DASH-IN web-based analyses - TUTORIAL

For *Debian jessie* and *Mac OS X*

By Rosario Lombardo, The Microsoft Research – University of Trento (COSBI)
written by Rosario Lombardo and Fabio Moriero (COSBI)

What we'll learn in this tutorial:

- Overview of the Dash-In infrastructure
- Installing the required software components

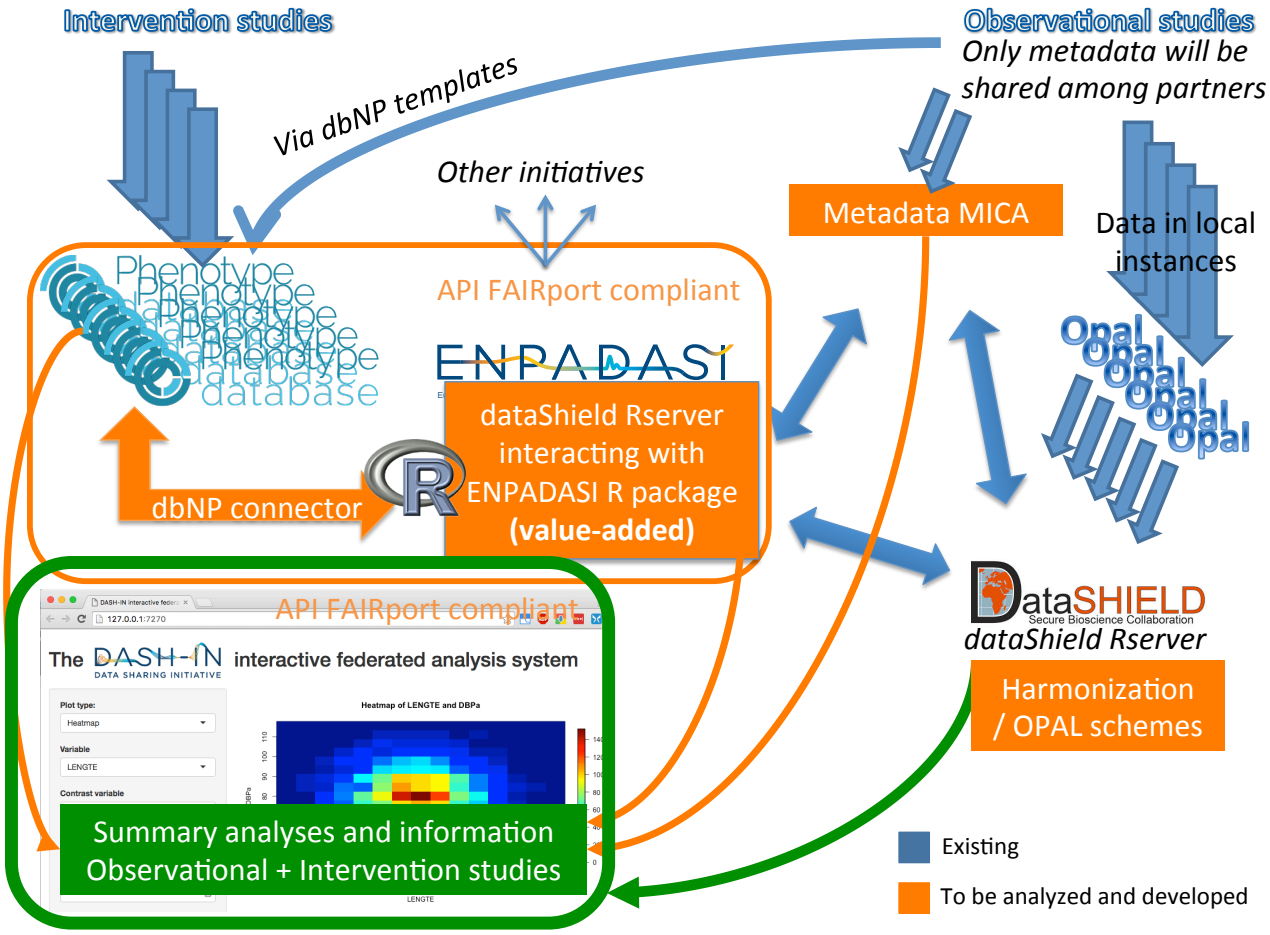
- Part 1
  - Creating a Shiny app
  - Structure of a Shiny app
- Part 2 – Creating dynamic UIs
- Part 3 – Linking into the Dash-In infrastructure
- Part 4 – The final application

- Server configuration and deployment of a multi-application server

Extensive references on Shiny can be obtained from http://shiny.rstudio.com/

Overview of the Dash-In infrastructure

As discussed in the Workshop the orange boxes are being examined and in this tutorial we'll examine the green parts below starting from zero to a working web application.



Installing the required software components

DEBIAN

RStudio Desktop

Is the software that helps us working with R – hence with shiny as well.

Install gdebi (used to install RStudio server and the shiny server)

```
$ sudo apt-get install gdebi-core
```

Download and install RStudio Desktop:

```
$ wget https://download1.rstudio.org/rstudio-0.99.902-amd64.deb

$ sudo gdebi rstudio-0.99.902-amd64.deb
```

If Debian is not your OS, then you can the other binaries or the sources here: https://www.rstudio.com/products/rstudio/download/

Install R and Shiny

Add the CRAN repository to get the latest version of R. In this tutorial we use the GARR repository, but you should choose the one that best fits you: https://cran.r-project.org/mirrors.html

Add the following statement in the file /etc/apt/sources.list.d/cran.list

```
deb http://cran.mirror.garr.it/mirrors/CRAN/bin/linux/debian jessie-cran3/
```

Then add the key for this Debian archive:

```
$ sudo apt-key adv --keyserver keys.gnupg.net --recv-key 381BA480
```

And update the packages list:

```
$ sudo apt-get update
```

Install R from the command line:

```
$ sudo apt-get install r-base
```

Then install the shiny package from either the command line:

```
$ sudo R -e "install.packages('shiny', repos='https://cran.rstudio.com/')"
```

Or, from the R prompt:

```
> install.packages('shiny', repos='https://cran.rstudio.com/')
```

MAC

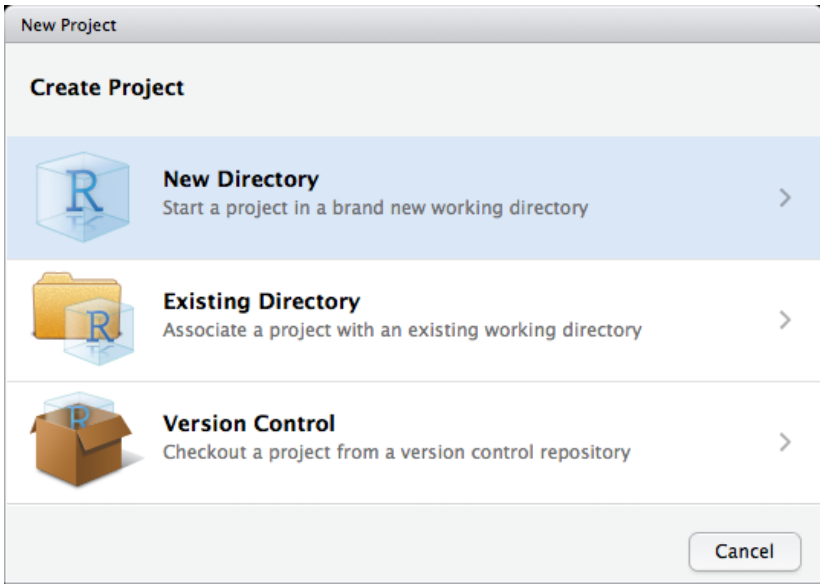
On Macs it is enough to download the latest R version from one of the mirrors at https://cran.r-project.org/mirrors.html and the latest RStudio version from https://www.rstudio.com/products/rstudio/download/.

Tutorial – Part 1

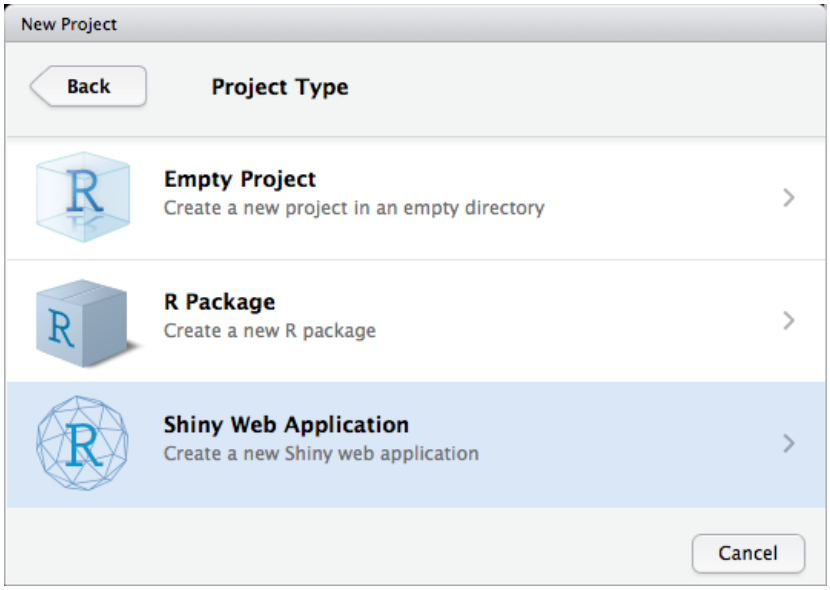
http://188.166.1.102/hackaton/part1

Creating a new Shiny App

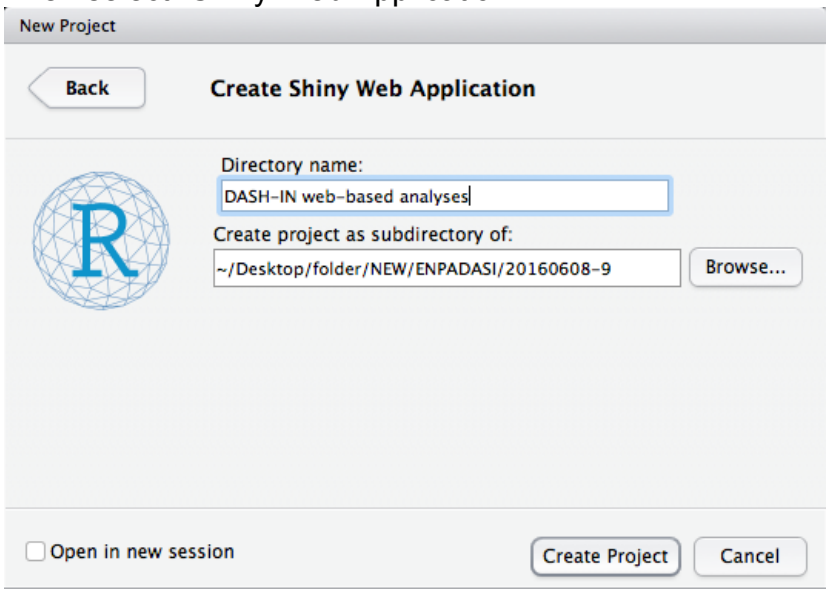
From RStudio menu: File > New Project…



Click on “New Directory”



Then select “Shiny Web Application”



Enter the project name in the “Directory name” field and select where you want your new project folder to be created.

The new project created in RStudio is composed of three files:

- **ui.R**: defines the user interface
- **server.R**: defines the server logic
- **shinyapp.Rproj:** a RStudio project file, not needed by the Shiny application

You can run the newly created shiny application either:

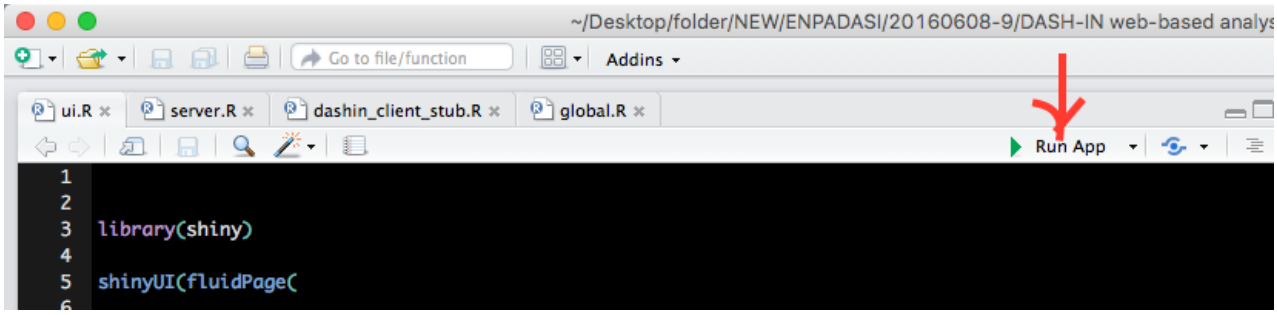
A. by hand
    1. launch R:

```
$ R
```

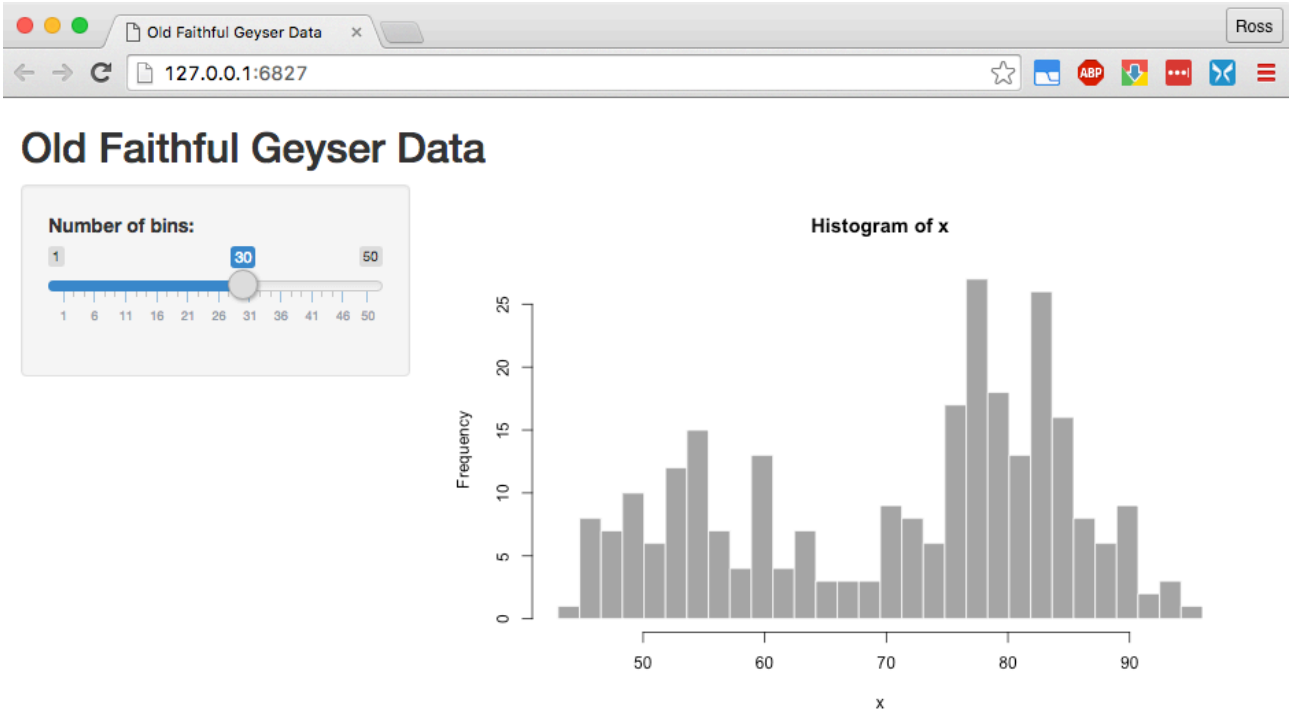
2. Then type:

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

B. or from RStudio, by clicking on the "Run App" button

1. 

And this is the web application than gets launched in the web browser:



Structure of a Shiny app

A shiny app is composed of two files:

- **ui.R**: defines the user interface
- **server.R**: defines the server logic

ui.R

Our **ui.R** file begins with:

```
library(shiny)
```

which loads the shiny library.

All the magic in this file happens inside one function:

```
shinyUI(...)
```

which contains other functions (with quite self-explanatory names):

- ***fluidPage*** it creates a fluid page
- ***titlePanel*** it renders the title of the page
- ***sidebarLayout*** it declares the structure of the page.
- ***sidebarPanel*** it allows to put the desired controls in a lateral panel
- ***mainPanel*** it allows to put the desired controls in the main panel

```
shinyUI(fluidPage(

  # Application title
  titlePanel("Old Faithful Geyser Data"),

  # Sidebar with a slider input for 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")
    )
  )
))
```

As we can see, these functions are structured in a way that defines the layout of the page. Then, there are the ***titlePanel***, ***sliderInput*** and ***plotOutput*** functions that define actual objects for that page – respectively, a title, a slider control for the user and a plot showing some data.

Common **HTML tags** can be used, you just need to specify one of the a shiny functions (there's a list at: shiny.rstudio.com/tutorial/lesson2) that map to the HTML tags.
e.g. the code:

```
h1("Title")
```

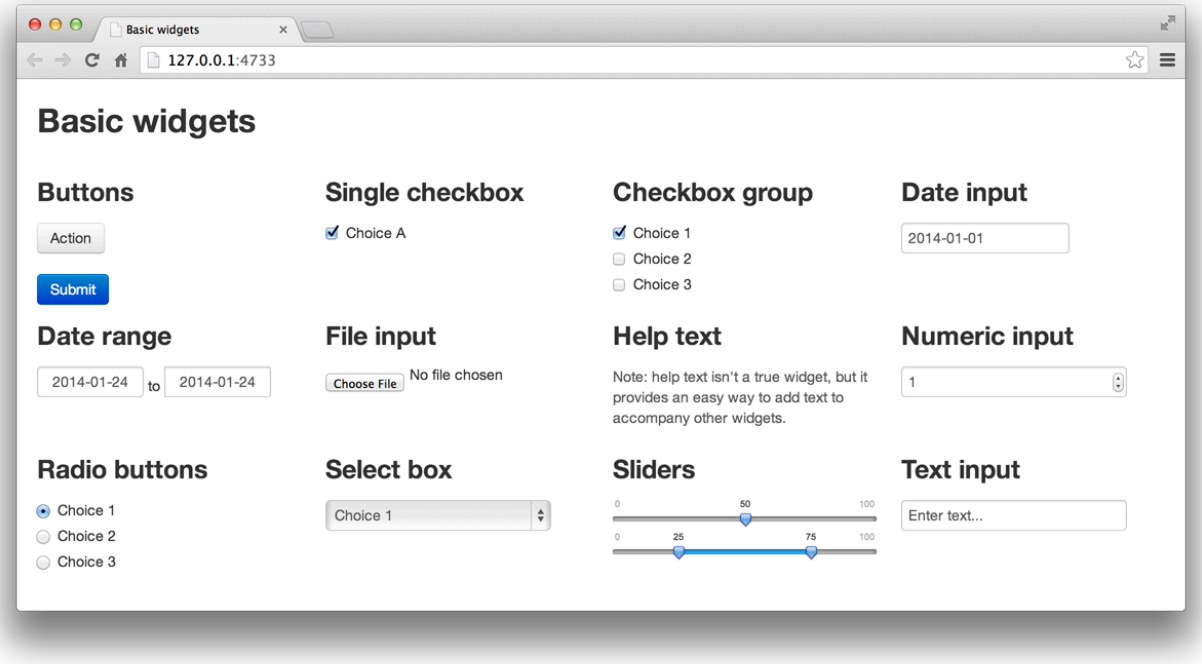
```
p("Text", style = "font-family: 'times'")
```

will result in:

```
<h1>Title</h1>
<p style="font-family: 'times'">Text</p>
```

Control widgets are available (check them at: shiny.rstudio.com/tutorial/lesson3). In our example, we are using the *sliderInput*. Others are:

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



server.R

Let's take a look at the **server.R** file.

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

  output$distPlot <- renderPlot({

    # generate bins based on input$bins from ui.R
    x    <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white')

  })
})
```

This file is composed of one single function as well:

```
shinyServer(...)
```

which takes an anonymous function as an argument

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

Notice the two arguments of the anonymous function:

- ***input*** is a list-like object containing the input elements we have in **ui.R** – that is, for our example, the ***sliderInput*** object.
- ***output*** is a list-like object containing the output elements from **ui.R** – in our example, ***plotOutput***

So, what's happening in our **server.R** file? As it can be below, we are using a ***renderPlot*** function to draw some plot in a ***distPlot*** object: that *distPlot* name refers to the **id** we gave to our ***plotOutput*** element in the **ui.R** file.

ui.R	**server.R**
plotOutput("distPlot") ...	output$distPlot <- renderPlot({ ...

The other way around, looking inside the ***renderPlot*** function we see that ***input$bins*** is used in some calculation: we are using the value of the ***sliderInput*** object with **id** ***bins***.

The important thing here, is that the shiny framework takes care of updating all these values in real-time, as, e.g., the user changes values in the input controls.

The unnamed function returns a list-like object named ***output*** that contains the code needed to update the R objects in the app: each R object needs to have its own entry in that list.

To add an entry, use one of the functions prefixed with “render”; e.g.:

```
output$text1 <- renderText({ "Example text" })
```

More render functions are:

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

In order to use the values of the UI objects, you need to use the ***input*** objects – which is similar to the *output* object.
As an example, we can have a label always updated with the text inserted by the user by just writing this:

ui.R	server.R

```
shinyUI(fluidPage(
mainPanel(
        selectInput("var",
        label = "Choose a variable",
        choices = c("A", "B", "C"),
        selected = "A"),
        textOutput("text1")
)
))
```

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

Here we see an important concept: ***reactivity***, which is the ability of a shiny app to take *input values* from a web page, make them available to R and have the results back as *output values* on the web page. These input and output values are bound and changes to the former are immediately reflected on the latter.

This is achieved by using reactive programming: it all starts with **reactive values** – that can change over time or in response to the user interaction – and these values are given to **reactive expressions**, which can execute other reactive expressions; so that, whenever a change occurs on the reactive values, the reactive expressions using them are re-executed.

- Reactive values are often ***input*** objects
- Reactive expressions are created by passing a normal expression into the ***reactive*** function

Tutorial – Part 2 – Creating dynamic UIs

http://188.166.1.102/hackaton/part2

Let's add some dynamism: in this part of the tutorial we are going to modify the previous sources in order to have a dynamic user interface that changes accordingly to the user interaction.

In **ui.R** we add new controls: ***selectInput***; for example, the code below add a control for selecting the type of plot that has to be drawn, choosing between two elements of a list we define:

```
selectInput("plotType",
            "Plot type:",
            list("Histogram" = "hist",
                 "Contour Plot" = "contour"))
```

Another kind of control we are introducing now is the ***conditionalPanel***, which will draw the controls given to it as parameters only if a given (javascript) condition is true; in the piece of code below, another *selectInput* is added only if the *selectInput* with id *plotType* selects the element "hist":

```
conditionalPanel(
      condition = "input.plotType != 'hist'",
      selectInput("var_y",
                  "Contrast variable",
                  list("var1", "var2")))
```

The new **ui.R**:

```
library(shiny)


shinyUI(fluidPage(

  # Application title
  titlePanel("Old Faithful Geyser Data"),

  # Sidebar with a slider input for number of bins
  sidebarLayout(
    sidebarPanel(

      selectInput("plotType",
```

```
            "Plot type:",
            list("Histogram" = "hist",
                 "Contour Plot" = "contour")
    ),
    selectInput("var_x",
            "Variable",
            list("var1", "var2")
    ),

    conditionalPanel(
      condition = "input.plotType != 'hist'",

      selectInput("var_y",
                  "Contrast variable",
                  list("var1", "var2")
      )
    ),


    conditionalPanel(
      condition = "input.plotType == 'hist'",

      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    )

  ),

  # Show a plot of the generated distribution
  mainPanel(
    plotOutput("distPlot")
  )
 )
))
```

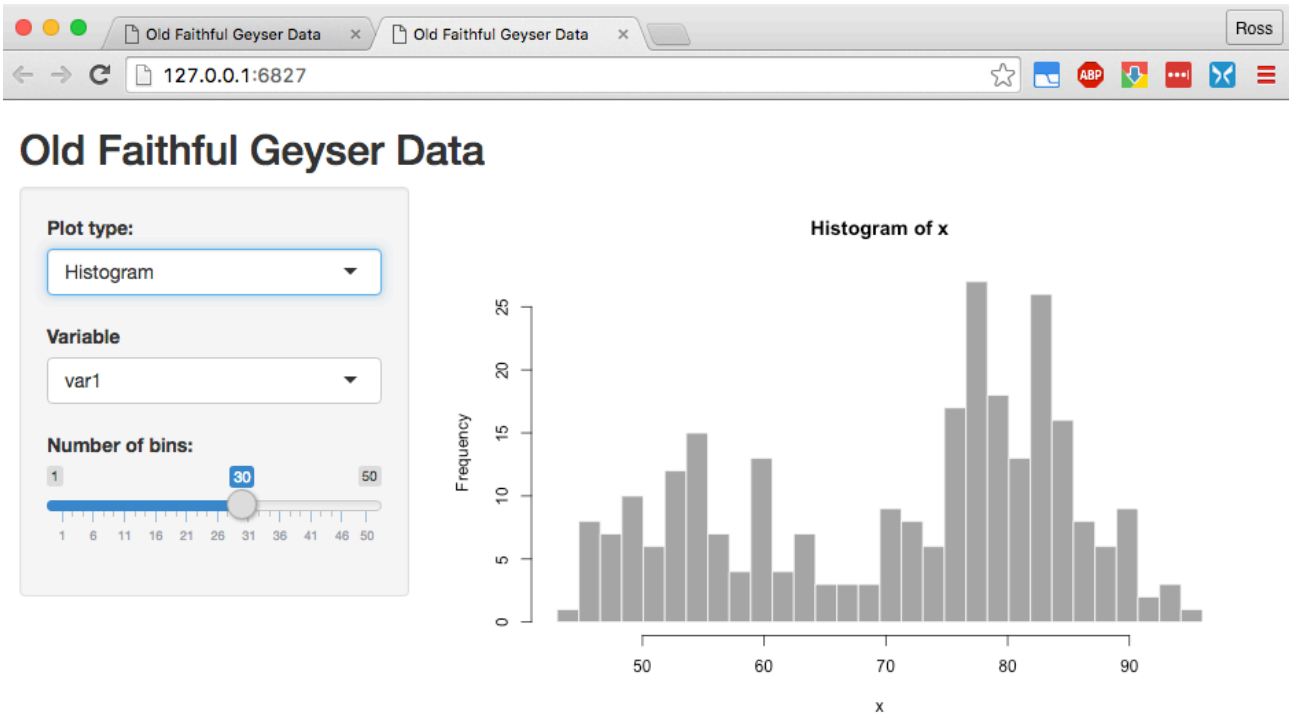
As for the **server.R** file, we need to add the logic to make the magic happen: we add an *if-else* construct that draws either one or the other kind of plots, depending on the value of the "plotType" *selectInput*.
So, if the "histogram" plot is selected, then the plot from the previous example will be drawn; otherwise, we will draw a new kind of plot.

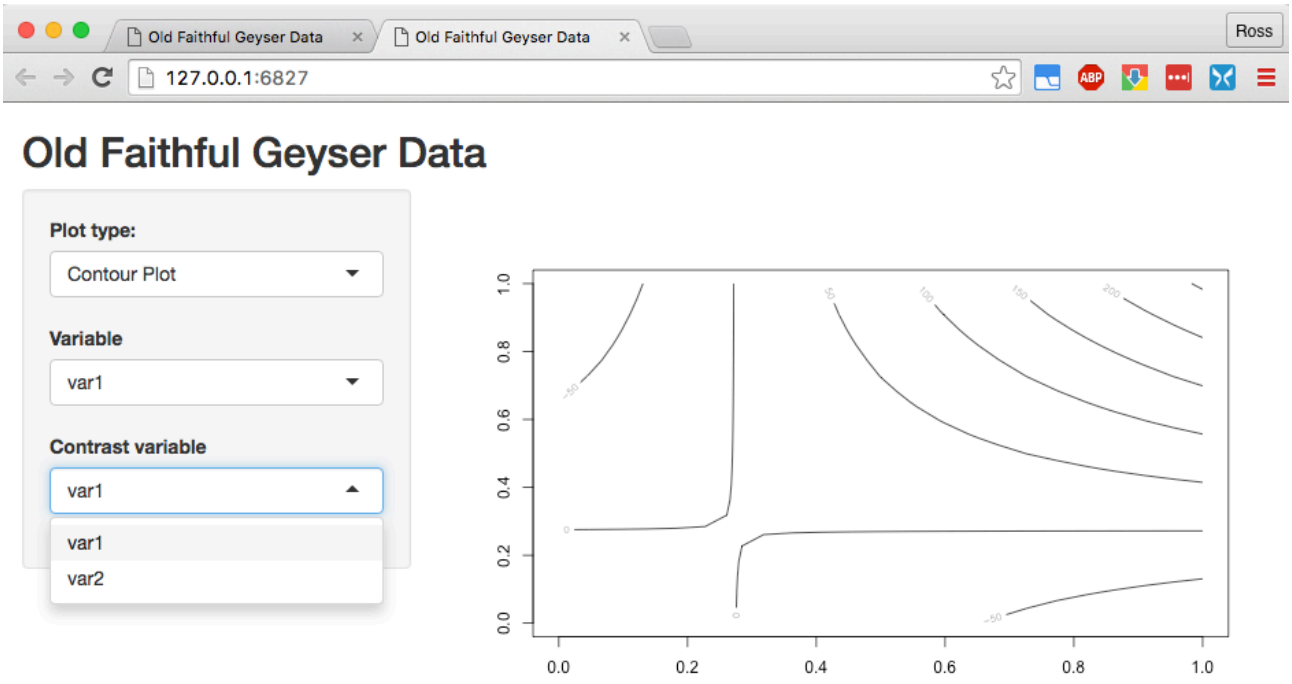
The new **server.R**:

```
library(shiny)
```

```
shinyServer(function(input, output) {
  output$distPlot <- renderPlot({

    if ( input$plotType == "hist") {

      # generate bins based on input$bins from ui.R
      x    <- faithful[, 2]
      bins <- seq(min(x), max(x), length.out = input$bins + 1)

      # draw the histogram with the specified number of bins
      hist(x, breaks = bins, col = 'darkgray', border = 'white')



    }
    else if ( input$plotType == "contour") {

      if (input$var_x == "var1") {
        x <- -6:16
      }
      else {
        x <- 20:30
      }

      if (input$var_y == "var1") {
        y <- -6:16
      }
      else {
        y <- 20:30
      }

      contour(outer(x, y), method = "edge")
    }

  })
})
```

The following are the two kind of plots rendered on the web from R analysis:





Read and try this code, and observe how the "var1" and "var2" variables are used in the plot. Again, the **ui.R** has some "logic" to hide or show the "contrast variable"; while the **server.R** uses their values for the "contour" plot.

We can go further in dynamic pages by having UI controls filled with custom information. For example, in the previous example, the *selectInput* control takes a list argument – *list("var1", "var2")*.

```
selectInput("var_x",
            "Variable",
            list("var1", "var2"))
```

We can define a function returning a list:

```
selectInput("var_x",
            "Variable",
            get_study_variables())


get_study_variables <- function() {
   return( list("var1", "var2") )
}
```

Please note that this function will have the code needed to, e.g., connect to a remote database and fetch some data; so that our *selectInput* control is created with the elements from an external service and this lays down the basis for building interactive web-based analyses for the Dash-In infrastructure.

Tutorial – Part 3 – Linking into the Dash-In infrastructure

http://188.166.1.102/hackaton/part3

In the last section of this tutorial, we saw a (very simple) way of having a dynamic user interface with functions allowing to potentially fetch data from external services before populating the UI controls in the web page. Now we are going to see how to use external data in our shiny application.

Let's introduce a new file, ***global.R*** – whatever is declared in this file, it is parsr first of any other Shiny file and it also accessible from both *ui.R* and *server.R* files – so let's put the definition of our functions there.

Below we see the complete *global.R*, ready to interact with the Dash-In infrasctrure and namely the DataShield system. Most of these commands have been covered in previous Datashield tutorials of the Hackaton so let's briefly say that the first commands perform a distributed login across all the sites from which we want to fetch data.

As prerequisites the following DataShield R packages should be installed system-wide (Debian):

```
sudo apt-get install r-cran-rjson
sudo apt-get install libcurl4-gnutls-dev libcurl4-openssl-dev
```

in R console:

```
install.packages('RCurl', repos='http://cloud.r-project.org', dependencies=TRUE)
```

Additionally the following packages need to be installed on any OS:

```
install.packages('opaladmin', repos='http://cran.obiba.org', dependencies=TRUE)
install.packages('dsBaseClient', repos=c(getOption('repos'), 'http://cran.obiba.org'), dependencies=TRUE)
install.packages('dsModellingClient', repos=c(getOption('repos'), 'http://cran.obiba.org'), dependencies=TRUE)
install.packages('dsStatsClient', repos=c(getOption('repos'), 'http://cran.obiba.org'), dependencies=TRUE)
install.packages('dsGraphicsClient', repos=c(getOption('repos'), 'http://cran.obiba.org'), dependencies=TRUE)
```

NOTE: *soon also an ENPADASI R package will be needed to fully connect the Dash-In infrastructure.*

The focus in this tutorial is the definition of the ***get_study_variables()*** function – as a demonstration of the web-based interactive analysis system offered within the Dash-In infrastructure for both intervention and observational studies.

global.R

```
library(opal)
library(dsBaseClient)
library(dsStatsClient)
library(dsGraphicsClient)
library(dsModellingClient)

##
## DATASHIELD commands

# load the login file
my_login<-read.table('../logins.txt', sep="", header=TRUE)
# log in to the remote servers
# assign=TRUE will have the remote opal server instruct the remote R
# instance to assign the dataframe into variable 'D'
opals <- datashield.login(logins=my_login, assign=TRUE, symbol = 'D')

# detect the list of variables in the study
get_study_variables <- function(symbol="D") {
```

```
  tryCatch({
     ds.colnames(x=symbol)[[1]]
   }, error = function(e) {
     print(e)
     return( list("No data was loaded! See error messages!") )
   }
  )
}
```

In the logins.txt files a list of different OPAL and DBNP DataShield-enabled servers can be entered. For the tutorial we'll use a guest account created on the RECAS Opal instance in Bari:

logins.txt

```
server  url     user    password        table
OpalRecas       http://90.147.170.46:8080       enpadasi.guest1 Et6w23AA        LifeLines.Lif
eLines
```

The ***get_study_variables()*** function fetches the variables in the study. It also handles some error condition, for example no internet connection or remote servers not reachable.

At the same time we extend the application with all DataShield supported plots, i.e. **histogram**, **contourPlot** and **heatmap**.

note*: since the UI is fetching the data from remote, it may take a short while for the page to load.*

In our new ***ui.R***, we replace the static lists with "var1" and "var2" with the new defined function, and we also removed the *sliderInput* for the number of bins, since we don't need it anymore. The new file is now this:

ui.R

```
library(shiny)

shinyUI(fluidPage(

  # Application title
  titlePanel("Old Faithful Geyser Data"),
```

```
 # Sidebar with a slider input for number of bins
 sidebarLayout(
  sidebarPanel(

    selectInput("plotType",
                "Plot type:",
                list("Histogram" = "hist",
                     "Contour Plot" = "contour",
                     "Heatmap" = "heatmap")
  ),
  selectInput("var_x",
              "Variable",
              get_study_variables()
  ),

  conditionalPanel(
    condition = "input.plotType != 'hist'",

    selectInput("var_y",
                "Contrast variable",
                get_study_variables()
    )
  )

 ),

 # Show a plot of the generated distribution
 mainPanel(
  plotOutput("distPlot")
 )
)
))
```

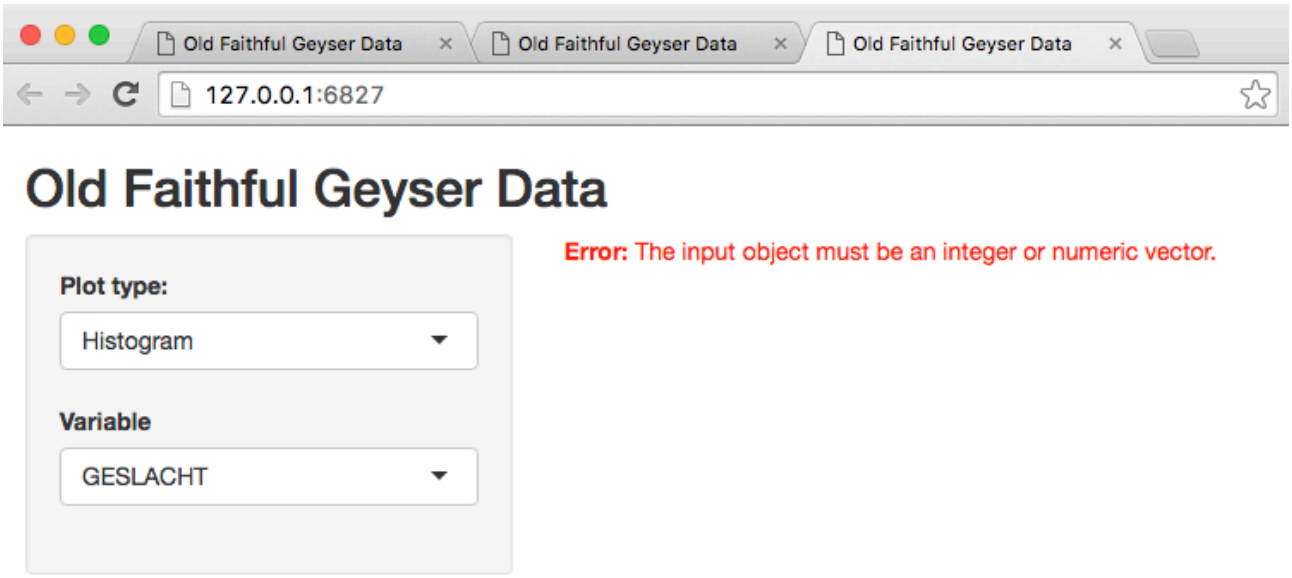
We need to modify ***server.R*** for drawing the plots using the data fetched from the Dash-In infrastructure. In particular we use the DataShield functions that have been previously tied to the correct data providers (opal / phenotype database).

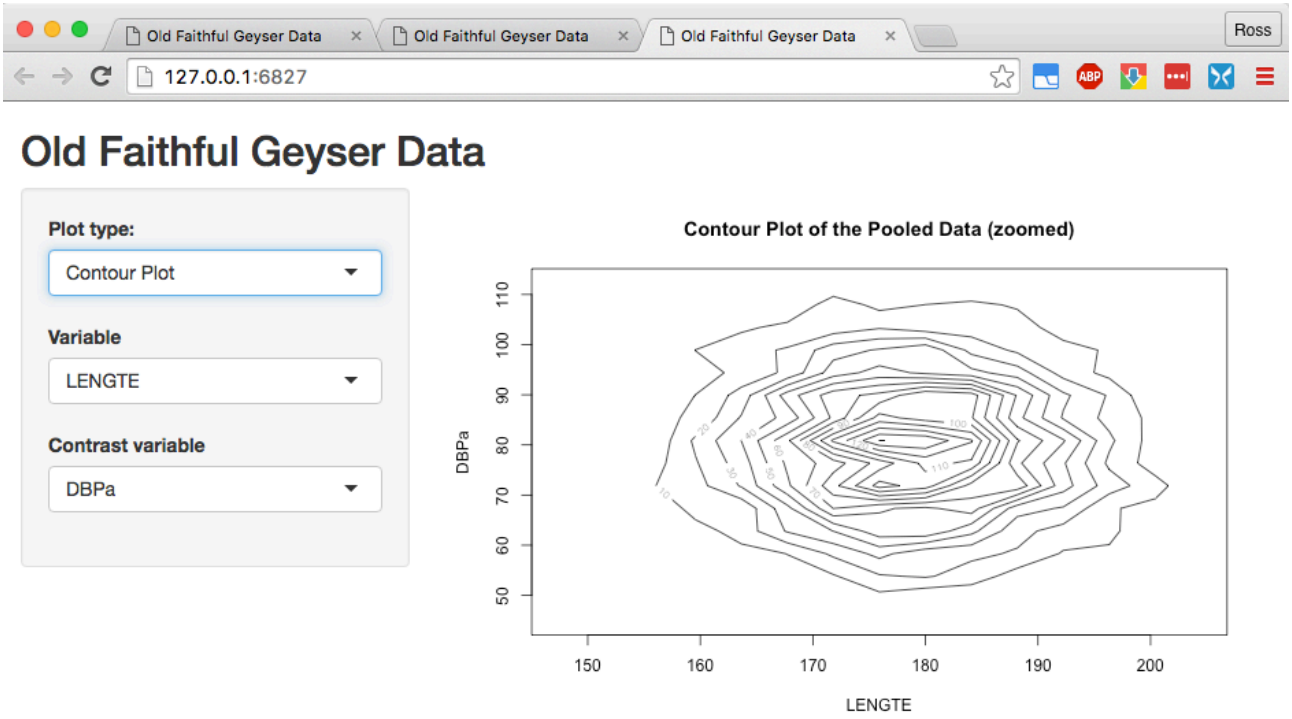
Note how the input variables for DataShield are created from the selected UI.

server.R

```
library(shiny)

shinyServer(function(input, output) {
  output$distPlot <- renderPlot({

    if ( input$plotType == "hist") {

      ds.histogram(x = paste0("D$", input$var_x))

    }
    else if ( input$plotType == "contour") {

      ds.contourPlot(x = paste0("D$", input$var_x),
                     y = paste0("D$", input$var_y),
                     show = "zoomed"
      )

    } else if ( input$plotType == "heatmap") {

      ds.heatmapPlot(x = paste0("D$", input$var_x),
                     y = paste0("D$", input$var_y),
                     show = "zoomed"
      )

    }
  })
})
```

note*: for our convenience, since this is just a demo, we are not doing the needed checks over the selected variable(s) that are passed to the plot – for this reason, some variable-plot combinations will produce an error line instead of drawing a result.*





Tutorial – Part 4 - The final application

http://188.166.1.102/hackaton/part4

Let's get step-by-step to the final application.

First, let's get rid of that "Old Faithful Geyser Data" title and let's have a more dynamic one, using also an image.

All the images have to be located in a ***www*** directory, which has to be at the same directory level of *ui.R* (and others). In that directory, let's put an image: "dash-in-png".

***note**: You may download the Dash-In logo from the part4 link above.*

As we have seen at the beginning of this tutorial, shiny offers some functions that map to HTML tags: one of these functions is ***img***:

```
img(src="dash-in.png", width="250px")
```

Let's use this image in our title, like this:

```
titlePanel(title = "", windowTitle = "DASH-IN interactive federated analysis system"),
h1("The", img(src="dash-in.png", width="250px"), "interactive federated analysis system")
```

Now let's add some customization to the labels of our plots. First of all, we add a bunch of controls in *ui.R*, for the user to (optionally) type the labels for the plots. We put these controls in a div in order to add some style to them; then, notice that we are adding only one *input* control, while the last two are *output* objects: what we are going to do with them is using the ***renderUI()*** functions in *server.R* to (kind of) "inject" the dynamically created *input* controls using the ***uiOutput()*** function in ui.R.

ui.R

```
div(style="font-size: .9em",

        hr(style="border-top-color: #aaa"),
        helpText("You may specify custom wording in the plot before exporting for publicati
on."),

        textInput("title", "title", ""),
        uiOutput("xlabel"),
        uiOutput("ylabel")

)
```

server.R

```
output$xlabel <- renderUI({
   textInput("xlabel", paste0("x label (for variable ", input$var_x, ")") , "")
 })


 output$ylabel <- renderUI({
   if ( input$plotType == "hist" ) {
     textInput("ylabel", "y label (for frequency)", "")
```

```
   } else {
     textInput("ylabel", paste0("y label (for variable ", input$var_y, ")") , "")
   }

 })
```

At this point we have the controls, but they will not react with the labels yet. Let's make them useful by editing a little bit more *server.R*:

- for the histogram

```
plot(x = h,
 main = ifelse(input$title  != "", input$title, paste("Histogram of", input$var_x)),
 xlab = ifelse(input$xlabel != "", input$xlabel, input$var_x),
 ylab = ifelse(input$ylabel != "", input$ylabel, "Frequency"))
```

- for the contour

```
title(main = ifelse(input$title != "",
 input$title,
 paste("Correlation of", input$var_x, "and", input$var_y)),
      col.main="black")
 mtext(ifelse(input$xlabel != "", input$xlabel, input$var_x), side=1, line=3, col = "black")
 mtext(ifelse(input$ylabel != "", input$ylabel, input$var_y), side=2, line=3, col = "black")
```

For convenience, we show here the complete files:

ui.R

```
library(shiny)


shinyUI(fluidPage(


  # Application title
  titlePanel(title = "", windowTitle = "DASH-IN interactive federated analysis system"),
  h1("The", img(src="dash-in.png", width="250px"), "interactive federated analysis system")
,


  # Sidebar with a slider input for number of bins
  sidebarLayout(
    sidebarPanel(
      selectInput("plotType",
                  "Plot type:",
```

```
                list("Histogram" = "hist",
                     "Contour Plot" = "contour",
                     "Heatmap" = "heatmap")
                ),

      selectInput("var_x",
                  "Variable",
                  get_study_variables()
      ),

      conditionalPanel(
        condition = "input.plotType != 'hist'",

        selectInput("var_y",
                    "Contrast variable",
                    get_study_variables()
        )
      ),

      div(style="font-size: .9em",

          hr(style="border-top-color: #aaa"),
          helpText("You may specify custom wording in the plot before exporting for publicati
on."),

          textInput("title", "title", ""),
          uiOutput("xlabel"),
          uiOutput("ylabel")
      )

    ),

   # Show a plot of the generated distribution
   mainPanel(
```

```
    plotOutput("distPlot")

  )
 )
))
```

server.R

```
library(shiny)

shinyServer(function(input, output) {

 output$distPlot <- renderPlot({

  if ( input$plotType == "hist") {

    h  <- ds.histogram(x = paste0("D$", input$var_x))
    plot(x = h,
        main = ifelse(input$title  != "", input$title, paste("Histogram of", input$var_x)),
        xlab = ifelse(input$xlabel != "", input$xlabel, input$var_x),
        ylab = ifelse(input$ylabel != "", input$ylabel, "Frequency"))

  } else if ( input$plotType == "contour") {

    # delete unclear labels and title
    par(col.main="white", col.lab="white")
    ds.contourPlot(x = paste0("D$", input$var_x),
                y = paste0("D$", input$var_y),
                show = "zoomed"
                )

    title(main = ifelse(input$title != "",
                    input$title,
                    paste("Correlation of", input$var_x, "and", input$var_y)),
```

```
                    col.main="black"
    )
    mtext(ifelse(input$xlabel != "", input$xlabel, input$var_x), side=1, line=3, col = "black")
    mtext(ifelse(input$ylabel != "", input$ylabel, input$var_y), side=2, line=3, col = "black")

  } else if ( input$plotType == "heatmap") {

    par(col.main="white", col.lab="white")
    ds.heatmapPlot(x = paste0("D$", input$var_x),
                   y = paste0("D$", input$var_y),
                   show = "zoomed"
    )

    title(main = ifelse(input$title != "",
                        input$title,
                        paste("Heatmap of", input$var_x, "and", input$var_y)),
          col.main="black"
    )
    mtext(ifelse(input$xlabel != "", input$xlabel, input$var_x), side=1, line=3, col = "black")
    mtext(ifelse(input$ylabel != "", input$ylabel, input$var_y), side=2, line=3, col = "black")

  }

}) # output$distPlot <- renderPlot({


output$xlabel <- renderUI({
  textInput("xlabel", paste0("x label (for variable ", input$var_x, ")") , "")
})


output$ylabel <- renderUI({
  if ( input$plotType == "hist" ) {
    textInput("ylabel", "y label (for frequency)", "")
  } else {
```

```
    textInput("ylabel", paste0("y label (for variable ", input$var_y, ")") , "")
  }

 })

})
```

global.R

```
library(opal)
library(dsBaseClient)
library(dsStatsClient)
library(dsGraphicsClient)
library(dsModellingClient)

##
## DATASHIELD commands

# load the login file
my_login<-read.table('../logins.txt', sep="", header=TRUE)
# log in to the remote servers
# assign=TRUE will have the remote opal server instruct the remote R
# instance to assign the dataframe into variable 'D'
opals <- datashield.login(logins=my_login, assign=TRUE, symbol = 'D')

# detect the list of variables in the study
get_study_variables <- function(symbol="D") {
 tryCatch({
    ds.colnames(x=symbol)[[1]]
  }, error = function(e) {
    print(e)
    return( list("No data was loaded! See error messages!") )
```

```
  }
 )
}
```

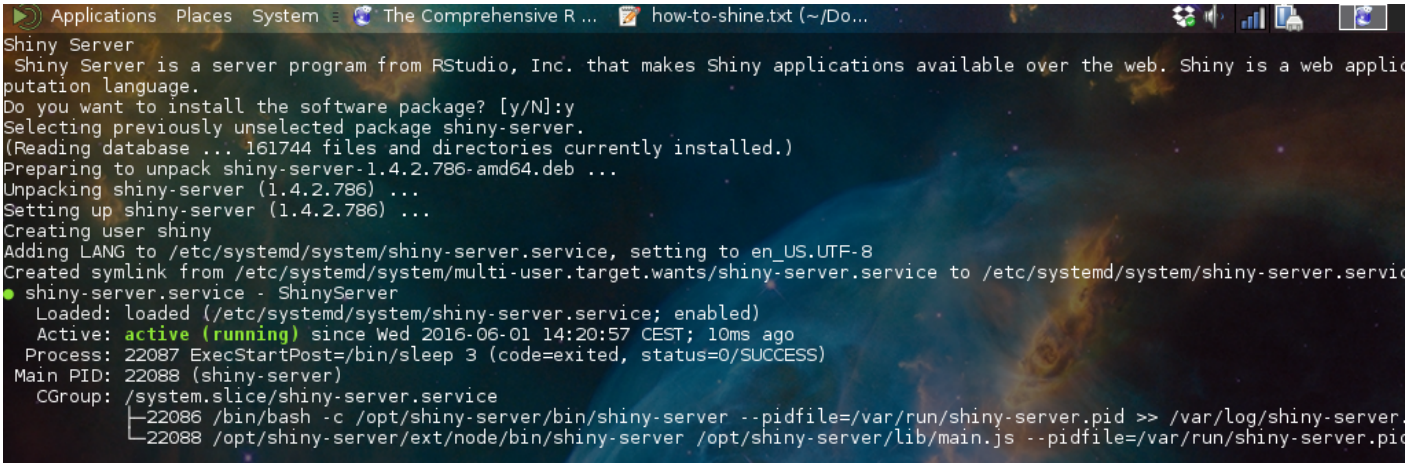
Server configuration and deployment of a multi-application server

DEBIAN

Install the shiny server

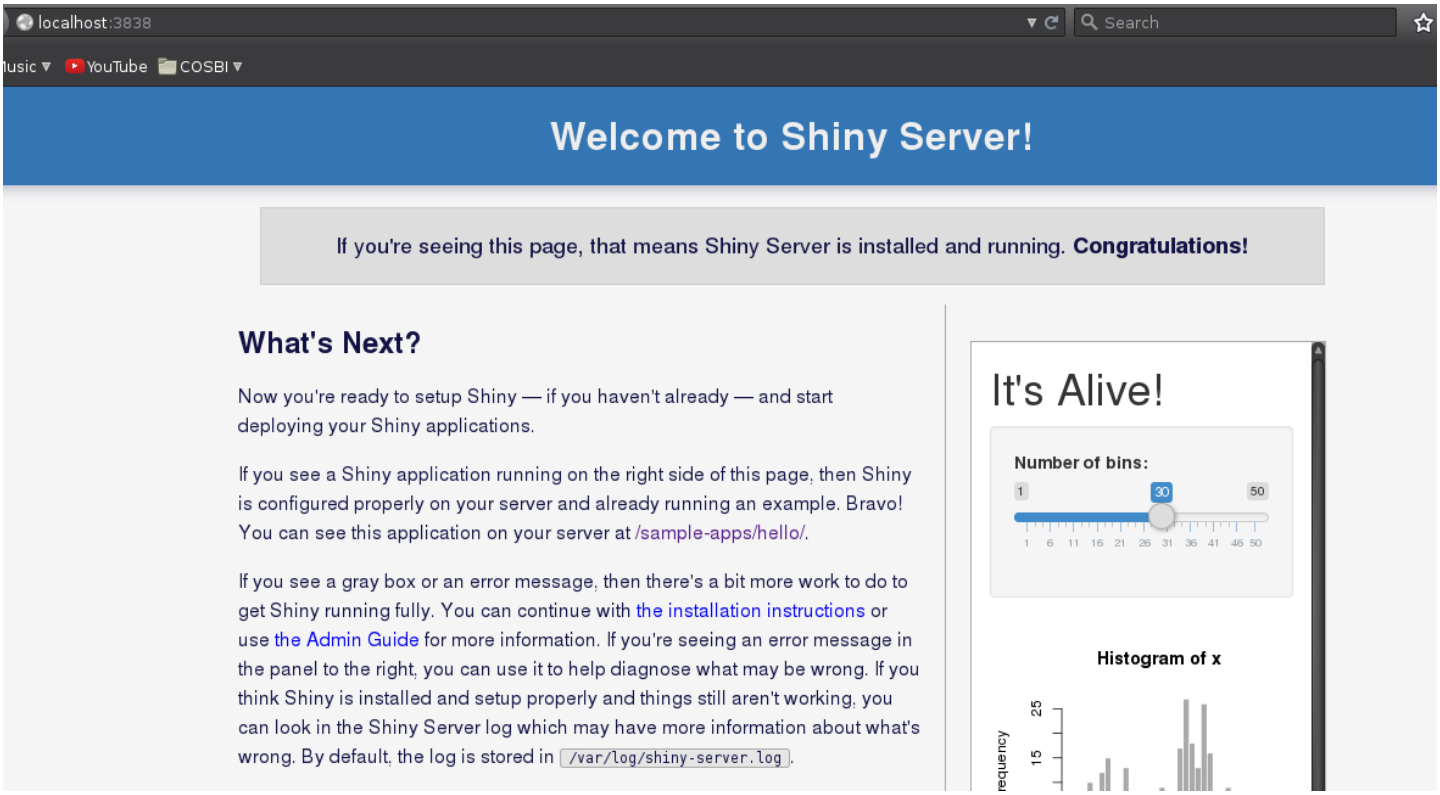
```
$ wget https://download3.rstudio.org/ubuntu-12.04/x86_64/shiny-server-1.4.2.786-amd64.deb
$ sudo gdebi shiny-server-1.4.2.786-amd64.deb
```

At this point the server should be automatically up running.



Test if it's running

with the default configuration test: *http://localhost:3838*



The configuration file is located at /etc/shiny-server/shiny-server.conf
The file is well commented, so it will be easy to understand what to edit in order to get the desired configuration.

To change the port, search and edit the line:

```
listen 3838;
```

To change the address:

```
location /put/here/your/address { ...
```

To reload the server with the new configuration:

```
$ sudo service shiny-server stop
$ sudo service shiny-server start
```

For the deployment of a multi-application server simply prepare different folders each containing its own ui.R, server.R (and optionally global.R) and the server will treat each such folder as a different application.

MAC

On Macs the shiny server needs to be complied form source. It all passes through homebrew. Install homebrew with the following command:

```
$ /usr/bin/ruby -e "$(curl –fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Using homebrew install the following software:

- python 2.6 or 2.7 (Really. 3.x will not work)
- cmake (>= 2.8.10)
- gcc
- g++
- git

typing commands as the following:

```
$ brew install python
```

Install a development version of R available from ATT: http://r.research.att.com/

Then install the shiny package in the system-wide library:

```
$ install.packages("shiny", repo="http://cran.rstudio.org", type="source")
```

Now proceed with the first steps – **stopping before the CMAKE step** – under "Installation" on the official page at
https://github.com/rstudio/shiny-server/wiki/Building-Shiny-Server-from-Source
The current *launcher.cc* source file must be edited to use the proc_pidpath() function on OSX instead of Linux proc (see this thread). Use this version from Nathan Weeks instead. After replacing the file, you can proceed with cmake and all subsequent installation steps.

See references:
https://github.com/rstudio/shiny-server/wiki/Building-Shiny-Server-from-Source
http://www.ducheneaut.info/installing-shiny-server-on-mac-os-x/
https://groups.google.com/forum/#!topic/shiny-discuss/WTXFtrEnR-k
https://github.com/nathanweeks/shiny-server/blob/d5240ef6d795dafc89c74a49d6f14d7fe0509541/src/launcher.cc