Modules

Mine Çetinkaya-Rundel

@minebocek
mine-cetinkaya-rundel
mine@rstudio.com

What is a module?

- A module is a self-contained, composable component of a Shiny app
  - self-contained like a function
  - can be combined to make an app
- Have their own UI and server (in addition to the app UI and server)
- Useful for reusability
  - rather than copy and paste code, you can use modules to help manage the pieces that will be repeated throughout a single app or across multiple apps
  - can be bundled into packages
- Essential for managing code complexity in larger apps

Limitations to just functionalizing

- It's possible to
  - write UI-generating functions and call them from your app's UI, and
  - write functions to be used in the server that define outputs and create reactive expressions.
- However names (ids) of the input and output components are global: all parts of your server function can see all parts of your UI.
- Modules give you the ability to create controls that can only be seen from within the module via **namespaces**:"spaces" of "names" that are isolated from the rest of the app.

- **04-modules > 01-histogram.R**

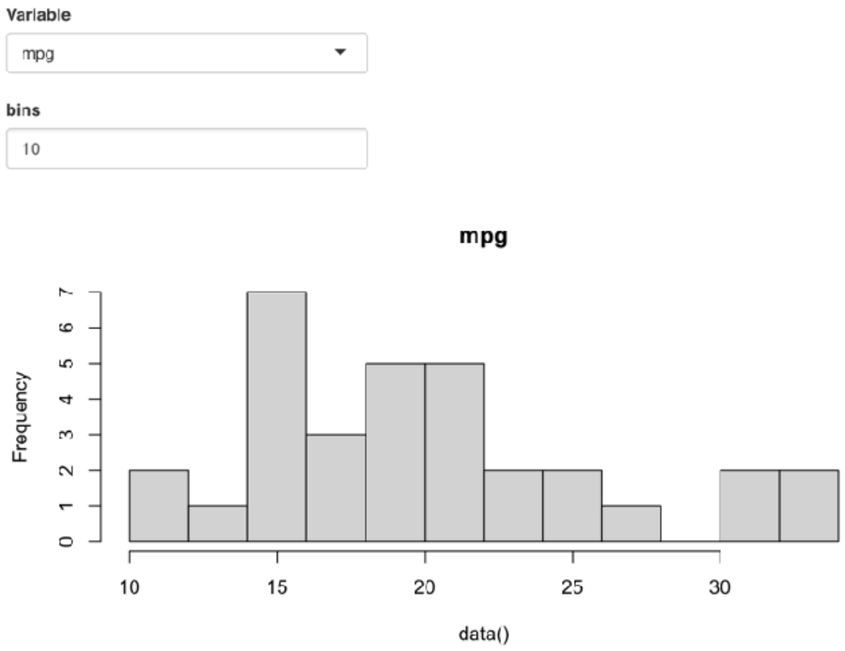
Demo

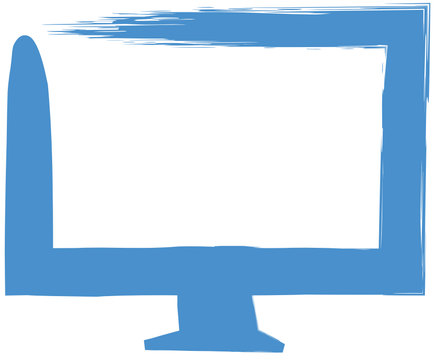
```
library(shiny)

ui ← fluidPage(
  selectInput("var", "Variable", names(mtcars)),
  numericInput("bins", "bins", 10, min = 1),
  plotOutput("hist")
)

server ← function(input, output, session) {
  data ← reactive(mtcars[[input$var]])
  output$hist ← renderPlot({
    hist(data(), breaks = input$bins, main = input$var)
  }, res = 96)
}

shinyApp(ui, server)
```





Module UI

```r
ui <- fluidPage(
  selectInput("var", "Variable", names(mtcars)),
  numericInput("bins", "bins", 10, min = 1),
  plotOutput("hist")
)
```

```r
histogramUI <- function(id) {
  tagList(
    selectInput(NS(id, "var"), "Variable", choices = names(mtcars)),
    numericInput(NS(id, "bins"), "bins", value = 10, min = 1),
    plotOutput(NS(id, "hist"))
  )
}
```

```r
histogramUI <- function(id) {
  tagList(
    selectInput(NS(id, "var"), "Variable", choices = names(mtcars)),
    numericInput(NS(id, "bins"), "bins", value = 10, min = 1),
    plotOutput(NS(id, "hist"))
  )
}
```

Module UI

- Put the UI code inside a function that has an `id` argument
- Wrap each existing ID in a call to `NS()`, so that (e.g.) `"var"` turns into `NS(id, “var")`
- Aside: `tagList()` allows you to bundle together multiple components without actually implying how they’ll be laid out

Module server

```
server <- function(input, output, session) {
  data <- reactive(mtcars[[input$var]])
  output$hist <- renderPlot({
    hist(data(), breaks = input$bins, main = input$var)
  }, res = 96)
}
```

```
histogramServer <- function(id) {
  moduleServer(id, function(input, output, session) {
    data <- reactive(mtcars[[input$var]])
    output$hist <- renderPlot({
      hist(data(), breaks = input$bins, main = input$var)
    }, res = 96)
  })
}
```

```r
histogramServer <- function(id) {
  moduleServer(id, function(input, output, session) {
    data <- reactive(mtcars[[input$var]])
    output$hist <- renderPlot({
      hist(data(), breaks = input$bins, main = input$var)
    }, res = 96)
  })
}
```

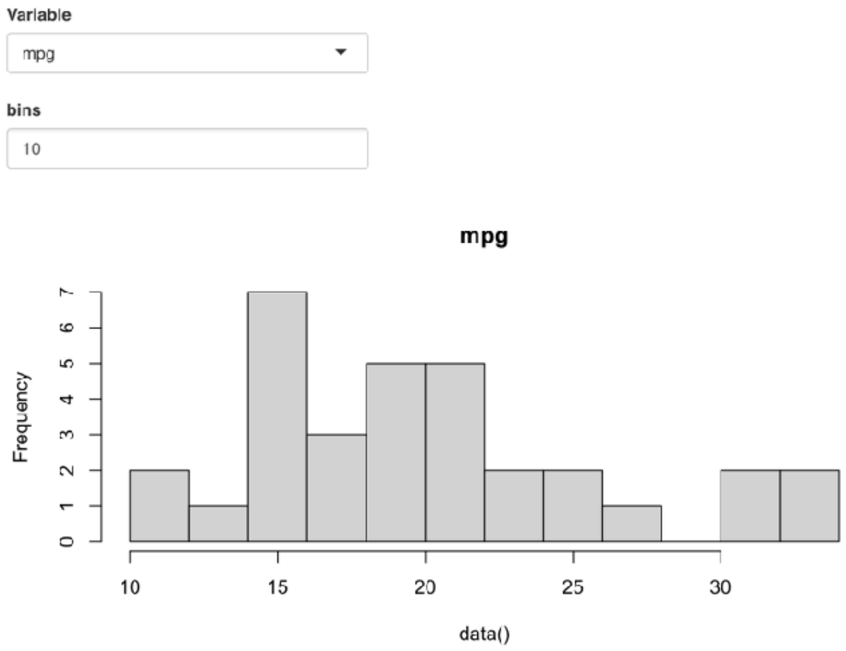
Module server

- Module server gets wrapped inside another function which must have an `id` argument
- This function calls `moduleServer()` with the `id`, and a function that looks like a regular server function
- `input$var` and `input$bins` refer to the inputs with names `NS(id, "var")` and `NS(id, "bins")`

Demo

- **04-modules > 02-histogram.R**

```
histogramApp ← function() {
  ui ← fluidPage(
    histogramUI("hist1")
  )
  server ← function(input, output, session) {
    histogramServer("hist1")
  }
  shinyApp(ui, server)
}
```

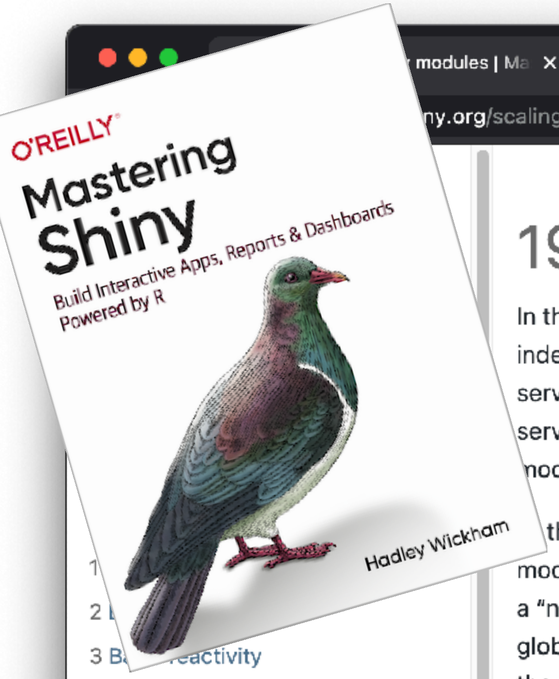


Naming conventions

- `R/histogram-modules.R` holds all the code for the module.
- `histogramUI()` is the module UI.
- `histogramServer()` is the module server.
- `histogramApp()` creates a complete app for interactive experimentation and more formal testing

Namespacing

- In the module UI, the namespacing is explicit: you have to call `NS(id, "name")` every time you create an input or output.
- In the module server, the namespacing is implicit. You only need to use `id` in the call to `moduleServer()` and then Shiny automatically namespaces input and output so that in your module code `input$name` means the input with name `NS(id, "name")`.



modules | Ma

ny.org/scaling-modules.html

19 Shiny modules

In the last chapter we used functions to decompose parts of your Shiny app into independent pieces. Functions work well for code that is either completely on the server side or completely on the client side. For code that spans both, i.e. whether the server code relies on specific structure in the UI, you'll need a new technique: modules.

At the simplest level, a module is a pair of UI and server functions. The magic of modules comes because these functions are constructed in a special way that creates a "namespace". So far, when writing an app, the names (`id`s) of the controls are global: all parts of your server function can see all parts of your UI. Modules give you the ability to create controls that can only be seen from within the module. This is called a **namespace** because it creates "spaces" of "names" that are isolated from the rest of the app.

Shiny modules have two big advantages. Firstly, namespacing makes it easier to understand how your app works because you can write, analyse, and test individual components in isolation. Secondly, because modules are functions they help you reuse code; anything you can do with a function, you can do with a module.

```
library(shiny)
```

19.1 Motivation

Before we dive into the details of creating modules, it's useful to get a sense for how they change the "shape" of your app. I'm going to borrow an example from Eric Nantz, who talked about modules at rstudio::conf(2019): https://youtu.be/yIlLVo2VL50. Eric

On this page

Learn more

Modules

Mine Çetinkaya-Rundel

@minebocek
mine-cetinkaya-rundel
mine@rstudio.com