mwshiny: Connecting Shiny Across Multiple Windows

By Hannah De los Santos, John Erickson, Kristin P. Bennett
Rensselaer Polytechnic Institute
What is Shiny?

- Shiny lets users develop interactive web applications (apps) with only R. The code is structured as `app.R`.

  - `ui` is the user interface, which defines how the app will be presented to the user.
  - `server(input, output, session)` handles the logic and data processing.
  - `shinyApp(ui, server)` is the function that combines `ui` and `server` to create the app.

Examples from RStudio Shiny Gallery: https://shiny.rstudio.com/gallery/
A Problem: Shiny apps have only one window!

- Shiny apps only span one window, and space is limited

- You can alleviate this problem by adding separate tabs, but it’s difficult to compare outputs

Examples from RStudio Shiny Gallery: https://shiny rstudio.com/gallery/
We Live In a Multi-Monitor World

Multiple Monitors at a Workstation

Controller Driving External Monitor
mwshiny: Multi-Window Shiny

- mwshiny extends Shiny across multiple separate windows
- Uses Shiny’s syntax and conventions, so not much additionally to learn

mwshiny does this by breaking app development into an easy workflow:

User Interface Development → Server Computation → Server Output

On CRAN as: mwshiny
Mwshiny: User Interface Development

- Specify each window with user interface arguments:
  - Window titles (`win_titles`): a vector of strings
  - User interfaces (`ui_win`): a list of traditional Shiny UIs corresponding to each title

**Example:**

```r
win_titles <- c("Controller","Scatter")

ui_win <- list()
ui_win[[1]] <- fluidPage(...) # controller
ui_win[[2]] <- fluidPage(...) # scatter
```
mwshiny: Server Calculations

- Observe events and create variables for output to be rendered:
  - Server calculations (serv_calc): a list of functions of the form function(calc, sess), where:
    - `calc`: reactive variable that contains Shiny input variables, as well as a place to put calculated variables
    - `sess`: traditional Shiny server session variable

**Example:**

```r
serv_calc <- list()
serv_calc[[1]] <- function(calc, sess){
  observeEvent(calc$go, {
    calc[['sub.df']] <- data.frame(calc$go)
  })
}
```
mwshiny: Server Output

- Render output based on input and calculated variables:
  - Server output (serv_out): a named list of functions of the form function(calc, sess), which returns a render() result, and is named corresponding to the output ID

Example:

```r
serv_out <- list()

serv_out[['iris_scatter']] <- function(calc, sess){
  renderPlot({
    if (!is.null(calc$sub.df)){
      ggplot(calc$sub.df)...
    }
  })
}
```

Example: Iris Dataset Explorer: Scatter
mwshiny: Breaking Down the Workflow

- Separating server into computation and output clarifies workflow
- Enhanced by list and function structure of mwsApp() variables
Examining mwshiny’s Workflow Through Three Case Studies

**Case 1:**
Multiple Monitors at a Workstation

**Case 2:**
Controller Driving External Monitor

**Case 3:**
Alternative Visualization Structures

Download scripts on Github: https://github.com/delosh653/mwshiny-examples
Case 1: Examining Population Dynamics Using Two Monitors

- Use 2010 US Census to examine population statistics at state and county levels
- Begin by defining our user interfaces for each window:

```r
win_titles <- c("Controller", "Map")
ui_list <- list()

ui_list["Controller"] <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      selectInput("stat", "Which statistic would you like to visualize?", choices = stat_choi),
      actionButton("go", "Visualize!")
    ),
    mainPanel(
      tabsetPanel(
        tabPanel("Aggregate Dynamics",
          plotOutput("overall_dens")),
        ...
      )
    )
  )
)

ui_list["Map"] <- fluidPage(
  plotlyOutput("map", height = "1000px"),
  ...
)
```

Download scripts on Github: 
https://github.com/delosh653/mwshiny-examples
Case 1: Server Calculations

- We then calculate variables based on which states or counties we’ve chosen that are required for visualizations:

```
serv_calc <- list()
serv_calc[[1]] <- function(calc, sess){
  observeEvent(calc$go, {
    ...
    calc[["over_df"]]<- ...
    calc[["state_df"]]<- ...
    calc[["merge_pop"]]<- ...
    ...
  })
}
```

- These will be used in rendering our plots

Download scripts on Github: https://github.com/delosh653/mwshiny-examples
Case 1: Server Output

In this case, we have our three outputs based on these calculations:

```r
serv_out <- list()
```

- `serv_out["overall_dens"]`:
  ```r
  function(calc, sess){
    renderPlot({
      ggplot(calc$over_df,...)+
      ... 
    })
  }
  ```

- `serv_out["state_dens"]`:
  ```r
  function(calc, sess){
    renderPlot({
      ggplot(calc$state_df,...)+
      ... 
    })
  }
  ```

- `serv_out["map"]`:
  ```r
  function(calc, sess){
    renderPlotly({
      calc$merge_pop %>%
      group_by(group) %>%
      plot_mapbox(...
    })
  }
  ```

Download scripts on Github:
https://github.com/delosh653/mwshiny-examples
Case 1: Result

- Putting all this together, we end up with our multi-monitor system:

Download scripts on Github: https://github.com/delosh653/mwshiny-examples
Case 2: Using an External Controller to Drive Cultural Awareness

- We next have a controller (phone, tablet) driving an external monitor in an art museum, so patrons can learn more about their favorite artists

- Begin by defining UI:

```
win_titles <- c("Controller", "Art_Monitor")
ui_list <- list()

ui_list[["Controller"]]
  fluidPage(
    sidebarLayout(
      sidebarPanel(
        selectInput("art_wait", ...), ...
      ))
  )

ui_list[["Art_Monitor"]]
  fluidPage(
    htmlOutput("info") ...
    tabPanel("Artist", ...
    htmlOutput("artist")
  )
```

Download scripts on Github: https://github.com/delosh653/mwshiny-examples
Case 2: Server Computation and Server Output

Server Computation

```r
serv_calc <- list()
serv_calc[[1]] <- function(calc, sess){
  observeEvent(calc$go, {
    calc["select_tab"] <- calc$which_tab
    calc["art_person"] <- calc$art_wait
    calc["art_born"] <- artist_info...
    updateTabsetPanel(sess, "art",
                     selected = calc$select_tab)
  })
}
```

Server Outputs

```r
serv_out <- list()
serv_out["info"] <-
  function(calc, sess){...
 serv_out["artist"] <- ...
 serv_out["painting_1"] <- ...
 serv_out["painting_2"] <- ...
 serv_out["painting_3"] <- ...
 serv_out["map"] <- ...
```

Information from Wikipedia

Download scripts on Github:
https://github.com/delosh653/mwshiny-examples
Case 2: Result

- With all these pieces, we end up with our ideal monitor control situation:
Case 3: An Immersive Shiny Application with the Rensselaer Campfire

- The Rensselaer Campfire’s 3D structure enhances knowledge discovery by reducing cognitive load

- Users surround and interact with the Campfire to explore their data

Download scripts on Github: https://github.com/delosh653/mwshiny-examples
Case 3: Using the Campfire to Explore Circadian Rhythm Data

- In this case study, we use the Campfire to explore the function of circadian rhythms:
  - ~24-hour cycles reinforced by external cues (light)
  - In mouse data, they are amplitude-changing cosine waves
  - Interrupting rhythms leads to increased health risk of many diseases (cancer, diabetes, etc.)

- Use the Campfire to explore different functions of circadian genes and how they interact with each other

ENCORE application: https://github.com/delosh653/ENCORE

Download scripts on Github: https://github.com/delosh653/mwshiny-examples
Case 3: User Interface Development

- As always, we begin with what we want our user interfaces to look like:

```r
win_titles <- c("Controller", "Wall", "Floor")
ui_list <- list()

ui_win[['Controller']] <- fluidPage(
  h2("Explore the function of circadian rhythms using the campfire!"),
  sidebarLayout(
    sidebarPanel(
      selectInput("new_path", "Which GO Term would you like to examine?",
        choices = c("metabolic process" = "GO:0008152", ...

        ),
        mainPanel()
      )
  )
)

ui_win[['Wall']] <- div(
  d3Output("wall", height = "663px")
)

ui_win[['Floor']] <- div(
  d3Output("floor", height = "895px")
)
```

Download scripts on Github:
https://github.com/delosh653/mwshiny-examples
Case 3: Server Calculations and Output

**Server Computation**

```r
serv_calc <- list()
serv_calc[[1]] <- function(serverValues, sess){
  observeEvent(serverValues$new_path, {...})
}
serv_calc[[2]] <- function(serverValues, sess){
  observeEvent(serverValues$dark, {...})
}
serv_calc[[3]] <- function(serverValues, sess){
  observeEvent(serverValues$undark, {...})
}
```

**Server Outputs**

```r
serv_out <- list()
serv_out[['wall']] <- function(calc, sess){...}
serv_out[['floor']] <- ...
```

Data from Hughes, et al. (2009)

Download scripts on Github: https://github.com/delosh653/mwshiny-examples
Case 3: Result

- Once we’ve gone through our workflow, we can see our example for the function “metabolic process” in the Campfire:

Full Campfire

Hovering to see connections for Rps11

Data from Hughes, et al. (2009)
Summary: Multi-Window Shiny (mwshiny)

- mwshiny extends standard Shiny to have multiple windows
- mwshiny requires a breakdown of Shiny workflow further to increase clarity
- Presented mwshiny’s efficacy in three scenarios:
  1. Multi-monitor systems (U.S. Population Dynamics)
  2. Controller-driven monitor (Art Cultural Exploration)
  3. Alternative visualizations in the Campfire (Circadian Rhythm Function)
Acknowledgements

- Thank you!
  - Hurley lab: Jennifer Hurley, Emily Collins, Meaghan Jankowski
  - IDEA lab: Kristin Bennett, John Erickson
- Support:
  - National Science Foundation
  - Rensselaer Polytechnic Institute
  - National Institutes of Health