Interactive Web Applications with Shiny

Sean Hellingman ©

Data Visualization and Manipulation through Scripting (ADSC1010)

shellingman@tru.ca

Fall 2024

THOMPSON RIVERS UNIVERSITY

Topics

2. Introduction
3. The Shiny Framework
4. Core Concepts
5. Application Structure
6. Designing User Interfaces
7. Static Content
8. Dynamic Inputs
9. Dynamic Outputs
10. UI Layouts
11. Developing Application Servers
12. Publishing Applications
13. Exercises and References

Introduction

- Adding interactivity to our reports can add an additional way to communicate your data.
- The **Shiny** framework will allow us to build interactive applications in R.
- *Shiny allows you to create dynamic systems in which users can choose what information they want to see, and how they want to see it.*
- Provides a structure for communicating between a user interface (web browser) and an R session.
- Users can interactively *change* the code that is run and the data that are output.

Shiny Framework I

- **Shiny is a web application framework for R.**
- We have covered generating static HTML files with R Markdown.
- The framework provides the *code* for producing and enabling dynamic web pages where the user may check boxes, click buttons, or input text to change how and what data is presented.
- The developer (you) provide variables or functions that the provided code will use to create the interactive page.

Shiny Framework II

- Our code needs to perform two tasks:
  1. Process and analyse information.
  2. Present that information to the user to see.

- The user **input** needs to be used to re-process the information, and then re-present the **output** results.

- Shiny is rendering a user interface for a web browser, so it actually generates a website.

User Interface (UI)

- The **User Interface (UI)** defines how the application is displayed in the browser.
- Provides a web page that renders R content such as text or graphics (similar to R markdown).
- Supports interactivity through *control widgets*, which are interactive controls for the application (buttons & sliders).
- Can specify a **layout** for the components allowing the developer to organise content in side-by-side panels, or across multiple tabs.

Server

- The **server** defines and processes the data that will be displayed by the UI.
- Generally, the server is a program running (usually remotely) on a computer that receives and provides content based on request.
- The interactive R session that the user will use to *run* the data processing functions that the UI displays.
- These data processing functions are **reactive**: They automatically change (react) whenever the input changes. (*Interactive outputs*)

Control Widget

- A **control widget** is an element in the UI that allows the user to provide input to the server.
- Examples: text input box, drop-down menu, or slider.
- Store input values, which are automatically updated as the user interacts with the widget.
- Updated values are sent from the UI to the server and the new content is generated and displayed.

Reactive Output

- A **reactive output** is an element in the UI that displays dynamic content produced by the server.
- Examples: a chart that dynamically updates when the user selects different data to display, or a table that responds to a search query.
- Automatically updates whenever the server sends it a new value to display.

Render Function

- A **render function** is in the server and produces output that can be understood and displayed by the UI's reactive outputs.
- A render function will automatically *re-execute* whenever a related control widget changes.
- Produces an updated value that will be read and displayed by a reactive output.

Reactivity

- Shiny apps are designed around **reactivity**.
- Updating some components in the UI (control widgets) will cause other elements (render functions) to *react* and and automatically re-execute.
- This is similar to how parameterized Excel spread sheets work.

Initialization

- A shiny application is written in a script file named **app.R**
  - The file should be saved in the root directory of a project (root directory of a `git` repository).
- In R Studio: File $\Rightarrow$ New File $\Rightarrow$ Shiny web app...
- In a pop up window we can give the application a name and select a directory to save the application in.
  - Application type: Single File (app.R)
- We will need the *shiny* R package.

Example 1

- In R, create a new shiny application.
- Save your application in your own TRU folder.
- Take some time to look at the example code that is automatically generated.
- Click the *Run App* tab in the top right corner of the app.R file and play around with the application.

Components

- Shiny applications are separated into two components:
  1. User Interface
  2. The server
- These are coded separately and then combined using the `shinyApp()` function.

1. User Interface (UI)

- The UI defines how the application is displayed in the browser.
- The UI for a Shiny application is defined as a **value**.
- It is almost always a value returned from calling one of Shiny's **layout functions**.

2. The Server

- The server defines and processes the data that will be displayed by the UI.
- The server is defined as a **function**.
- This function needs to take in two lists as arguments called `input` and `output`.
  - The `input` list is received from the UI and is used to create content (calculate information/make graphics).
  - The content is then saved in the `output` list so that it can be sent back to the UI to be rendered in the browser.
- The server uses **render functions** to automatically assign the values to `output` so the content will automatically be recalculated.

Example 2

- Open the app.R file included in the moodle folder (Example 2).
- Work through the code and comments to see if things are starting to make sense.
- Run the application.

Structure Comments

- The responsibilities of the application are split between the UI and the server.
- Enabling this **separation of concerns** is a fundamental principle when designing computer programs.
- It allows for developers (you) to isolate problems and better scale your applications.

Designing User Interfaces

UI Design

- We want to create interfaces that prioritize important information in a clear and organised way.
- We can use structural elements within the Shiny framework to construct such pages.
- When we define a UI, we are defining how the app will be displayed in the browser.
- We can create a UI by calling a layout function such as `fluidPage()`

Layout Function

- A layout function can take as many content elements as needed.
- Each element is an additional argument usually placed on separate lines.
- Example:

```
ui <- fluidPage(element1, element2, element3)
```

- Example 2 has three content elements: `h2()`, `textInput()`, and `textOutput()`.
- *There are many types of content elements that can be passed to a layout function.*
- *You can even start with an empty **server** function to test your UI.*

Static Content I

- **Static Content** is the simplest element that a UI can contain.
- These elements will not change as the user interacts with the interface.
- *Generally used to add further explanatory information about what the user is looking at.*
- **Content elements are created by calling specific functions that create them.**

Some Static Content

Static Function	Markdown Equivalent	Description
`h1("Heading 1")`	# Heading 1	First-level heading
`h2("Heading 2")`	## Heading 2	Second-level heading
`p("text")`	text	Paragraph of plain text
`em("text")`	*text*	*Italic* text
`strong("text")`	**text**	**Bold** text
`a("text", href = "url")`	[text](url)	Hyperlink (anchor)
`img("description", src = "path")`	![description](path)	An image

- **Note: Save image in a folder called www**

Static Content II

- It is common to include a number of static elements to describe your application.
- Formatted content may even be nested (other content elements within one).
  - Example: `h1("My", em("Awesome"), "App")`
- Almost all Shiny apps have a `titlePanel()` content element.
  - Provides a second-level heading and specifies the title shown in the tab of a web browser.
- If you are familiar with HTML syntax you can use the `HTML()` to pass a string of the HTML you want to include.

Example 3

- Open the app.R file included in the moodle folder (Example 2).
- Take some time to add some more **static content** to the application.
- Make sure to include a ',' after each element.
- Run the application.

Dynamic Inputs

- Users interact with content elements called **control widgets**.
- These elements allow the users to provide input to the server.
- Each widget stores the value the user has input and passes this value as a variable to the server to change the output(s).
- There are many different widgets available in the Shiny package.

textInput()

- Creates a box in which the user can enter text. (Example app)
- Usage:

```
textInput(inputID = "Slot used to access variable",
  label = "Display label for the control",
  placeholder = "Character string example")
```

- textInput()

`sliderInput()`

- Creates a slider in which the user can drag to choose a value (or range of values).

- Usage:

```
sliderInput(inputID = "Slot used to access variable",
   label = "Display label for the control",
   min = "The minimum value (inclusive) that can be
selected.",
   max = "The maximum value (inclusive) that can be
selected.",
   value = inital value of the slider)
```

  - Note: *if you pass a vector of two elements to the* `value` *argument, you will create a double-ended range slider.*

- `sliderInput()`

selectInput()

- Creates a drop-down menu user can use to choose from.

- Usage:

```
selectInput(inputID = "Slot used to access variable",
  label = "Display label for the control",
  choices = "List of values to select from")
```

  - Note: *If elements of the list are named, then that name, rather than the value, is displayed to the user. It's also possible to group related inputs by providing a named list whose elements are (either named or unnamed) lists, vectors, or factors.*

- `selectInput()`

checkboxInput()

- Creates a box of logical values the user can check from (using checkboxGroupInput() to group them).

- Usage:

```
checkboxInput(inputID = "Slot used to access variable",
  label = "Display label for the control",
  value = TRUE or FALSE) (default)
```

  - Note: *This is structured for a TRUE or FALSE checkbox.*

- checkboxInput()
- checkboxGroupInput()

radioButtons()

- Creates buttons that the user can only select one choice at a time.
- Usage:

```
radioButtons(inputID = "Slot used to access variable",
   label = "Display label for the control",
   choices = "List of values to select from")
```

  - Note: *If elements of the list are named, then that name, rather than the value, is displayed to the user. It's also possible to group related inputs by providing a named list whose elements are (either named or unnamed) lists, vectors, or factors.*

- `radioButtons()`

Widget Functions

- All widget functions contain an `inputId` and a `label` argument.
- We have only covered a few of the widgets available.
- These widgets are used to get input values from users to be sent to the server.

Example 4

- Using the app.R file from Example 3.
- Add the following widgets to the application:
  1. A `sliderInput()` (min = 0, max = 10)
  2. A `radioButtons()` with four choices (A, B, C, D)
- Make sure to include a ',' after each element.
- Run the application.

Dynamic Outputs

- The UI uses **reactive output** elements to display output values from the server.
- These elements are similar to static contents, but it displays changing content produced by the server.
- Example: A chart that updates its contents when a user selects different data to display.
- These reactive (dynamic) outputs will automatically update when the server sends new information to display.

`textOutput()`

- Displays output as plain text (use htmlOutput() if you want to render HTML content)
- Usage:

```
textOutput(outputId = "output variable to read the value from")
```

- `textOutput()`

`tableOutput()`

- Displays output as a data table (similar to `kable()` in R markdown).
- Usage:

```
tableOutput(outputId = "output variable to read table
from")
```

- Note: *The* `dataTableOutput()` *function from the DT package will display an interactive table.*
- `tableOutput()`

`plotOutput()`

- Displays output as a graphical plot, such as one created with *ggplot2*.
- Usage:

```
plotOutput(outputId = "output variable to read the plot
from")
```

- Note: *The* `plotlyOutput()` *function from the plotly can be used to render an interactive plot.*
- `plotOutput()`

verbatimTextOutput()

- Displays content as a formatted code block, such as if you wanted to print a non-string variable like a vector or a data frame.
- Usage:

```
verbatimTextOutput(outputId = "output variable to read
the plot from")
```

- verbatimTextOutput()

Layouts I

- You can specify how content is organised on the page by using different **layout** content elements.
- Layout elements are used to specify the position of different pieces of content on the page.
- Layout functions all take as a sequence of of other content elements that will be shown on the page following a specific layout.
- The `fluidPage()` layout from our example places each elements *following* their order on the page.

`sidebarLayout()`

- One of the most commonly used layout elements is the `sidebarLayout()` function.

- Organises your content into two columns:
  1. *Sidebar*, usually used for widgets or related content.
  2. *Main* section, usually used to present reactive outputs such as tables or plots.

- We need to pass the `sidebarLayout()` two arguments:
  1. A `sidebarPanel()` element that contains the content for the sidebar.
  2. A `mainPanel()` element that contains the content for the main section.

Example 5

- Use your code from Example 4 and the Example 5 app in moodle to do the following:
  1. Create a sidebar and main panel.
  2. Add static content labeling each element (sidebar or main panel).
  3. Place all of your widgets in the sidebar.
  4. Place your text output in the main panel.

`tabPanel()` in `navbarPage()`

- Use the `tabPanel()` to create a tab, fill in the contents as you would a regular UI.
- You can then create multiple tabs in the same way.
- Save the elements in variables and then include them in your UI object.
- *This may be useful if we want to add a navigation bar with tabs to the top of our app.*

Layout II

- Layouts and their contents are often nested, it is a good idea to use line breaks and indents to help visualise the structures.
- You can also store the returned layouts in variables.
- Save the elements in variables and then include them in your UI object.
- *This may be useful if we want to add a navigation bar with tabs to the top of our app.*

`tabPanel()` in `navbarPage()`

- Use the `tab_page1 <- tabPanel()` to create a and save a tab, fill in the contents as you would a regular UI.
- You can then create multiple tabs in the same way.
- Create your UI object by passing the tabs objects to the `navbarPage()` function.
- Example:

```
ui <- navbarPage( "Application Title",
tab_page1,
tab_page2,
tab_page3 )
```

Example 6

- Use your code from Example 5 and the Example 6 app in moodle to do the following:

  1. Create an application with three tabs (each having a sidebar and a main panel).
  2. Add a widget to each tab in the sidebar.
  3. Place your text output in the main panel.
  4. Run your application.

Developing Application Servers

Application Servers

- The **server** defines and processes the data that will be displayed by the UI.
- Shiny servers are created by defining a function rather than calling a provided one.
- The function must be be defined to take at least two arguments:
  1. A list to hold `input` values.
  2. A list to hold `output` values.
- *We can see this in the examples we have done so far.*

Server Function

- The **server function** is just a normal function used to *set up* the application's reactive data processing.
- You can include any code statements that would normally go in a function.
- The code will only be ran once when the application is first started (unless defined as part of a render function).

`input` List

- The `input` list contains any values stored in the control widgets in the UI.
- Each `inputId("name")` in a control widget will be the key in this list.
- *In out examples we have* `inputId = "username"` *as the labeled* `input` *list value*.
- These lists are **reactive**, so the values will automatically change as the user interacts with the UI's control widgets.

`output` List

- The primary purpose of the server is to assign new values to the `output` list.
- The **render functions** produce the values assigned to the `output` list.
- The render functions ensure that the content sent to the UI is in a format that can be understood by the UI's outputs.
- **The server uses different render functions for the different types of output it provides.**

Render Functions

- The result of a render function *must* be assigned to a key in the `output` list that matches the `outputId("name")`.
- Example: If the UI includes `outputId = "message"`, then the value **must** be assigned to `output$message`
- The *type* of render function must match the *type* of reactive output.
- *Shiny server functions will usually have multiple render function assigning values to the* `output` *list*.

Some Render Functions

Render Function (Server)	Reactive Output (UI)	Content Type
`renderText()`	`textOutput()`	Unformatted text (strings)
`renderTable()`	`tableOutput()`	A simple data table
`renderDataTable()`	`dataTableOutput()`	Interactive data table (*DT* package)
`renderPlot()`	`plotOutput()`	A graphical plot (e.g. ggplot object)
`renderPlotly()`	`plotlyOutput()`	An interactive *Plotly* plot
`renderLeaflet()`	`leafletOutput()`	An interactive Leaflet map
`renderPrint()`	`verbatimTextOutput()`	Any output produced with `print()`

Reactive Expressions

- All render functions take a **reactive expression** as an argument.
- Reactive expressions are a lot like a function:
  - They are written as a block of code in curly braces ({}) that returns the value to be rendered.
- The only differences between writing a function and a reactive expression is that you don't include the keyword `function` or a list of arguments (only the block of code).
- Your server defines a series of *functions* (render functions) that specify how the `output` should change based on changes to the `input`.

Example 7

- Use your code from Example 6 and the Example 7 app in moodle.
- Create an application with three tabs (each having a sidebar and a main panel):
  1. Tab 1: Ask the user for their name and welcome them to your application by name.
  2. Tab 2: Use a slider to allow the user to select the number of bins in a histogram for the highway MPG (*MPG.highway*) from the *Cars93* dataset in the *MASS* package.
  3. Tab 3: Use radio buttons to allow the user to create a boxplot for the *Horsepower* of a specific *DriveTrain*.
- Be sure to add appropriate static content throughout your application and run your app when you are finished.

Hosting Applications

- In order to share your application with the world, you will need a website that is able to host Shiny applications.
- GitHub for example, does not come with an `R interpreter` to run your server. You will need to host somewhere that has an `R interpreter`.
- RStudio has a hosting platform at *shinyapps.io*
- There are also other hosting platforms and your employers may have their own.

Hosting on *shinyapps.io*

- You can sign up using your GitHub account or a Google account.

- Once signed up:
  1. Select an account name (it will be part of the URL so be careful).
  2. Install the *rsconnect* package.
  3. Set up your authorization token (password) for uploading your app.

     Click the green *Copy to Clipboard* button, and then paste that selected command into the Console in RStudio. (only need to do this once per machine).
  4. Finally, run your app on your machine and click the *Publish* button in the upper-right corner of the app viewer.

- Soon your app should be available online: *https://USERNAME.shinyapps.io/APP_NAME/*

Tips

- Always test and debug your app locally. It is much easier to fix errors on your machine before putting it online.
- The `showLogs()` function from the *rsconnect* will print the error logs of your deployed application.
- Use the correct folder structures (paths). All of the files needed for your application should reside in the same folder. Do not use the `setwd()` function as your hosting platform will have its own working directory.
- Make sure all external packages are referenced with the `library()` function.

Exercise 1

- Take some time to explore the Shiny website. Be sure to familiarize yourself with the different widgets and possible render/output functions.

Exercise 2

- Use your code from Example 7 and create an application with three tabs (each having a sidebar and a main panel):

  1. Tab 1: Ask the user for their name and welcome them to your application by name. Then ask them to select their favourite colour from a *select input* (drop down bar). Use the colour they selected as a colour in subsequent graphics.

  2. Tab 2: Use another two *select inputs* (drop down bars) to allow the user to select which two features of the *iris* dataset they would like to create a scatterplot with. Generate the scatterplot in the main panel of this tab.

  3. Tab 3: Use radio buttons to allow the user to select which feature of the *iris* dataset they would like a summary `summary()` table to be presented for. Return the table in the main panel of this tab.

- Be sure to add appropriate static content throughout your application

References & Resources

1. Michael Freeman, Joel Ross, *Programming Skills for Data Science: Start Writing Code to Wrangle, Analyze, and Visualize Data with R*, 2019, ISBN-13: 978-0-13-513310-1

- https://shiny.posit.co/
- https://shiny.posit.co/r/articles/build/interactive-docs/
- https://www.shinyapps.io/