of `imageOutput` or `plotOutput`.

### 3.115 Create an object representing click

# Create an object representing click options

```
clickOpts(id = NULL, clip = TRUE)
```

## Arguments

- `id` Input value name. For example, if the value is `"plot_click"`, then the click coordinates will be available as `input$plot_click`.
- `clip` Should the click area be clipped to the plotting area? If `FALSE`, then the server will receive click events even when the mouse is outside the plotting area, as long as it is still inside the image.

## Description

This generates an object representing click options, to be passed as the `click` argument of `imageOutput` or `plotOutput`.

### 3.116 Create an object representing

# Create an object representing double-click options

```
dblclickOpts(id = NULL, clip = TRUE, delay = 400)
```

## Arguments

- id** Input value name. For example, if the value is "plot\_dblick", then the click coordinates will be available as `input$plot_dblick`.
- clip** Should the click area be clipped to the plotting area? If `FALSE`, then the server will receive double-click events even when the mouse is outside the plotting area, as long as it is still inside the image.
- delay** Maximum delay (in ms) between a pair clicks for them to be counted as a double-click.

## Description

This generates an object representing double-click options, to be passed as the `dblclick` argument of `imageOutput` or `plotOutput`.

### 3.117 Create an object representing hover

# Create an object representing hover options

```
hoverOpts(id = NULL, delay = 300, delayType = c("debounce", "throttle"), clip = TRUE,
  nullOutside = TRUE)
```

## Arguments

<code>id</code>	Input value name. For example, if the value is <code>"plot_hover"</code> , then the hover coordinates will be available as <code>input\$plot_hover</code> .
<code>delay</code>	How long to delay (in milliseconds) when debouncing or throttling, before sending the mouse location to the server.
<code>delayType</code>	The type of algorithm for limiting the number of hover events. Use <code>"throttle"</code> to limit the number of hover events to one every <code>delay</code> milliseconds. Use <code>"debounce"</code> to suspend events while the cursor is moving, and wait until the cursor has been at rest for <code>delay</code> milliseconds before sending an event.
<code>clip</code>	Should the hover area be clipped to the plotting area? If <code>FALSE</code> , then the server will receive hover events even when the mouse is outside the plotting area, as long as it is still inside the image.
<code>nullOutside</code>	If <code>TRUE</code> (the default), the value will be set to <code>NULL</code> when the mouse exits the plotting area. If <code>FALSE</code> , the value will stop changing when the cursor exits the plotting area.

## Description

This generates an object representing hovering options, to be passed as the `hover` argument of `imageOutput` or `plotOutput`.

### 3.118 Find rows of data that are near a

# Find rows of data that are near a click/hover/double-click

```
nearPoints(df, coordinfo, xvar = NULL, yvar = NULL, panelvar1 = NULL, panelvar2 = NULL,
  threshold = 5, maxpoints = NULL, addDist = FALSE, allRows = FALSE)
```

## Arguments

<code>df</code>	A data frame from which to select rows.
<code>coordinfo</code>	The data from a mouse event, such as <code>input\$plot_click</code> .
<code>xvar</code>	A string with the name of the variable on the x or y axis. This must also be the name of a column in <code>df</code> . If absent, then this function will try to infer the variable from the brush (only works for <code>ggplot2</code> ).
<code>yvar</code>	A string with the name of the variable on the x or y axis. This must also be the name of a column in <code>df</code> . If absent, then this function will try to infer the variable from the brush (only works for <code>ggplot2</code> ).
<code>panelvar1</code>	Each of these is a string with the name of a panel variable. For example, if with <code>ggplot2</code> , you facet on a variable called <code>cy1</code> , then you can use <code>"cy1"</code> here. However, specifying the panel variable should not be necessary with <code>ggplot2</code> ; Shiny should be able to auto-detect the panel variable.
<code>panelvar2</code>	Each of these is a string with the name of a panel variable. For example, if with <code>ggplot2</code> , you facet on a variable called <code>cy1</code> , then you can use <code>"cy1"</code> here. However, specifying the panel variable should not be necessary with <code>ggplot2</code> ; Shiny should be able to auto-detect the panel variable.
<code>threshold</code>	A maximum distance to the click point; rows in the data frame where the distance to the click is less than <code>threshold</code> will be returned.
<code>maxpoints</code>	Maximum number of rows to return. If <code>NULL</code> (the default), return all rows that are within the threshold distance.
<code>addDist</code>	If <code>TRUE</code> , add a column named <code>dist_</code> that contains the distance from the coordinate to the point, in pixels. When no mouse event has yet occurred, the value of <code>dist_</code> will be <code>NA</code> .
<code>allRows</code>	If <code>FALSE</code> (the default) return a data frame containing the selected rows. If <code>TRUE</code> , the input data frame will have a new column, <code>selected_</code> , which indicates whether the row was inside the selected by the mouse event ( <code>TRUE</code> ) or not ( <code>FALSE</code> ).

## Description

This function returns rows from a data frame which are near a click, hover, or double-click, when used with `plotOutput`. The rows will be sorted by their distance to the mouse event.

## Details

It is also possible for this function to return all rows from the input data frame, but with an additional column



`selected_`, which indicates which rows of the input data frame are selected by the brush (`TRUE` for selected, `FALSE` for not-selected). This is enabled by setting `allRows=TRUE` option. If this is used, the resulting data frame will not be sorted by distance to the mouse event.

The `xvar`, `yvar`, `panel var1`, and `panel var2` arguments specify which columns in the data correspond to the x variable, y variable, and panel variables of the plot. For example, if your plot is `plot(x=cars$speed, y=cars$dist)`, and your click variable is named `"cars_click"`, then you would use `nearPoints(cars, input$cars_brush, "speed", "dist")`.

## Examples

```
## Not run:
# # Note that in practice, these examples would need to go in reactives
# # or observers.
#
# # This would select all points within 5 pixels of the click
# nearPoints(mtcars, input$plot_click)
#
# # Select just the nearest point within 10 pixels of the click
# nearPoints(mtcars, input$plot_click, threshold = 10, maxpoints = 1)
#
# ## End(Not run)
```

## See also

[plotOutput](#) for more examples.

## 3.119 Create a Shiny app object

# Create a Shiny app object

```
shinyApp(ui = NULL, server = NULL, onStart = NULL, options = list(), uiPattern = "/")
```

```
shinyAppDir(appDir, options = list())
```

```
as.shiny.appobj(x)
```

```
"as.shiny.appobj" (x)
```

```
"as.shiny.appobj" (x)
```

```
"as.shiny.appobj" (x)
```

```
is.shiny.appobj(x)
```

```
"print" (x, ...)
```

```
"as.tags" (x, ...)
```

## Arguments

ui	The UI definition of the app (for example, a call to <code>fluidPage()</code> with nested controls)
server	A server function
onStart	A function that will be called before the app is actually run. This is only needed for <code>shinyAppObj</code> , since in the <code>shinyAppDir</code> case, a <code>global.R</code> file can be used for this purpose.
options	Named options that should be passed to the <code>runApp</code> call. You can also specify <code>width</code> and <code>height</code> parameters which provide a hint to the embedding environment about the ideal height/width for the app.
uiPattern	A regular expression that will be applied to each <code>GET</code> request to determine whether the <code>ui</code> should be used to handle the request. Note that the entire request path must match the regular expression in order for the match to be considered successful.
appDir	Path to directory that contains a Shiny app (i.e. a <code>server.R</code> file and either <code>ui.R</code> or <code>www/index.html</code> )
x	Object to convert to a Shiny app.
...	Additional parameters to be passed to <code>print</code> .

## Value

An object that represents the app. Printing the object or passing it to `runApp` will run the app.

## Description

These functions create Shiny app objects from either an explicit UI/server pair ( `shinyApp` ), or by passing the path of a directory that contains a Shiny app ( `shinyAppDir` ). You generally shouldn't need to use these functions to create/run applications; they are intended for interoperability purposes, such as embedding Shiny apps inside a knitr document.

## Details

Normally when this function is used at the R console, the Shiny app object is automatically passed to the `print()` function, which runs the app. If this is called in the middle of a function, the value will not be passed to `print()` and the app will not be run. To make the app run, pass the app object to `print()` or `runApp()` .

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  shinyApp(
    ui = fluidPage(
      numericInput("n", "n", 1),
      plotOutput("plot")
    ),
    server = function(input, output) {
      output$plot <- renderPlot( plot(head(cars, input$n)) )
    }
  )

  shinyAppDir(system.file("examples/01_hello", package="shiny"))

# The object can be passed to runApp()
app <- shinyApp(
  ui = fluidPage(
    numericInput("n", "n", 1),
    plotOutput("plot")
  ),
  server = function(input, output) {
    output$plot <- renderPlot( plot(head(cars, input$n)) )
  }
)

runApp(app)
}
```

### 3.120 Evaluate an expression without a

# Evaluate an expression without a reactive context

`maskReactiveContext (expr)`

## Arguments

`expr` An expression to evaluate.

## Value

The value of `expr`.

## Description

Temporarily blocks the current reactive context and evaluates the given expression. Any attempt to directly access reactive values or expressions in `expr` will give the same results as doing it at the top-level (by default, an error).

## See also

`isolate`

## 3.121 OVERVIEW

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Stop the currently running Shiny app

```
stopApp(returnValue = NULL)
```

## Arguments

`returnValue` The value that should be returned from `runApp`.

## Description

Stops the currently running Shiny app, returning control to the caller of `runApp`.

## 3.122 Create a web dependency

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Create a web dependency

```
createWebDependency(dependency)
```

## Arguments

`dependency` A single HTML dependency object, created using [htmlDependency](#) . If the `src` value is named, then `href` and/or `file` names must be present.

## Value

A single HTML dependency object that has an `href` -named element in its `src` .

## Description

Ensure that a file-based HTML dependency (from the `htmltools` package) can be served over Shiny's HTTP server. This function works by using [addResourcePath](#) to map the HTML dependency's directory to a URL.

## 3.123 Resource Publishing

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Resource Publishing

`addResourcePath(prefix, directoryPath)`

## Arguments

- `prefix` The URL prefix (without slashes). Valid characters are a-z, A-Z, 0-9, hyphen, period, and underscore; and must begin with a-z or A-Z. For example, a value of 'foo' means that any request paths that begin with '/foo' will be mapped to the given directory.
- `directoryPath` The directory that contains the static resources to be served.

## Description

Adds a directory of static resources to Shiny's web server, with the given path prefix. Primarily intended for package authors to make supporting JavaScript/CSS files available to their components.

## Details

You can call `addResourcePath` multiple times for a given `prefix`; only the most recent value will be retained. If the normalized `directoryPath` is different than the directory that's currently mapped to the `prefix`, a warning will be issued.

## Examples

```
addResourcePath('datasets', system.file('data', package='datasets'))
```

## See also

[singleton](#)

Shiny is an [RStudio](#) project. © 2014 RStudio, Inc.



## 3.124 Register an Input Handler

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Register an Input Handler

```
registerInputHandler(type, fun, force = FALSE)
```

## Arguments

- type** The type for which the handler should be added -- should be a single-element character vector.
- fun** The handler function. This is the function that will be used to parse the data delivered from the client before it is available in the `input` variable. The function will be called with the following three parameters:
1. The value of this input as provided by the client, deserialized using RJSONIO.
  2. The `shinySession` in which the input exists.
  3. The name of the input.
- force** If `TRUE`, will overwrite any existing handler without warning. If `FALSE`, will throw an error if this class already has a handler defined.

## Description

Adds an input handler for data of this type. When called, Shiny will use the function provided to refine the data passed back from the client (after being deserialized by RJSONIO) before making it available in the `input` variable of the `server.R` file.

## Details

This function will register the handler for the duration of the R process (unless Shiny is explicitly reloaded). For that reason, the `type` used should be very specific to this package to minimize the risk of colliding with another Shiny package which might use this data type name. We recommend the format of "packageName.widgetName".

Currently Shiny registers the following handlers: `shiny.matrix`, `shiny.number`, and `shiny.date`.

The `type` of a custom Shiny Input widget will be deduced using the `getType()` JavaScript function on the registered Shiny inputBinding.

## Examples

```
## <strong>Not run</strong>:
# # Register an input handler which rounds a input number to the nearest integer
# registerInputHandler("mypackage.validint", function(x, shinySession, name) {
#   if (is.null(x)) return(NA)
#   round(x)
# })
#
# ## On the Javascript side, the associated input binding must have a corresponding getType method:
# getType: function(e1) {
#   return "mypackage.validint";
# }
#
# ## <strong>End(Not run)</strong>
```

## See also

[removeInputHandler](#)

## 3.125 Deregister an Input Handler

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Deregister an Input Handler

`removeInputHandler(type)`

## Arguments

`type` The type for which handlers should be removed.

## Value

The handler previously associated with this `type`, if one existed. Otherwise, `NULL`.

## Description

Removes an Input Handler. Rather than using the previously specified handler for data of this type, the default RJSONIO serialization will be used.

## See also

[registerInputHandler](#)

## 3.126 Mark a function as a render function

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Mark a function as a render function

```
markRenderFunction(uiFunc, renderFunc)
```

## Arguments

`uiFunc` A function that renders Shiny UI. Must take a single argument: an output ID.

`renderFunc` A function that is suitable for assigning to a Shiny output slot.

## Value

The `renderFunc` function, with annotations.

## Description

Should be called by implementers of `renderXXX` functions in order to mark their return values as Shiny render functions, and to provide a hint to Shiny regarding what UI function is most commonly used with this type of render function. This can be used in R Markdown documents to create complete output widgets out of just the render function.

## 3.127 Validate input values and other

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Validate input values and other conditions

```
validate(..., errorClass = character(0))
```

```
need(expr, message = paste(label, "must be provided"), label)
```

## Arguments

...	A list of tests. Each test should equal <code>NULL</code> for success, <code>FALSE</code> for silent failure, or a string for failure with an error message.
<code>errorClass</code>	A CSS class to apply. The actual CSS string will have <code>shiny-output-error-</code> prepended to this value.
<code>expr</code>	An expression to test. The condition will pass if the expression meets the conditions spelled out in Details.
<code>message</code>	A message to convey to the user if the validation condition is not met. If no message is provided, one will be created using <code>label</code> . To fail with no message, use <code>FALSE</code> for the message.
<code>label</code>	A human-readable name for the field that may be missing. This parameter is not needed if <code>message</code> is provided, but must be provided otherwise.

## Description

For an output rendering function (e.g. `renderPlot()`), you may need to check that certain input values are available and valid before you can render the output. `validate` gives you a convenient mechanism for doing so.

## Details

The `validate` function takes any number of (unnamed) arguments, each of which represents a condition to test. If any of the conditions represent failure, then a special type of error is signaled which stops execution. If this error is not handled by application-specific code, it is displayed to the user by Shiny.

An easy way to provide arguments to `validate` is to use the `need` function, which takes an expression and a string; if the expression is considered a failure, then the string will be used as the error message. The `need` function considers its expression to be a failure if it is any of the following:

- `FALSE`
- `NULL`
- `""`
- An empty atomic vector
- An atomic vector that contains only missing values
- A logical vector that contains all `FALSE` or missing values
- An object of class `"try-error"`
- A value that represents an unclicked `actionButton`

If any of these values happen to be valid, you can explicitly turn them to logical values. For example, if you allow `NA` but not `NULL`, you can use the condition `!is.null(input$foo)`, because `!is.null(NA) == TRUE`.

If you need validation logic that differs significantly from `need`, you can create other validation test functions. A passing test should return `NULL`. A failing test should return an error message as a single-element character vector, or if the failure should happen silently, `FALSE`.

Because validation failure is signaled as an error, you can use `validate` in reactive expressions, and validation failures will automatically propagate to outputs that use the reactive expression. In other words, if reactive expression `a` needs `input$x`, and two outputs use `a` (and thus depend indirectly on `input$x`), it's not necessary for the outputs to validate `input$x` explicitly, as long as `a` does validate it.

## Examples

```
# in ui.R
fluidPage(
  checkboxGroupInput('in1', 'Check some letters', choices = head(LETTERS)),
  selectizeInput('in2', 'Select a state', choices = state.name),
  plotOutput('plot')
)

<div class="container-fluid">
  <div id="in1" class="form-group shiny-input-checkboxgroup shiny-input-container">
    <label class="control-label" for="in1">Check some letters</label>
    <div class="shiny-options-group">
      <div class="checkbox">
        <label>
          <input type="checkbox" name="in1" id="in11" value="A"/>
          <span>A</span>
        </label>
      </div>
    </div>
  </div>
</div>
```

```

    </label>
  </div>
  <div class="checkbox">
    <label>
      <input type="checkbox" name="in1" id="in12" value="B"/>
      <span>B</span>
    </label>
  </div>
  <div class="checkbox">
    <label>
      <input type="checkbox" name="in1" id="in13" value="C"/>
      <span>C</span>
    </label>
  </div>
  <div class="checkbox">
    <label>
      <input type="checkbox" name="in1" id="in14" value="D"/>
      <span>D</span>
    </label>
  </div>
  <div class="checkbox">
    <label>
      <input type="checkbox" name="in1" id="in15" value="E"/>
      <span>E</span>
    </label>
  </div>
  <div class="checkbox">
    <label>
      <input type="checkbox" name="in1" id="in16" value="F"/>
      <span>F</span>
    </label>
  </div>
</div>
<div class="form-group shiny-input-container">
  <label class="control-label" for="in2">Select a state</label>
  <div>
    <select id="in2"><option value="Alabama" selected>Alabama</option>
<option value="Alaska">Alaska</option>
<option value="Arizona">Arizona</option>
<option value="Arkansas">Arkansas</option>
<option value="California">California</option>
<option value="Colorado">Colorado</option>
<option value="Connecticut">Connecticut</option>
<option value="Delaware">Delaware</option>
<option value="Florida">Florida</option>
<option value="Georgia">Georgia</option>
<option value="Hawaii">Hawaii</option>
<option value="Idaho">Idaho</option>
<option value="Illinois">Illinois</option>
<option value="Indiana">Indiana</option>
<option value="Iowa">Iowa</option>

```

*Validate input values and other*

```

<option value="Kansas">Kansas</option>
<option value="Kentucky">Kentucky</option>
<option value="Louisiana">Louisiana</option>
<option value="Maine">Maine</option>
<option value="Maryland">Maryland</option>
<option value="Massachusetts">Massachusetts</option>
<option value="Michigan">Michigan</option>
<option value="Minnesota">Minnesota</option>
<option value="Mississippi">Mississippi</option>
<option value="Missouri">Missouri</option>
<option value="Montana">Montana</option>
<option value="Nebraska">Nebraska</option>
<option value="Nevada">Nevada</option>
<option value="New Hampshire">New Hampshire</option>
<option value="New Jersey">New Jersey</option>
<option value="New Mexico">New Mexico</option>
<option value="New York">New York</option>
<option value="North Carolina">North Carolina</option>
<option value="North Dakota">North Dakota</option>
<option value="Ohio">Ohio</option>
<option value="Oklahoma">Oklahoma</option>
<option value="Oregon">Oregon</option>
<option value="Pennsylvania">Pennsylvania</option>
<option value="Rhode Island">Rhode Island</option>
<option value="South Carolina">South Carolina</option>
<option value="South Dakota">South Dakota</option>
<option value="Tennessee">Tennessee</option>
<option value="Texas">Texas</option>
<option value="Utah">Utah</option>
<option value="Vermont">Vermont</option>
<option value="Virginia">Virginia</option>
<option value="Washington">Washington</option>
<option value="West Virginia">West Virginia</option>
<option value="Wisconsin">Wisconsin</option>
<option value="Wyoming">Wyoming</option></select>
  <script type="application/json" data-for="in2">{}</script>
</div>
</div>
<div id="plot" class="shiny-plot-output" style="width: 100% ; height: 400px"></div>
</div>

```

```
# in server.R
```

```

function(input, output) {
  output$plot <- renderPlot({
    validate(
      need(input$in1, 'Check at least one letter!'),
      need(input$in2 != '', 'Please choose a state.')
    )
    plot(1:10, main = paste(c(input$in1, input$in2), collapse = ', '))
  })
}

```



*Validate input values and other*

```
function(input, output) {  
  output$plot <- renderPlot({  
    validate(  
      need(input$in1, 'Check at least one letter!'),  
      need(input$in2 != '', 'Please choose a state.')  
    )  
    plot(1:10, main = paste(c(input$in1, input$in2), collapse = ', '))  
  })  
}  
<environment: 0x5174bb8>
```

Shiny is an RStudio project. © 2014 RStudio, Inc.

## 3.128 Session object

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Session object

## Value

`clientData`

A [reactiveValues](#) object that contains information about the client.

- `allowDataUriScheme` is a logical value that indicates whether the browser is able to handle URIs that use the `data:` scheme.
- `pixelRatio` reports the "device pixel ratio" from the web browser, or 1 if none is reported. The value is 2 for Apple Retina displays.
- `singletons` - for internal use
- `url_protocol`, `url_hostname`, `url_port`, `url_pathname`, `url_search`, and `url_hash_initial` can be used to get the components of the URL that was requested by the browser to load the Shiny app page. These values are from the browser's perspective, so neither HTTP proxies nor Shiny Server will affect these values. The `url_search` value may be used with [parseQueryString](#) to access query string parameters.

`clientData` also contains information about each output. `output_outputId_width` and `output_outputId_height` give the dimensions (using `offsetWidth` and `offsetHeight`) of the DOM element that is bound to `outputId`, and `output_outputId_hidden` is a logical that indicates whether the element is hidden. These values may be `NULL` if the output is not bound.

`input`

The session's `input` object (the same as is passed into the Shiny server function as an argument).

`isClosed()`

A function that returns `TRUE` if the client has disconnected.

`onEnded(callback)`

Synonym for `onSessionEnded`.

`onFlush(func, once=TRUE)`

Registers a function to be called before the next time (if `once=TRUE`) or every time (if `once=FALSE`) Shiny flushes the reactive system. Returns a function that can be called with no arguments to cancel the registration.

`onFlushed(func, once=TRUE)`

Registers a function to be called after the next time (if `once=TRUE`) or every time (if `once=FALSE`) Shiny flushes the reactive system. Returns a function that can be called with no arguments to cancel the registration.

`onSessionEnded(callback)`

Registers a function to be called after the client has disconnected. Returns a function that can be called with no arguments to cancel the registration.

`output`

The session's `output` object (the same as is passed into the Shiny server function as an argument).

`reactlog`

For internal use.

`registerDataObj(name, data, filterFunc)`

Publishes any R object as a URL endpoint that is unique to this session. `name` must be a single element character vector; it will be used to form part of the URL. `filterFunc` must be a function that takes two arguments: `data` (the value that was passed into `registerDataObj`) and `req` (an environment that implements the Rook specification for HTTP requests). `filterFunc` will be called with these values whenever an HTTP request is made to the URL endpoint. The return value of `filterFunc` should be a Rook-style response.

`request`

An environment that implements the Rook specification for HTTP requests. This is the request that was used to initiate the websocket connection (as opposed to the request that downloaded the web page for the app).

`sendCustomMessage(type, message)`

Sends a custom message to the web page. `type` must be a single-element character vector giving the type of message, while `message` can be any RJSONIO-encodable value. Custom messages have no meaning to Shiny itself; they are used solely to convey information to custom JavaScript logic in the browser. You can do this by adding JavaScript code to the browser that calls `Shiny.addCustomMessageHandler(type, function(message){...})` as the page loads; the function you provide to `addCustomMessageHandler` will be invoked each time `sendCustomMessage` is called on the server.

`sendInputMessage(inputId, message)`

Sends a message to an input on the session's client web page; if the input is present and bound on the page at the time the message is received, then the input binding object's `receiveMessage(e1, message)` method will be called. `sendInputMessage` should generally not be called directly from Shiny apps, but through friendlier wrapper functions like `updateTextInput`.

## Description

Shiny server functions can optionally include `session` as a parameter (e.g.

`function(input, output, session)`). The session object is an environment that can be used to access information and functionality relating to the session. The following list describes the items available in the environment; they can be accessed using the `$` operator (for example, `session$clientData$url_search`).

## 3.129 Convert an expression to a function

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Convert an expression to a function

```
exprToFunction(expr, env = parent.frame(2), quoted = FALSE, caller_offset = 1)
```

## Arguments

- `expr` A quoted or unquoted expression, or a function.
- `env` The desired environment for the function. Defaults to the calling environment two steps back.
- `quoted` Is the expression quoted?
- `caller_offset` If specified, the offset in the callstack of the function to be treated as the caller.

## Description

This is to be called from another function, because it will attempt to get an unquoted expression from two calls back.

## Details

If `expr` is a quoted expression, then this just converts it to a function. If `expr` is a function, then this simply returns `expr` (and prints a deprecation message). If `expr` was a non-quoted expression from two calls back, then this will quote the original expression and convert it to a function.

## Examples

```
# Example of a new renderer, similar to renderText
# This is something that toolkit authors will do
renderTriple <- function(expr, env=parent.frame(), quoted=FALSE) {
  # Convert expr to a function
  func <- shiny::exprToFunction(expr, env, quoted)

  function() {
    value <- func()
    paste(rep(value, 3), collapse=", ")
  }
}

# Example of using the renderer.
# This is something that app authors will do.
values <- reactiveValues(A="text")

## <strong>Not run</strong>:
# # Create an output object
# output$tripleA <- renderTriple({
#   values$A
# })
# ## <strong>End(Not run)</strong>

# At the R console, you can experiment with the renderer using isolate()
tripleA <- renderTriple({
  values$A
})

isolate(tripleA())

[1] "text, text, text"

# "text, text, text"
```

## 3.130 Install an expression as a function

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Install an expression as a function

```
installExprFunction(expr, name, eval.env = parent.frame(2), quoted = FALSE,  
  assign.env = parent.frame(1), label = as.character(sys.call(-1)[[1]]))
```

## Arguments

<code>expr</code>	A quoted or unquoted expression
<code>name</code>	The name the function should be given
<code>eval.env</code>	The desired environment for the function. Defaults to the calling environment two steps back.
<code>quoted</code>	Is the expression quoted?
<code>assign.env</code>	The environment in which the function should be assigned.
<code>label</code>	A label for the object to be shown in the debugger. Defaults to the name of the calling function.

## Description

Installs an expression in the given environment as a function, and registers debug hooks so that breakpoints may be set in the function.

## Details

This function can replace `exprToFunction` as follows: we may use `func <- exprToFunction(expr)` if we do

not want the debug hooks, or `installExprFunction(expr, "func")` if we do. Both approaches create a function named `func` in the current environment.

## See also

Wraps [exprToFunction](#) ; see that method's documentation for more documentation and examples.

## 3.131 Parse a GET query string from a URL

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Parse a GET query string from a URL

```
parseQueryString(str, nested = FALSE)
```

## Arguments

**str** The query string. It can have a leading "?" or not.

**nested** Whether to parse the query string of as a nested list when it contains pairs of square brackets []. For example, the query `a[i1][j1]=x&b[i1][j1]=y&b[i2][j1]=z` will be parsed as `list(a = list(i1 = list(j1 = 'x')), b = list(i1 = list(j1 = 'y'), i2 = list(j1 = 'z')))` when `nested = TRUE`, and `list(`a[i1][j1]` = 'x', `b[i1][j1]` = 'y', `b[i2][j1]` = 'z')` when `nested = FALSE`.

## Description

Returns a named character vector of key-value pairs.

## Examples

```
parseQueryString("?foo=1&bar=b%20a%20r")
```

```
$foo  
[1] "1"
```

```
$bar
```



```
[1] "b a r"
```

```
## <strong>Not run</strong>:  
# # Example of usage within a Shiny app  
# shinyServer(function(input, output, clientData) {  
#  
#   output$queryText <- renderText({  
#     query <- parseQueryString(clientData$url_search)  
#  
#     # Ways of accessing the values  
#     if (as.numeric(query$foo) == 1) {  
#       # Do something  
#     }  
#     if (query[["bar"]] == "targetstring") {  
#       # Do something else  
#     }  
#  
#     # Return a string with key-value pairs  
#     paste(names(query), query, sep = "=", collapse=", ")  
#   })  
# })  
# ## <strong>End(Not run)</strong>
```

## 3.132 Run a plotting function and save the

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Run a plotting function and save the output as a PNG

```
plotPNG(func, filename = tempfile(fileext = ".png"), width = 400, height = 400, res = 72,
...)
```

## Arguments

<code>func</code>	A function that generates a plot.
<code>filename</code>	The name of the output file. Defaults to a temp file with extension <code>.png</code> .
<code>width</code>	Width in pixels.
<code>height</code>	Height in pixels.
<code>res</code>	Resolution in pixels per inch. This value is passed to <code>png</code> . Note that this affects the resolution of PNG rendering in R; it won't change the actual ppi of the browser.
<code>...</code>	Arguments to be passed through to <code>png</code> . These can be used to set the width, height, background color, etc.

## Description

This function returns the name of the PNG file that it generates. In essence, it calls `png()`, then `func()`, then `dev.off()`. So `func` must be a function that will generate a plot when used this way.

## Details

For output, it will try to use the following devices, in this order: quartz (via `png`), then `CairoPNG`, and finally `png`. This is in order of quality of output. Notably, plain `png` output on Linux and Windows may not antialias some point shapes, resulting in poor quality output.

In some cases, `Cairo()` provides output that looks worse than `png()`. To disable Cairo output for an app, use `options(shiny.usecairo=FALSE)`.

## 3.133 Make a random number generator

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Make a random number generator repeatable

```
repeatable(rngfunc, seed = runif(1, 0, .Machine$integer.max))
```

## Arguments

`rngfunc` The function that is affected by the R session's seed.

`seed` The seed to set every time the resulting function is called.

## Value

A repeatable version of the function that was passed in.

## Description

Given a function that generates random data, returns a wrapped version of that function that always uses the same seed when called. The seed to use can be passed in explicitly if desired; otherwise, a random number is used.

## Note

When called, the returned function attempts to preserve the R session's current seed by snapshotting and restoring

## Examples

```
rnormA <- repeatable(rnorm)
rnormB <- repeatable(rnorm)
rnormA(3) # [1] 1.8285879 -0.7468041 -0.4639111

[1] 1.5308819 0.9697510 0.5213101

rnormA(3) # [1] 1.8285879 -0.7468041 -0.4639111

[1] 1.5308819 0.9697510 0.5213101

rnormA(5) # [1] 1.8285879 -0.7468041 -0.4639111 -1.6510126 -1.4686924

[1] 1.5308819 0.9697510 0.5213101 1.0563204 0.1629233

rnormB(5) # [1] -0.7946034 0.2568374 -0.6567597 1.2451387 -0.8375699

[1] 0.2644419 0.2545899 -0.1941631 0.6304822 -0.7231721
```

## 3.134 Print message for deprecated

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Print message for deprecated functions in Shiny

```
shinyDeprecated(new = NULL, msg = NULL, old = as.character(sys.call(sys.parent()))[1L],  
  version = NULL)
```

## Arguments

- `new` Name of replacement function.
- `msg` Message to print. If used, this will override the default message.
- `old` Name of deprecated function.
- `version` The last version of Shiny before the item was deprecated.

## Description

To disable these messages, use `options(shiny.deprecation.messages=FALSE)` .

## 3.135 Collect information about the Shiny

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Collect information about the Shiny Server environment

`serverInfo()`

## Value

A list of the Shiny Server information.

## Description

This function returns the information about the current Shiny Server, such as its version, and whether it is the open source edition or professional edition. If the app is not served through the Shiny Server, this function just returns `list(shinyServer = FALSE)`.

## Details

This function will only return meaningful data when using Shiny Server version 1.2.2 or later.

## 3.136 Global options for Shiny

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Global options for Shiny

## Description

There are a number of global options that affect Shiny's behavior. These can be set with (for example) `options(shiny.trace=TRUE)` .

## Details

`shiny.launch.browser`

A boolean which controls the default behavior when an app is run. See [runApp](#) for more information.

`shiny.trace`

If `TRUE` , all of the messages sent between the R server and the web browser client will be printed on the console. This is useful for debugging.

`shiny.reactlog`

If `TRUE` , enable logging of reactive events, which can be viewed later with the [showReactLog](#) function. This incurs a substantial performance penalty and should not be used in production.

`shiny.usecairo`

This is used to disable graphical rendering by the Cairo package, if it is installed. See [plotPNG](#) for more information.

`shiny.maxRequestSize`

This is a number which specifies the maximum web request size, which serves as a size limit for file uploads. If unset, the maximum request size defaults to 5MB.



`shiny.suppressMissingContextError`

Normally, invoking a reactive outside of a reactive context (or `isolate()`) results in an error. If this is `TRUE`, don't error in these cases. This should only be used for debugging or demonstrations of reactivity at the console.

`shiny.host`

The IP address that Shiny should listen on. See [runApp](#) for more information.

`shiny.json.digits`

The number of digits to use when converting numbers to JSON format to send to the client web browser.

`shiny.error`

This can be a function which is called when an error occurs. For example, `options(shiny.error=recover)` will result a the debugger prompt when an error occurs.

`shiny.observer.error`

This can be a function that is called by an observer when an unhandled error occurs in it or an upstream reactive. By default, these errors will result in a warning at the console, and the websocket connection will close.

`shiny.table.class`

CSS class names to use for tables.

`shiny.deprecation.messages`

This controls whether messages for deprecated functions in Shiny will be printed. See [shinyDeprecated](#) for more information.

## 3.137 Create a Shiny app object

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Create a Shiny app object

```
shinyApp(ui = NULL, server = NULL, onStart = NULL, options = list(), uiPattern = "/")
```

```
shinyAppDir(appDir, options = list())
```

```
as.shiny.appobj(x)
```

```
"as.shiny.appobj"(x)
```

```
"as.shiny.appobj"(x)
```

```
"as.shiny.appobj"(x)
```

```
is.shiny.appobj(x)
```

```
"print"(x, ...)
```

```
"as.tags"(x, ...)
```

## Arguments

**ui** The UI definition of the app (for example, a call to `fluidPage()` with nested controls)

**server** A server function

**onStart** A function that will be called before the app is actually run. This is only needed for `shinyAppObj`,

since in the `shinyAppDir` case, a `global.R` file can be used for this purpose.

<code>options</code>	Named options that should be passed to the <code>runApp</code> call. You can also specify <code>width</code> and <code>height</code> parameters which provide a hint to the embedding environment about the ideal height/width for the app.
<code>uiPattern</code>	A regular expression that will be applied to each <code>GET</code> request to determine whether the <code>ui</code> should be used to handle the request. Note that the entire request path must match the regular expression in order for the match to be considered successful.
<code>appDir</code>	Path to directory that contains a Shiny app (i.e. a <code>server.R</code> file and either <code>ui.R</code> or <code>www/index.html</code> )
<code>x</code>	Object to convert to a Shiny app.
<code>...</code>	Additional parameters to be passed to <code>print</code> .

## Value

An object that represents the app. Printing the object or passing it to `runApp` will run the app.

## Description

These functions create Shiny app objects from either an explicit UI/server pair (`shinyApp`), or by passing the path of a directory that contains a Shiny app (`shinyAppDir`). You generally shouldn't need to use these functions to create/run applications; they are intended for interoperability purposes, such as embedding Shiny apps inside a knitr document.

## Details

Normally when this function is used at the R console, the Shiny app object is automatically passed to the `print()` function, which runs the app. If this is called in the middle of a function, the value will not be passed to `print()` and the app will not be run. To make the app run, pass the app object to `print()` or `runApp()`.

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  shinyApp(
    ui = fluidPage(
      numericInput("n", "n", 1),
      plotOutput("plot")
    ),
    server = function(input, output) {
      output$plot <- renderPlot( plot(head(cars, input$n)) )
    }
  )

  shinyAppDir(system.file("examples/01_hello", package="shiny"))

# The object can be passed to runApp()
app <- shinyApp(
  ui = fluidPage(
```

Create a Shiny app object

```
  numericInput("n", "n", 1),
  plotOutput("plot")
),
server = function(input, output) {
  output$plot <- renderPlot( plot(head(cars, input$n)) )
}
)

runApp(app)
}
```

Shiny is an RStudio project. © 2014 RStudio, Inc.

## 3.138 Evaluate an expression without a

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

# Evaluate an expression without a reactive context

`maskReactiveContext (expr)`

## Arguments

`expr` An expression to evaluate.

## Value

The value of `expr`.

## Description

Temporarily blocks the current reactive context and evaluates the given expression. Any attempt to directly access reactive values or expressions in `expr` will give the same results as doing it at the top-level (by default, an error).

## See also

`isolate`

## A Program listings to create this document

### A.1 recipe.txt

```
==== ../rNotes/shiny/20150906[recipe.txt] ====
```

1) for articles.pdf, reference.pdf, tutorial.pdf:

```
create from web page
get 2 levels
  stay on same server
  stay on same path

settings:
  UNCHECKED: no headers/footers
  CHECKED:   create bookmarks
```

2) generate memo.tex

```
final version:
  stg02.mkPdfSet.py --verb --stopaft 0

full version with extracted *.pdf and *.txt references:
  stg02.mkPdfSet.py --verb --stopaft 0 --srcid

test version, only first 25 pages, each section, with extracted *.pdf and *.txt references:
  stg02.mkPdfSet.py --verb --stopaft 25 --srcid
```

3) generate memo.pdf, with indexing

```
fulltex.bat
```

### A.2 stg02.mkPdfSet.py

```
#### \LGnotes\rNotes\shiny\20150906[stg02.mkPdfSet.py] ---- paginate, index, and consolidate Shiny *pdfs
```

```
import re
import os
import sys
import subprocess

import optparse

prsr = optparse.OptionParser()
prsr.add_option("--noburst", action="store_true", dest="noburst", default=False)
prsr.add_option("--verbose", action="store_true", dest="verbose", default=False)
prsr.add_option("--stopaft", action="store", dest="stopaft", default=10, type="int")
prsr.add_option("--srcid", action="store_true", dest="srcid", default=False)
prsr.add_option("--nodrop", action="store_false", dest="dodrop", default=True)

opts,args = prsr.parse_args()

if opts.noburst:
  sys.stderr.write(".... --noburst is TRUE: skipping pdftk burst ....\n")
else:
  sys.stderr.write(".... bursting articles.pdf ....\n")
  ret = subprocess.call("pdftk.exe articles.pdf burst output \\tmp\\2art_%04d.pdf ")

  sys.stderr.write(".... bursting tutorial.pdf ....\n")
  ret = subprocess.call("pdftk.exe tutorial.pdf burst output \\tmp\\1tut_%04d.pdf ")

  sys.stderr.write(".... bursting reference.pdf ....\n")
  ret = subprocess.call("pdftk.exe reference.pdf burst output \\tmp\\3ref_%04d.pdf ")

fh_out = open("memo.tex","w")
fh_out.write(r"""
\def\Orientation{\PDFPORTRAIT}
\input{startup.tex}
\input{\SDEstudies/SASmacro/TeXfiles/TeXPrologA.tex}
\def\LongTitleID{Shiny Consolidated Documentation Set}
\def\TitleStr{from shiny.rstudio.com}
\def\SectHeadStr{Shiny Tutorial and Articles}
\input{\SDEstudies/SASmacro/TeXfiles/TeXStyle.tex}
""")
```

```

\def\SectHead#1{%%%%
\markboth{\quad#1} \hfill \quad}
  {#1} \hfill \quad}
}%%

\setlength{\evensidemargin}{-.5in}
\setlength{\oddsidemargin}{-.5in}
\setlength{\topmargin}{-.8in}
\setlength{\textwidth}{7.5in}
\setlength{\textheight}{10.3in}

\usepackage{makeidx}
\makeindex
\input{\SDEstudies/SASmacro/TeXfiles/TeXPrologB.tex}

\printindex
\addcontentsline{toc}{section}{Index}

""")

#### indexpats maps regular expressions to index terms to be used in TeX
indexpats = {
  re.compile("action.?Button",re.I)      : r"\index{inputs!actionButton}"
, re.compile("checkboxGroupInput",re.I)    : r"\index{inputs!checkboxGroupInput}"
, re.compile("checkboxInput",re.I)         : r"\index{inputs!checkboxInput}"
, re.compile("dateInput",re.I)          : r"\index{inputs!dateInput}"
, re.compile("dateRangeInput",re.I)     : r"\index{inputs!dateRangeInput}"
, re.compile("numericInput",re.I)       : r"\index{inputs!numericInput}"
, re.compile("textInput",re.I)          : r"\index{inputs!textInput}"
, re.compile("selectInput",re.I)        : r"\index{inputs!selectInput}"
, re.compile("submitButton",re.I)       : r"\index{inputs!submitButton}"
, re.compile("clickId",re.I)            : r"\index{mouse!clickId}"
, re.compile("clientdata",re.I)         : r"\index{clientData}"
, re.compile("plotOutput",re.I)         : r"\index{outputs!plotOutput}"
, re.compile("dataTableOutput",re.I)    : r"\index{outputs!dataTableOutput}"
, re.compile("verbatimTextOutput",re.I) : r"\index{outputs!verbatimTextOutput}"
, re.compile("fluidPage",re.I)          : r"\index{fluidPage}"
, re.compile("fluidrow",re.I)           : r"\index{fluidrow}"
, re.compile("hoverId",re.I)            : r"\index{mouse!hoverId}"
, re.compile("HTML\\(")                 : r"\index{HTML}"
, re.compile("isolate",re.I)            : r"\index{isolate}"
, re.compile("navbarMenu",re.I)         : r"\index{navbarMenu}"
, re.compile("navbarPage",re.I)        : r"\index{navbarPage}"
, re.compile("outputOptions",re.I)      : r"\index{outputOptions}"
, re.compile("plotPNG",re.I)            : r"\index{plotPNG}"
, re.compile("renderDataTable",re.I)    : r"\index{renderDataTable}"
, re.compile("invalidateLater",re.I)    : r"\index{invalidateLater}"
, re.compile("conditionalPanel",re.I)   : r"\index{conditionalPanel}"
, re.compile("renderUI",re.I)           : r"\index{renderUI}"
, re.compile("RJSONIO",re.I)            : r"\index{RJSONIO}"
, re.compile("selectInput",re.I)        : r"\index{selectInput}"
, re.compile("session!\$clientData",re.I) : r"\index{session!clientData}"
, re.compile("session!\$output",re.I)    : r"\index{session!output}"
, re.compile("session!\$user",re.I)     : r"\index{session!user}"
, re.compile("session\\)")              : r"\index{session!argument}"
, re.compile("reactiveValues",re.I)     : r"\index{reactiveValues}"
, re.compile("tabPanel",re.I)           : r"\index{tabPanel}"
, re.compile("tabsetPanel",re.I)        : r"\index{tabsetPanel}"
, re.compile("tabsetPanel",re.I)        : r"\index{tabsetPanel}"
, re.compile("updateCheckboxGroupInput",re.I) : r"\index{updates!updateCheckboxGroupInput}"
, re.compile("updateCheckboxInput",re.I) : r"\index{updates!updateCheckboxInput}"
, re.compile("updateDateRangeInput",re.I) : r"\index{updates!updateDateRangeInput}"
, re.compile("updateDateInput",re.I)    : r"\index{updates!updateDateInput}"
, re.compile("updateSelectInput",re.I)   : r"\index{updates!updateSelectInput}"
, re.compile("updateNumericInput",re.I)  : r"\index{updates!updateNumericInput}"
, re.compile("updateTextInput",re.I)    : r"\index{updates!updateTextInput}"
, re.compile("updateTabsetPanel",re.I)   : r"\index{updates!updateTabsetPanel}"
, re.compile("validate",re.I)           : r"\index{validate}"
, re.compile("updateRadioButtons",re.I)  : r"\index{updates!updateRadioButtons}"
, re.compile("includeText",re.I)        : r"\index{includeText}"
, re.compile("jQuery",re.I)             : r"\index{jQuery}"
, re.compile("log.visualizer",re.I)     : r"\index{logging}"
, re.compile("MathJax",re.I)            : r"\index{MathJax}"
, re.compile("animate",re.I)            : r"\index{animate}"
, re.compile("xxxx",re.I)                : r"\index{xxxx}"          #### to copy and change xxxx
}

```

```

dropdict = { #### manually drop these burst pages
    "2art_0020.pdf" : 1
    , "2art_0023.pdf" : 1
}

pgcnt = 0
sctn = "Unk"
srchead = "Undef"
rgxAnchor = re.compile("OVERVIEW *TUTORIAL *ARTICLES *GALLERY *REFERENCE *DEPLOY *HELP")

for f in sorted(os.listdir("/tmp")) :
    if opts.verbose:
        sys.stderr.write(".... %s ....\n" % f)

    if opts.dodrop and f in dropdict:
        sys.stderr.write(".... dropping %s ....\n" % f)
        continue

    mtch = re.search("(2art|1tut|3ref).+([0-9]{4}).pdf$",f)
    if mtch:
        pgcnt += 1
        if opts.verbose:
            sys.stderr.write(".... mg1: %s mg2: %s ....\n" % (mtch.group(1), mtch.group(2)))
        if int(mtch.group(2)) == 1:
            clrpg = ""
            pgcnt = 1
            if mtch.group(1)=="1tut":
                fh_out.write(r"\clearpage\section{Tutorial}\SectHead{Shiny Tutorial} %s" % "\n")
                sctn = "tut"
            if mtch.group(1)=="2art":
                fh_out.write(r"\clearpage\section{Articles}\SectHead{Shiny Articles} %s" % "\n")
                sctn = "art"
            if mtch.group(1)=="3ref":
                fh_out.write(r"\clearpage\section{Function Reference}\SectHead{Shiny Function Reference} %s" % "\n")
                sctn = "ref"
        else:
            clrpg = r"\clearpage"

    if opts.stopaft > 0 and pgcnt > opts.stopaft:
        continue;

    sys.stderr.write("\n... page %d: %s ....\n" % (pgcnt,f))
    ret = subprocess.call("pdfcrop.exe --margins \"-1 -2 -1 -2\" --clip \\tmp\\%s \\tmp\\clp_%s " % (f,f))
    ret = subprocess.call("pdftotext -raw \\tmp\\%s" % f)

    fh_txt = open(("tmp/%s" % f).replace(".pdf",".txt"))
    pdftxt = [ln.rstrip() for ln in fh_txt.readlines()]
    jointxt = " ".join(pdftxt)

    txtlen = len(pdftxt)
    if txtlen > 3:
        if opts.srcid:
            srchead = "%s: %d lines" % (f.replace("_", "\\_"), txtlen)

    if rgxAnchor.search(jointxt):
        if sctn == "tut":
            ret = subprocess.call("pdfcrop.exe --margins \"-1 -68 -1 -1 \" --clip \\tmp\\%s \\tmp\\clp_%s " % (f,f))
            if opts.srcid == False:
                srchead = "%s: %s" % (pdftxt[2],pdftxt[3])
            else:
                srchead = "%s: %s [%s: %d lines]" % (pdftxt[2], pdftxt[3], f.replace("_", "\\_"), txtlen)

            if re.match("LESSON", pdftxt[2]):
                fh_out.write(r"%s\subsection{%s}%s" % (clrpg,pdftxt[2],pdftxt[3],"\n"))
            else:
                fh_out.write(r"%s\subsection{%s}%s" % (clrpg,pdftxt[2],"\n"))
            clrpg = ""

        if sctn == "art":
            ret = subprocess.call("pdfcrop.exe --margins \"-1 -68 -1 -1 \" --clip \\tmp\\%s \\tmp\\clp_%s " % (f,f))
            if opts.srcid == False:
                srchead = pdftxt[2]
            else:
                srchead = "%s [%s: %d lines]" % (pdftxt[2], f.replace("_", "\\_"), txtlen)

            fh_out.write(r"%s\subsection{%s}%s" % (clrpg,pdftxt[2],"\n"))

```



```

    clrpg = ""

if sctn == "ref":
    nbase = 1
    if rgxAnchor.search(pdftxt[nbase+1]):
        nbase = nbase + 1
    if re.search("OVERVIEW",pdftxt[1]) and re.search("HELP",pdftxt[7]):
        nbase = 7

    nbase = nbase+1
    ret = subprocess.call("pdfcrop.exe --margins \"-1 -68 -1 -1 \" --clip \\tmp\\%s \\tmp\\clp_%s " % (f,f))
    if opts.srcid == False:
        srchead = pdftxt[nbase]
    else:
        srchead = "%s [%s: %d lines]" % (pdftxt[nbase], f.replace("_", "\\_"), txtlen)

    mtch = re.match("([^(]+)", pdftxt[nbase])
    if mtch:
        ##### fh_out.write(r"%s\subsection{%s: %s}%s" % (clrpg,mtch.group(1),pdftxt[2],"\n"))
        fh_out.write(r"%s\subsection{%s}%s" % (clrpg,pdftxt[nbase].replace("_", "\\_"),12),"\n"))
    else:
        fh_out.write(r"%s\subsection{%s}%s" % (clrpg,pdftxt[nbase].replace("_", "\\_"),12),"\n"))
    clrpg = ""

    fh_out.write(r"%s\SectHead{%s}\includegraphics[scale=.95]{/tmp/clp_%s}%s" % (clrpg, srchead, f,"\n") )

    for xpr in indexpats.keys() :
        if xpr.search(jointxt):
            sys.stderr.write(".... %s ....\n" % indexpats[xpr])
            fh_out.write(indexpats[xpr])
    else:
        sys.stderr.write("++++ skipping page %d ++++\n" % pgcnt)
    fh_txt.close()

fh_out.write(r"""
\cleardoublepage\appendix\section{Program listings to create this document} \SectHead{programs}
\subsection{recipe.txt}\SectHead{recipe.txt}
{\footnotesize\importverbatim{recipe.txt}
\end{verbatim}}

\subsection{stg02.mkPdfSet.py}\SectHead{stg02.mkPdfSet.py}
{\footnotesize\importverbatim{stg02.mkPdfSet.py}
\end{verbatim}}

\subsection{fulltex.bat}\SectHead{fulltex.bat}
{\footnotesize\importverbatim{fulltex.bat}
\end{verbatim}}

\end{document}%s
""" % "\n")

fh_out.close()

```

### A.3 fulltex.bat

```
@echo ==== fulltex.bat ====
```

```

call latex memo
call latex memo
makeindex memo.idx
call latex memo
call latex memo
makeindex memo.idx
call latex memo
call latex memo
call latex memo

```