MZES SSDL - Shiny Apps: Development and Deployment

Konstantin Gavras

MZES, University of Mannheim

15 October 2019

Shiny

Shiny from RStudio

- Shiny is a package by RStudio to build interactive web pages. . .
  - without having any knowledge of web development (HTML/CSS/JavaScript)
- Shiny Apps interact with R
  - Allows for calculations, display of R objects, presentation of results . . .
- Examples: Democracy and MeTwo

Shiny App Components

1. Front end

- the web page actually shown to the user
- the HTML page written by Shiny
- includes layout, appearance, design features
- in Shiny terminology: `ui` (user interface)

2. Back end

- code running the app, including all functions, data import, etc.
- involves the logic of the app
- responsible for creating objects on the front end
- in Shiny terminology: `server`

Setting up a Shiny App

Shiny Apps can be set up in two different ways:

1. Single file App
   - `ui` and `server` are stored in one script
   - used when developing very simple Shiny Apps
   - name of the file has to be `app.R`!!!
2. Two file App
   - `ui` and `server` are stored in separate scripts
   - clear separation between front end and back end
   - highly preferable when developing more advanced Shiny Apps
   - names of the files have to be `ui.R` and `server.R`!!!

→ We are going to develop Shiny Apps using the Two File method

Developing Shiny Apps - Step by Step

Let's get started!

Workshop materials:
https://github.com/KostaGav/shiny-development-deployment

Features covered in the workshop:

Development:

1. Building a Shiny App from scratch
2. Building the plain UI
3. Getting output objects and control widgets into the UI
4. Implementing the server logic
5. Output/Input Reaction
6. Rendering objects
7. Reactivity

Deployment:

1. Deploy your app using `shinyapps.io`
2. Deploy on your own VM using `Shiny Server`

Building a Shiny App from scratch

```r
install.packages("shiny")
library(shiny)
runExample("01_hello")
#To show alternative Apps, please type runExample(NA)
#and choose another example
```

Building a Shiny App from scratch

- Create a new folder with two R scripts:

`ui.R`:

```
library(shiny)
ui <- fluidPage()
```

`server.R`:

```
server <- function(input, output){}
```

- Launch the Shiny App by pressing the ‘Run App’ button in the top right corner

Building the plain UI

Building the plain UI

- When building a Shiny App, one should have, in general, in mind how the app should 'look' like
- Thus, we build the UI first
- In simple Shiny App, the whole UI fits in the `fluidpage`
  - every new object is passed comma-separated
  - text can be passsed to the UI by entering strings
- In order to format text, Shiny uses HTML wrappers:
  - these wrappers are functions taking one object as argument (+ further style options)
  - `h1()`: Top-level header
  - `h2()`: secondary header
  - `strong()`: make text bold
  - `em()`: make text italicized
  - `br()`: add line break
- We can add an official header using `titlePanel()`

Q: Can you see any particular differences between using `h1()` and the `titlePanel()` when using them as title?

Building the plain UI

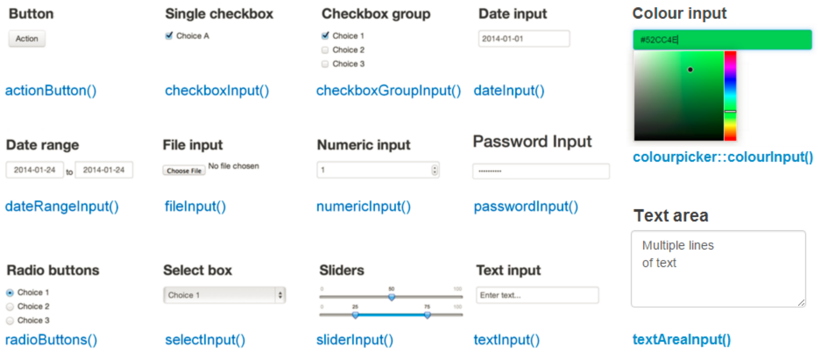
- Until now, we only have a plain white page
- We need a proper layout to make it appear nicer:
  - `sidebarlayout` is the simplest layout format
  - Input and control widgets on the left side, results and plots on the right hand side

```r
ui <- fluidPage(
  titlePanel("Title of my Shiny App"),
  sidebarLayout(
    sidebarPanel("My input goes here"),
    mainPanel("The results go here")
  )
)
```

Adding Input and Control Widgets

- In order to interact with Shiny Apps, we need control widgets
- User can specify inputs, enter text or select specific dates to create a certain results
- all input function have two arguments: `inputId` and `label`
  - `inputId`: name Shiny uses to refer to this input, when retrieving values for the back end
  - has to be unique! (WARNING: if you provide two Ids with the same name, there won't be an error message!)
  - `label`: Text displaying the label of the control widget

Adding Input and Control Widgets



Adding Input and Control Widgets

- control widgets go in the `sidebarPanel`
- always choose control widgets depending on the design of your app!
- Most common:
  - `radioButtons()`
  - `selectInput()`
  - `sliderInput()`

Adding Input and Control Widgets - Radio Buttons

- We specify the possible values, range and appearance in the control widget

```r
radioButtons(
  "buttons",
  "Choose a mode of transmission",
  choices = c("automatic" = 0, "manual" = 1),
  selected = 0
)
```

Adding Input and Control Widgets - Select Input

- We specify the possible values, range and appearance in the control widget

```r
selectInput(
  "selector",
  "Select the number of cylinders",
  choices = c(4, 6, 8)
)
```

Adding Input and Control Widgets - Slider Input

- We specify the possible values, range and appearance in the control widget

```
sliderInput(
  "slider",
  "Pick a range of gross horsepower",
  min = 0,
  max = 400,
  value = c(10,300),
  pre = "hp"
)
```

Preparing the output

- We need elements to specify where the outputs should be displayed
- These outputs might be plots, tables, text, images, maps, or ...
- In the UI, we only build the placeholder, which will be filled using the server logic
- every output function has an `outputId` argument to identify the output created in `server.R`
- Plot output:

```r
plotOutput("greatPlot")
```

- other output forms: `tableOutput()`, `textOutput()`, etc.
- output elements should always be added to the `mainPanel()`

Exercise I - Building your own UI

- Using the mtcars data set, we will now start creating our own Shiny Apps

1. Make yourself familiar with the data set, if you don't know it yet.

```
library(tidyverse)
glimpse(mtcars)
?mtcars
```

2. We want to create a classic data presentation app. Think of an appropriate UI for the data presentation. If you like, you can draw a sketch.
3. Think of useful control widgets to control data presentation.
4. Create the two files needed for Shiny Apps, add the relevant code to initiate the app and set up an UI with a sidebar.
5. Add the control widgets and specify the conditions, you want the users to manipulate
6. Add a placeholder for the output you want to create
7. Run the app regularly to see how you proceed

Implementing the server logic

Implementing the server logic

We now switch from front end to back end!

- Server logic in Shiny Apps builds upon an `input` and an `output` argument
  - `input`: Contains the values of given input by the users
  - `output`: Plots and tables created, based on the values from the input
- Output objects can be created without any input specification, but always need to be connected with the UI by the `outputId`
- Standard procedure:
  1. Save the output object into the `output` list
  2. Build the object using a `render` function. For every object type, there is a unique `render` function, e.g. `renderPlot()`
  3. (Access input values using the `input` list)

Implementing the server logic

- We have to make sure, that two conditions match: Ids and output/render functions

```r
output$greatPlot <- renderPlot({
  plot(rnorm(1000))
})
```

- Since we named our plotOutput in the UI `greatPlot`, we have to assign it to this Id
- Also, we are using the `plotOutput` function in the UI. Thus, we need the `renderPlot` function to match it

Output/Input Reaction

- Presenting static plots is kinda boring. We want users to fiddle around with the data to gain insights from the data
- Our plots need a connection to the input and the control widgets

```r
output$greatPlot <- renderPlot({
  plot(rnorm(input$slider[1]))
})
```

- The logic behind the `input` list is the same as for the `output` list
  - based to the Ids provided in the UI, we access the respective object (control widget)
  - given the structure of the input, we specify it to fit the output object

Rendering Plots

- We can add further code to the `renderPlot` function in order to increase interactivity of plots

```
library(ggplot2)

output$greatPlot <- renderPlot({
  ggplot(mtcars, aes(mpg)) +
    geom_histogram()
})
```

Rendering Plots

- A very simple way of increasing interactivity is filtering

```r
library(ggplot2)

output$greatPlot <- renderPlot({

  filteredData <-
    mtcars %>%
    filter(hp >= input$slider[1],
           hp <= input$slider[2],
           cyl == input$selector,
           am == input$buttons)

  ggplot(filteredData, aes(mpg)) +
    geom_histogram()
})
```

- Now the plot is interactive, but the whole code will get re-executed every time the user re-specifies anything $\rightarrow$ Can become very memory intensive and repetitive when building several plots!

Rendering Plots - Make it more Interactive!

- One way of increasing the user experience with your app are `plotly` plots

```r
library(plotly)
# ui.R
## Replace plotOutput() with plotlyOutput()
# server.R
  output$greatPlot <- renderPlotly({

    filteredData <-
      mtcars %>%
      filter(hp >= input$slider[1],
             hp <= input$slider[2],
             cyl == input$selector,
             am == input$buttons)

    plotHist <- ggplot(filteredData, aes(mpg)) +
      geom_histogram()

    ggplotly(plotHist)
  })
```

Reactivity

- We can use reactive variables to extract the filtering from the rendering functions

```r
library(ggplot2)

  filtered <- reactive({
    mtcars %>%
      filter(hp >= input$slider[1],
             hp <= input$slider[2],
             cyl == input$selector,
             am == input$buttons)
  })

output$greatPlot <- renderPlotly({
  plotHist <- ggplot(filtered(), aes(mpg)) +
    geom_histogram()
  ggplotly(plotHist)
})
```

- Reactive objects can now be used for different objects, which change simultaneously with different input

Exercise II

1. Build the server logic for your app. Reconsider your ideas for the app and try to implement them in `server.R`
2. Create a reactive variable to add flexibility to your app
3. Find a solution to deactivate or reset all filter options with one additional control widget
4. Add a second plot to your app with different functionalities
5. Replace your plots with `plotly` plots

Deployment

Deployment on `shinyapps.io`

- Until now, we only ran our apps locally on our machine
- In order to present your apps to your friends and family, we need to deploy it in the www
- However, we have to take care that our app is protected by a firewall and we have a stable URL.
- For simple apps, we can just use `shinyapps.io`
- They will take care of the homepage maintenance

Deployment on `shinyapps.io`

Deployment using `shinyapps.io`:

1. Create free account on `shinyapps.io` (use your university email address)
2. Go back to RStudio, press the deploy button in the top right corner
3. Re-enter your credentials, select the correct files and let the magic happen :)

Exercise III

1. Create an account on `shinyapps.io`
2. Try to deploy your app using the deploy button
3. Share your URL with friends and family - You're a web developer now!

Deployment using Shiny Server

- For certain apps you might not want to deploy on `shinyapps.io` because
  - their size would require you to use a costly plan
  - you want full control over the app and host it by yourself
  - you love playing around with Unix code...
- A free alternative is Shiny Server, allowing you to host your own app
- However, you need an own server to host the app (e.g. Digital Ocean, Amazon Web Services, ...) $\rightarrow$ Still costly...
- Or use the web services provided by the universities of Baden-Wurttemberg: **bwCloud** $\rightarrow$ completely for free (for members of the U of Mannheim)!!!

Deployment using Shiny Server

When deploying by yourself with Shiny Server, your life will be a bit more complicated:

1. Register for the bwCloud
2. Log into the bwCloud Dashboard
3. Create an SSH-key pair to connect to your server (find a short intro here)
4. Install PuTTY (Windows) or start a remote connection in the Shell (MAC) as well as Filezilla
5. Set up the SSH-client to access remote connections
6. Build up a virtual machine (the server) using your bwCloud dashboard and connect to your VM using SSH with PuTTY

- We have not even downloaded Shiny Server ;)

Deployment using Shiny Server

When having set up the server, we need to enter our Unix system and install R, all relevant packages and Shiny Server using beautiful Unix code

1. Install R on your machine

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

- If you are lucky, the correct version is being installed... most probably you are unlucky, then see here for Ubuntu

2. Install dependencies to install R packages...

```
sudo apt-get -y install libcurl4-gnutls-dev
sudo apt-get -y install libxml2-dev libssl-dev
```

3. Install R packages within R, including `shiny` (easy!)
4. Install Shiny Server (check for latest version!)

```
wget https://download3.rstudio.org/ubuntu-12.04/x86_64/shiny-ser
sudo gdebi shiny-server-1.5.6.875-amd64.deb
```

Deployment using Shiny Server

5. Test whether Shiny Server runs correctly:
   `http://134.155.108.111:3838/` (replace with your IP-address)
6. Use Filezilla to access your server and upload your app files to the `/srv/shiny-server/` folder. It should run when typing in the correct URL associated with your app, for example
   `http://134.155.108.111:3838/SupForDemocracy/Democratic_Decon`
7. Most probably it won't work immediately and you need to troubleshoot... Enjoy! :)

Thank you!