bslib, dynamic UIs, and publishing

Lecture 19

Dr. Colin Rundel

Shiny & bootstrap

The interface provided by Shiny is based on the html elements, styling, and javascript provided by the Bootstrap library.

As we've seen so far, knowing the specifics of Bootstrap are not needed for working with Shiny - but understanding some of its conventions goes a long way to helping you customize the elements of your app (via custom CSS and other components).

This is not the only place that Bootstrap shows up in the R ecosystem - e.g. both RMarkdown and Quarto html documents use Bootstrap for styling as well.

bslib

The bslib R package provides a modern UI toolkit for Shiny, R Markdown, and Quarto based on Bootstrap.

It facilitates:

- Custom theming of Shiny apps and R Markdown documents.
- Use of modern (or older) versions of Bootstrap and Bootswatch
- Creation of delightful and customizable Shiny dashboards via useful UI components (e.g., cards, value boxes, sidebars, etc)

bslib components

Cards

> Cards are a common organizing unit for modern user interfaces. At their core, they're just rectangular containers with borders and padding. However, when utilized properly to group related information, they help users better digest, engage, and navigate through content. This is why most successful dashboard/UI frameworks make cards a core feature of their component library.

```
card(
  card_header(
    "A header"
  ),
  card_body(
    shiny::markdown(
      "Some text with a [link](https://githu
    )
  )
)
```

More options

```
card(
  max_height = 225,
  card_header(
    "A long, scrolling, description",
    class = "bg-dark"
  ),
  card_body(
    lorem::ipsum(
      paragraphs = 3,
      sentences = 5
    )
  )
)
```

```
card(
  max_height = 225,
  card_header(
    "A leaflet map",
    class = "bg-success"
  ),
  card_body(
    class = "p-0",
    leaflet::leaflet() |>
      leaflet::addTiles()
  )
)
```

Multiple card bodies

```
card(
  max_height = 500,
  card_header(
    "A long, scrolling, description",
    class = "bg-dark"
  ),
  card_body(
    leaflet::leaflet() |>
      leaflet::addTiles()
  ),
  card_body(
    lorem::ipsum(paragraphs = 1, sentences =
  )
)
```

Value boxes

These are simplified cards that are designed to show basic numeric or text values.

```
library(bsicons)
library(htmltools)

value_box(
  title = "I got",
  value = "99 problems",
  showcase = bs_icon("music-note-beamed"),
  theme = "cyan",
  p("bslib ain't one", bs_icon("emoji-smile"
  p("hit me", bs_icon("suit-spade"))
)
```

Multiple value boxes

```
library(bsicons)
library(htmltools)

page_fillable(
  value_box(
    title = "1st value",
    value = "123",
    theme = "",
    showcase = bs_icon("bar-chart"),
    p("The 1st detail")
  ),
  value_box(
    title = "2nd value",
    value = "456",
    showcase = bs_icon("graph-up"),
    theme = "danger",
    p("The 2nd detail"),
    p("The 3rd detail")
  )
)
```

Layouts

Fixed layout

```
library(leaflet)
page_fillable(
  card(
    max_height = 200,
    card_header("Card 1"),
    lorem::ipsum(1,3)
  ),
  card(
    max_height = 100,
    card_header("Card 2"),
    "This is it."
  ),
  card(
    max_height = 200,
    card_header("Card 3"),
    leaflet() |> addTiles()
  )
)
```

Column layout

```
library(leaflet)
page_fillable(
  layout_columns(
    height = 200,
    card(
      card_header("Card 1"),
      lorem::ipsum(1,3)
    ),
    card(
      card_header("Card 2"),
      "This is it."
    )
  ),
  layout_columns(
    height = 300,
    card(
      card_header("Card 3"),
      leaflet() |> addTiles()
    )
  )
)
```

Column layout

Column widths layout

```
library(leaflet)
page_fillable(
  layout_columns(
    col_widths = c(8, 4, -1, 10, -1),
    row_heights = c("200px", "300px"),
    card(
      card_header("Card 1"),
      lorem::ipsum(1,3)
    ),
    card(
      card_header("Card 2"),
      "This is it."
    ),
    card(
      card_header("Card 3"),
      leaflet() |> addTiles()
    )
  )
)
```

Column widths layout

Dynamic layouts

```
library(leaflet)
layout_column_wrap(
  width = 1/2,
  card(
    max_height = 250,
    card_header("Card 1"),
    lorem::ipsum(1,3)
  ),
  card(
    max_height = 250,
    card_header("Card 2"),
    "This is it."
  ),
  card(
    max_height = 250,
    card_header("Card 3"),
    leaflet() |> addTiles()
  )
) |>
  anim_width("100%", "33%")
```

Dynamic layouts

Responsive columns

```
library(leaflet)
layout_column_wrap(
  width = "200px",
  fixed_width = FALSE,
  card(
    max_height = 250,
    card_header("Card 1"),
    lorem::ipsum(1,3)
  ),
  card(
    max_height = 250, fill=FALSE,
    card_header("Card 2"),
    "This is it."
  ),
  card(
    max_height = 250,
    card_header("Card 3"),
    leaflet() |> addTiles()
  )
) |>
  anim_width("100%", "33%")
```

Responsive columns

Nested Layouts

```
library(leaflet)
layout_column_wrap(
  width = 1/2,
  card(
    card_header("Card 1"),
    lorem::ipsum(1,3)
  ),
  layout_column_wrap(
    width = 1,
    heights_equal = "row",
    card(
      card_header("Card 2"),
      "This is it."
    ),
    card(
      max_height = 300,
      card_header("Card 3"),
      leaflet() |> addTiles()
    )
  )
)
```

Nested Layouts

Dynamic UI components with `!!!`

When building UIs dynamically (e.g., in `renderUI()`), you often need to pass a list of components to layout functions. The `!!!` operator (splice/unquote-splice) from rlang expands a list into individual arguments.

```
# Create a list of cards
cards = list(
  card(card_header("Card 1"), "Content 1"),
  card(card_header("Card 2"), "Content 2"),
  card(card_header("Card 3"), "Content 3")
)

# Splice the list into layout_column_wrap
layout_column_wrap(width = 1/3, !!!cards)
```

This can also be done with `do.call()` but `!!!` is often more readable and integrates better with tidyverse code.

For either approach be careful about naming of list elements - these become argument names which may or may not do

Theming

Bootswatch

Due to the ubiquity of Bootstrap a large amount of community effort has gone into developing custom themes - a large free collection of these are available at bootswatch.com/.

`bs_theme()`

Provides a high level interface to adjusting the theme for an entire Shiny app,

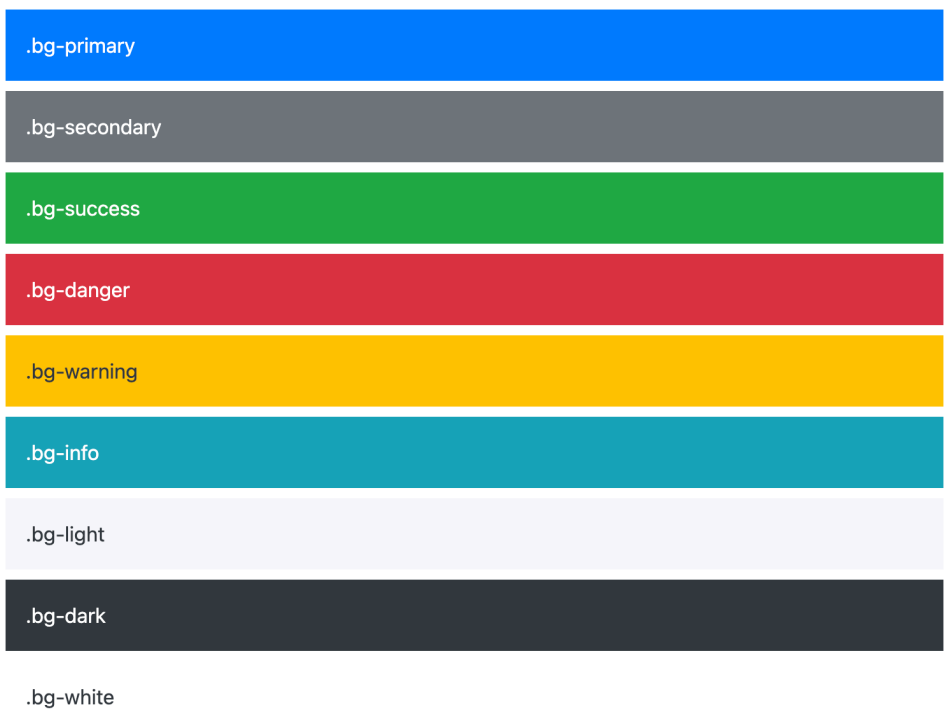
- Change bootstrap version via `version` argument
- Pick a bootswatch theme via `bootswatch` argument
- Adjust basic color palette (`bg`, `fg`, `primary`, `secondary`, etc.)
- Adjust fonts (`base_font`, `code_font`, `heading_font`, `font_scale`)
- and more

The object returned by `bs_theme()` can be passed to the `theme` argument of `fluidPage()` and similar page UI elements.

In a Shiny app dynamic theming can be enabled by including `bs_themer()` in the server function of your app.

Bootstrap colors palettes

Bootstrap provides a large number of built-in colors for styling html elements via CSS. Within these colors, a smaller subset are selected to create a color palette that is the basis for most themes and is used for the default styling of Bootstrap components.



- *Primary* - Main theme color, used for hyperlinks, focus styles, and component and form active states.
- *Secondary* - used to complement the primary color without drawing too much attention, used for less prominent UI elements.
- *Success* - used for positive or successful actions and information.
- *Danger* - used for errors and dangerous actions.
- *Warning* - used for non-destructive warning messages.
- *Info* - used for neutral and informative content.
- *Light* - Additional theme option for less contrasting colors.
- *Dark* - Additional theme option for higher contrasting colors.

Bootstrap theme colors with Shiny

Theme colors can be specifically applied to most Shiny elements using the `class` argument.

```
1 actionButton("primary", "Primary", class = "btn-primary")
```

```
1 actionButton("primary", "Danger", class = "btn-danger")
```

```
1 actionLink("danger", "Danger", class = c("link-info","bg-success"))
```

Note - bootstrap classes make use of prefixes to help specialize the behavior to specific types of html elements.

thematic

This package provides a way of simplifying the process of theming ggplot2, lattice, and base R graphics. Additionally, it also provides mechanisms to automatically integrate these themes with Shiny apps, RMarkdown and Quarto documents.

While it is not perfect, it can do much of the heavy lifting and can get you close to a working theme with a minimal amount of intervention.

In order to enable this automatic theming, just include `thematic_shiny()` in your R script before you call `shinyApp()`.

Deploying Shiny apps

Organizing your app

For deployment generally apps will be organized as a single folder that contains all the necessary components (R script, data files, other static content).

- Pay attention to the nature of any paths used in your code
  - Absolute paths are almost certainly going to break
  - Relative paths should be to the root of the app folder
- Static files (e.g. css, js, etc.) are generally placed in the `www/` subfolder
- Your script *does not* need to be named `app.R` or `ui.R`/`server.R` but some tools prefer this pattern
- Check / think about package dependencies

Demo - shinyapps.io

This is a cloud based hosting service for Shiny apps provided by Posit. It provides a very easy to use interface for deploying apps directly from RStudio.

For minimal use it is free, but there are paid plans for more intensive use.

Demo - shinylive

One of the really exciting developments in the last couple of years is the ability to run R (and Python) inside a web browser using WebAssembly. shinylive is a package that lets you bundle your shiny app as a static website that can be hosted anywhere you can host static html.

Other publishing options

- For other R users - you can share your script(s) and data directly
  - or better yet, bundle them into an R package
- Run a local instance of shiny server
- Use shinyapps.io (public) or posit.cloud (within a team)
- Use Posit Connect