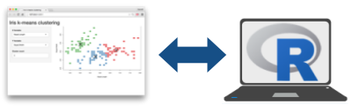
Interaktive Web-Anwendungen **mit shiny** Schummelzettel

Mehr auf shiny.rstudio.com

RStudio

Grundlagen

Eine **Shiny** Anwendung ist eine Internetseite mit Benutzeroberfläche (engl. **ui**, user interface) die mit einem Computer verbunden ist, auf dem eine aktive R-Session läuft (engl. **server**).



Anwender können die Benutzeroberfläche anpassen. Der Server führt dann R-Quellcode aus, der die Bildschirmanzeige der Benutzeroberfläche aktualisiert.

Mustervorlage

Eine neue Anwendung kann mit dieser Vorlage erstellt werden. Durch das Ausführen des R-Quellcodes in der Kommandozeile wird eine Vorschau angezeigt.

```
library(shiny)
ui <- fluidPage()
server <- function(input, output){}
shinyApp(ui = ui, server = server)
```

- **ui** – verschachtelte R-Funktionen, die die HTML-Benutzeroberfläche der Anwendung erzeugen.
- **server** – eine Funktion mit Anleitungen, wie die R-Objekte für die Benutzeroberfläche erstellt und aktualisiert werden sollen.
- **shinyApp** – verbindet `ui` und `server` zu einer funktionierenden Anwendung. Wird in **runApp()** gepackt um es innerhalb einer Funktion oder in einem Script auszuführen.

Freigeben der Anwendung

Ein einfacher Weg, die fertige Anwendung mit anderen zu teilen, ist, sie auf shinyapps.io (einem Cloud-basierten Service von RStudio) zu hosten.

1. Erstelle einen kostenlosen oder bezahlten Account auf **http://shinyapps.io**
2. Klicke auf die Schaltfläche **Publish** in der RStudio IDE (Version ≥ 0.99) oder führe folgendes aus: **rsconnect::deployApp(**"<Verzeichnispfad>"**)**

Erstellen oder kaufen eines eigenen Shiny Servers auf www.rstudio.com/products/shiny-server/

Anwendung erstellen – neue Argumente in fluidPage() und in der Server-Funktion einfügen

Eingabewerte in der Benutzeroberfläche mit ***Input()** Funktionen einfügen

Ausgabewerte mit ***Output()** Funktionen

Server mitteilen, wie die Ausgaben mit R in Server-Funktionen zu rendern sind. Anleitung:

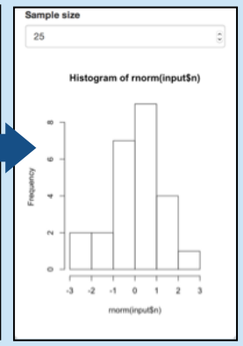
1. Ausgaben mittels **output$<id>** benennen
2. Auf Eingaben mittels **input$<id>** verweisen
3. Code in eine **render*()** Funktion packen bevor die Ausgabe gespeichert wird

```
library(shiny)

ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)

server <- function(input, output){
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}

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



Die Anwendung wird als eine Datei (**app.R**) oder als zwei Dateien (**ui.R** und **server.R**) gespeichert.

```
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
server <- function(input, output){
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui = ui, server = server)
```

```
# ui.R
fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
```

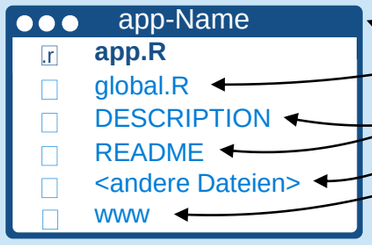
```
# server.R
function(input, output){
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
```

ui.R enhält alles, das zur Benutzeroberfläche zählt.

server.R endet mit der Server-Funktion.

shinyApp() muss nicht aufgerufen werden.

Jede Anwendung wird in einem eigenen Verzeichnis gespeichert. Dieses beinhaltet entweder **app.R** oder die zwei Dateien **server.R** und **ui.R**, optional auch weitere Dateien.



Der Name des Verzeichnisses ist der Name der Anwendung

(optional) legt Objekte fest, die für ui.R und server.R verfügbar sind

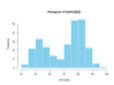
(optional) im Schaufenster verwendet

(optional) Daten, Skripten, etc.

(optional) Verzeichnis namens "**www**„ mit Dateien, die für den Webbrowser gedacht sind (Bilder, CSS, .js, usw.)

Starte Anwendungen mittels `runApp(<Verzeichnispfad>)`

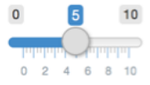
Ausgabewerte – render*() und *Output() Funktionen erstellen zusammen die R-Ausgabe des UI

	DT::**renderDataTable**(expr, options, callback, escape, env, quoted)	**dataTableOutput**(outputId, icon, …)
	**renderImage**(expr, env, quoted, deleteFile)	**imageOutput**(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)
	**renderPlot**(expr, width, height, res, …, env, quoted, func)	**plotOutput**(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)
	**renderPrint**(expr, env, quoted, func, width)	**verbatimTextOutput**(outputId)
	**renderTable**(expr,…, env, quoted, func)	**tableOutput**(outputId)
foo	**renderText**(expr, env, quoted, func)	**textOutput**(outputId, container, inline)
	**renderUI**(expr, env, quoted, func)	& **uiOutput**(outputId, inline, container, …) **htmlOutput**(outputId, inline, container, …)

arbeitet zusammen mit

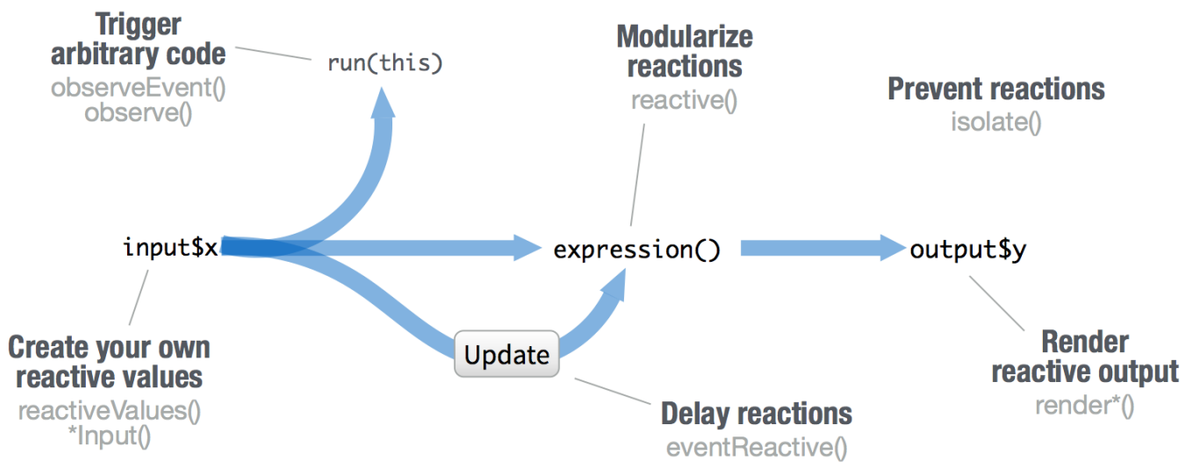
Eingabewerte – vom Benutzer erheben

Auf aktuelle Werte eines Eingabeobjektes wird mit **input$<inputId>** zugegriffen. Eingabewerte sind reaktionsfähig (engl. **reactive**).

Action	**actionButton**(inputId, label, icon, …)
Link	**actionLink**(inputId, label, icon, …)
☑ Choice 1 ☑ Choice 2 ☐ Choice 3	**checkboxGroupInput**(inputId, label, choices, selected, inline)
☑ Check me	**checkboxInput**(inputId, label, value)
	**dateInput**(inputId, label, value, min, max, format, startview, weekstart, language)
	**dateRangeInput**(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)
Choose File	**fileInput**(inputId, label, multiple, accept)
1	**numericInput**(inputId, label, value, min, max, step)
•••••••	**passwordInput**(inputId, label, value)
◉ Choice A ○ Choice B ○ Choice C	**radioButtons**(inputId, label, choices, selected, inline)
Choice 1 Choice 1 Choice 2	**selectInput**(inputId, label, choices, selected, multiple, selectize, width, size) (auch **selectizeInput()**)
	**sliderInput**(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)
Apply Changes	**submitButton**(text, icon) (verhindert automatische Reaktionen in der ganzen Anwendung)
Enter text	**textInput**(inputId, label, value)

Reaktionsfähigkeit

Reaktionsfähige Werte wirken zusammen mit reaktionsfähigen Funktionen. Um dem Fehler `Operation not allowed without an active reactive context.` vorzubeugen, werden reaktionsfähige Werte innerhalb der Argumente von reaktionsfähigen Funktionen aufgerufen.



Eigene reaktionsfähige Werte erstellen

```
library(shiny)

ui <- fluidPage(
 textInput("a","")
)

server <-
function(input,output){
 rv <- reactiveValues()
 rv$number <- 5
}

shinyApp(ui, server)
```

*Input() Funktionen

(siehe vorherige Seite)

Erstellt einen reaktionsfähigen Wert, der als **input$<inputId>** gespeichert wird.

reactiveValues(...)

Erstellt eine Liste reaktionsfähiger Werte, deren Werte man selbst festlegen kann.

Reaktionsfähige Ausgaben rendern

```
library(shiny)
ui <- fluidPage(
 textInput("a","")
)

server <-
function(input,output){
 output$b <-
  renderText({
   input$a
  })
}

shinyApp(ui, server)
```

render*() Funktionen

(siehe vorherige Seite)

Erstellt ein Objekt zur Anzeige. Führt Code im Funktionskörper nochmals aus um das Objekt neu zu erstellen, wenn sich ein reaktionsfähiger Wert im Code ändert. Speichert das Resultat in **output$<outputId>**.

Reaktionsfähigkeit verhindern

```
library(shiny)
ui <- fluidPage(
 textInput("a",""),
 textOutput("b")
)

server <-
function(input,output){
 output$b <-
  renderText({
   isolate({input$a})
  })
}

shinyApp(ui, server)
```

isolate(expr)

Führt einen Codeblock aus und gibt eine **nicht-reaktionsfähige** Kopie des Ausgabewertes zurück.

Eigenmächtichen Code triggern

```
library(shiny)
ui <- fluidPage(
 textInput("a",""),
 actionButton("go", "")
)

server <-
function(input,output){
 observeEvent(input$go,
   {print(input$a)})
}

shinyApp(ui, server)
```

observeEvent(eventExpr, handlerExpr, event.env, event.quoted, handler.env, handler.quoted, labe, suspended, priority, domain, autoDestroy, ignoreNULL)

Führt Quellcode im 2.Argument aus sobald sich reaktionsfähige Werte im 1.Argument ändern. Siehe **observe()** für Alternativen.

Reaktionen modularisieren

```
library(shiny)
ui <- fluidPage(
 textInput("a",""),
 textInput("z", "")
)

server <-
function(input,output){
 re <- reactive({
  paste(input$a,input$b)})
 output$b <- renderText({
   re()
  })
}

shinyApp(ui, server)
```

reactive(x, env, quoted, label, domain)

Erstellt **reaktionsfähigen Ausdruck**, der

- seinen Wert **zwischenspeichert** um Berechnungen zu reduzieren
- von anderem Code ausgerufen werden kann
- seine Abhängigkeiten verständigt wenn er ungültig wird.

Der Ausdruck wird mit Funktionssyntax aufgerufen, z. B. `re()`.

Reaktionen verzögern

```
library(shiny)
ui <- fluidPage(
 textInput("a",""),
 actionButton("go", "")
)

server <-
function(input,output){
 re <- eventReactive(
  input$go,{input$a})
 output$b <- renderText({
   re()
  })
}

shinyApp(ui, server)
```

eventReactive(eventExpr, valueExpr, event.env, event.quoted, value.env, value.quoted, label, domain, ignoreNULL)

Erstellt reaktionsfähigen Ausdruck mit Code im 2.Argument der nur ungültig wird, wenn sich reaktionsfähige Werte im 1.Argument ändern.

Benutzeroberfläche (UI)

Die Benutzeroberfläche einer Anwendung ist ein HTML-Dokument, erzeugt durch Shiny's Funktionen in R.

```
fluidPage(
  textInput("a","")
)
## <div class="container-fluid">
##  <div class="form-group shiny-input-container">
##     <label for="a"></label>
##     <input id="a" type="text"
##        class="form-control" value=""/>
##   </div>
## </div>
```

HTML ist Ausgabe

Einfügen statischer HTML Elemente mit **tags**, einer Liste von Funktionen die mit weitverbreiteten HTML tags vergleichbar sind, z. B. **tags$a()**. Unbenannte Argumente werden in die `tags` weitergereicht; benannte Argumente werden zu `tag`-Attributen.

tags$a	tags$data	tags$h6	tags$nav	tags$span
tags$abbr	tags$datalist	tags$head	tags$noscript	tags$strong
tags$address	tags$dd	tags$header	tags$object	tags$style
tags$area	tags$del	tags$hgroup	tags$ol	tags$sub
tags$article	tags$details	tags$hr	tags$optgroup	tags$summary
tags$aside	tags$dfn	tags$HTML	tags$option	tags$sup
tags$audio	tags$div	tags$i	tags$output	tags$table
tags$b	tags$dl	tags$iframe	tags$p	tags$tbody
tags$base	tags$dt	tags$img	tags$param	tags$td
tags$bdi	tags$em	tags$input	tags$pre	tags$textarea
tags$bdo	tags$embed	tags$ins	tags$progress	tags$tfoot
tags$blockquote	tags$eventsource	tags$kbd	tags$q	tags$th
tags$body	tags$fieldset	tags$keygen	tags$ruby	tags$thead
tags$br	tags$figcaption	tags$label	tags$rp	tags$time
tags$button	tags$figure	tags$legend	tags$rt	tags$title
tags$canvas	tags$footer	tags$li	tags$s	tags$tr
tags$caption	tags$form	tags$link	tags$samp	tags$track
tags$cite	tags$h1	tags$mark	tags$script	tags$u
tags$code	tags$h2	tags$map	tags$section	tags$ul
tags$col	tags$h3	tags$menu	tags$select	tags$var
tags$colgroup	tags$h4	tags$meta	tags$small	tags$video
tags$command	tags$h5	tags$meter	tags$source	tags$wbr

Die haufigsten `tags` haben Wrapper-Funktionen. Ihre Namen müssen **tags$** nicht voranstellen.

```
ui <- fluidPage(
  h1("Header 1"),
  hr(),
  br(),
  p(strong("bold")),
  p(em("italic")),
  p(code("code")),
  a(href="", "link"),
  HTML("<p>Raw html</p>")
)
```

CSS Dateien werden eingeschlossen durch **includeCSS()** oder durch

1. Datei einfügen im Unterverzeichnis **www**
2. Link zur Datei einfügen mit

```
tags$head(tags$link(rel = "stylesheet",
  type = "text/css", href = "<file name>"))
```

JavaScript wird eingefügt mittels **includeScript()** oder durch

1. Datei einfügen im Unterverzeichnis **www**
2. Link zur Datei einfügen mit

```
tags$head(tags$script(src = "<file name>"))
```

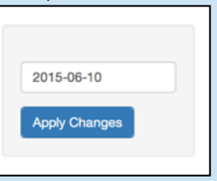
Bilder werden inkludiert durch

1. Datei einfügen im Unterverzeichnis **www**
2. Link zur Datei einfügen mit `img(src="<file name>")`

Layout

Mit einer Paneel-Funktion werden Elemente zu einem einzelnen Element kombiniert, das seine eigenen Eigenschaften hat, z. B.

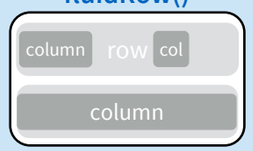
```
wellPanel(
  dateInput("a", ""),
  submitButton()
)
```



absolutePanel()	inputPanel()	tabPanel()
conditionalPanel()	mainPanel()	tabsetPanel()
fixedPanel()	navlistPanel()	titlePanel()
headerPanel()	sidebarPanel()	wellPanel()

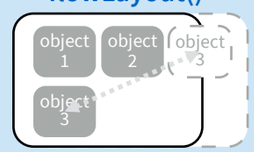
Paneele und Elemente werden mit Layout-Funktionen angeordnet. Elemente sind Eingabeargumente dieser Funktionen.

fluidRow()




```
ui <- fluidPage(
  fluidRow(column(width = 4),
    column(width = 2, offset = 3)),
  fluidRow(column(width = 12))
)
```

flowLayout()

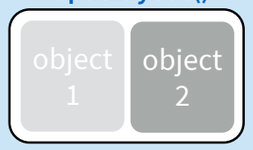



```
ui <- fluidPage(
  flowLayout( # object 1,
              # object 2,
              # object 3
  )
)
```

sidebarLayout()

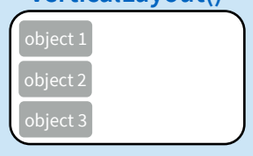

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

splitLayout()




```
ui <- fluidPage(
  splitLayout( # object 1,
               # object 2
  )
)
```

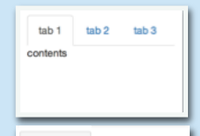
verticalLayout()




```
ui <- fluidPage(
  verticalLayout( # object 1,
                  # object 2,
                  # object 3
  )
)
```

Man kann tabPaneele übereinander legen und zwischen ihnen navigieren.

```
ui <- fluidPage( tabsetPanel(
    tabPanel("tab 1", "contents"),
    tabPanel("tab 2", "contents"),
    tabPanel("tab 3", "contents")))
```




```
ui <- fluidPage( navlistPanel(
    tabPanel("tab 1", "contents"),
    tabPanel("tab 2", "contents"),
    tabPanel("tab 3", "contents")))
```


```
ui <- navbarPage(title = "Page",
    tabPanel("tab 1", "contents"),
    tabPanel("tab 2", "contents"),
    tabPanel("tab 3", "contents"))
```

Weitere Schummelzettel auf
http://www.rstudio.com/resources/cheatsheets/