CREATING WEB APPLICATIONS WITH R-SHINY

A SHINY APPLICATION APPLIED ON VOLLEYBALL DATA

20630 - Introduction to Sports Analytics (Group 1)

Andrea Marchesi (3272606)
Emanuele Giacomo Branzoni (3273784)
Zhuohao Hu (3128849)

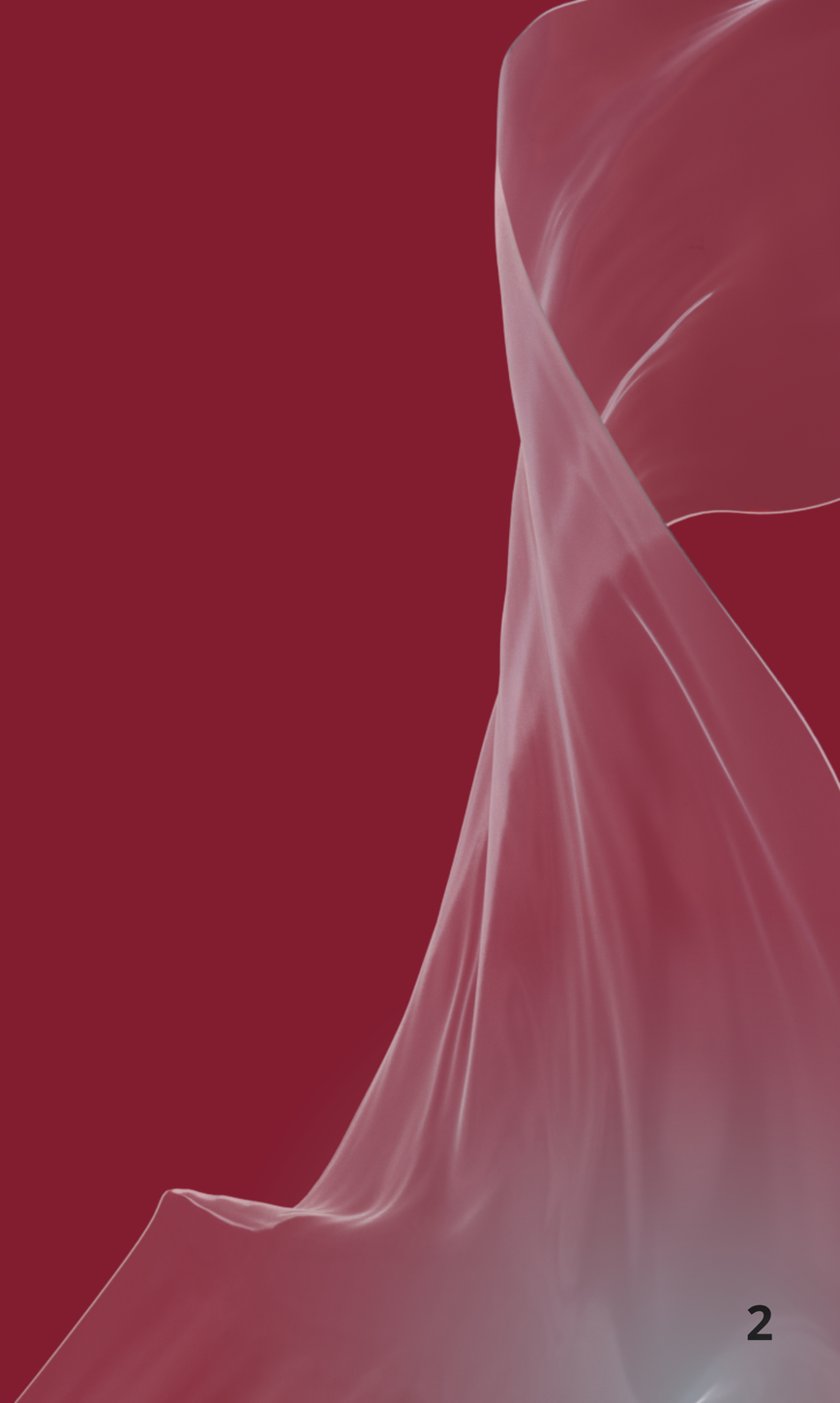
ROADMAP

PART 1:

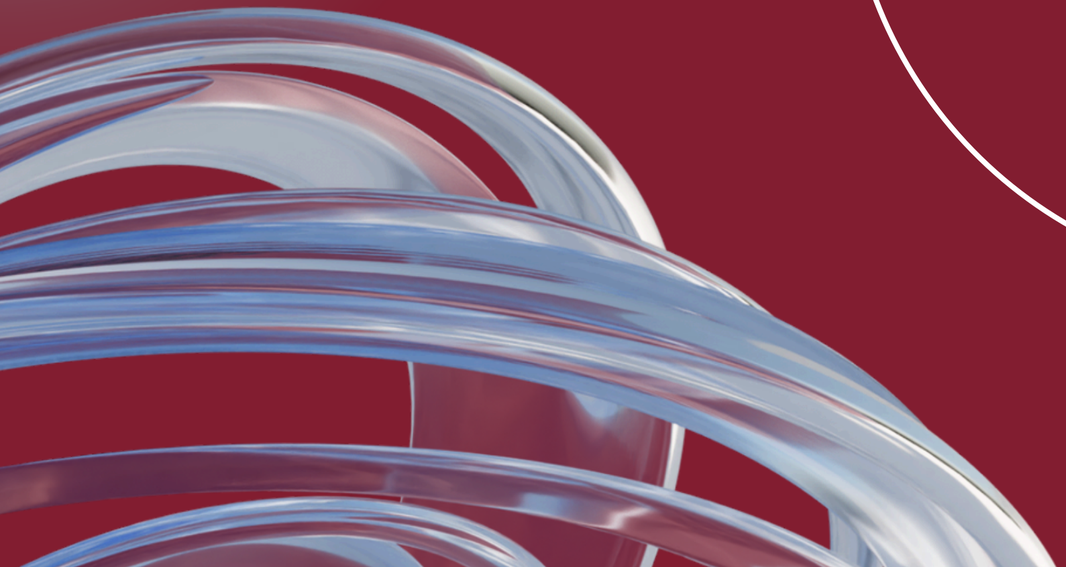
- What is Shiny?
- UI and its Functionalities
- Sever and its Functionalities
- Shiny Workflow
- Introduction to our Data
- Web Scraping Techniques
- R Shiny: Key Points and Alternatives
- Deploying our Application Online

PART 2:

- Introduction to our Shiny Web App
- Main App Structure
- Helpers
- Player Dashoboard Overview
- Player Statistics
- Conclusions
- Shiny Web App Showcase



PART 1



What is Shiny?

Shiny is a framework for creating interactive web applications using R, allowing users to explore and visualize data dynamically. It integrates seamlessly into R, making it a valuable tool for sharing analysis results interactively.

R Shiny consists of two main components:

- UI (User Interface): Defines the graphical layout of the application and the elements users can interact with, such as buttons, sliders, and charts.
- Server: Manages the logic and functionality of the app, processing data and dynamically updating content based on user input.

UI and its Functionalities

Layouts and UI Structure

- Defines the app's structure and organizes UI elements for a user-friendly experience.
- Common layout options:
- FluidPage – Responsive and adjusts to different screen sizes.
- FixedPage – Static layout with a consistent design across devices.
- NavbarPage – Multi-tab layout for organizing sections within an app.

Input Elements

- Allow users to provide values or make selections to interact with the app.

Types of input elements:

- Numeric Inputs & Sliders– Accept user-defined numbers or range selections.
- Sliders are best for small ranges where precision is not critical.

Date Selection

- Date Input – Select a single date.
- Date Range Input – Select a start and end date.
- Customizable format, language, and week start settings for international users.

Limited Choice Inputs

- Select Input – Dropdown menu for predefined options (saves space).
- Radio Buttons – Display all choices at once (best for short lists).
- Customizable options with icons and names for improved usability.

Server and its Functionalities

Shiny automatically calls the server function for each new session, creating a unique environment that manages user interactions, processes inputs, and dynamically updates the UI. . The server function comprises three main components:

- input: Collects user input from the UI.
- output: Controls dynamic UI elements.
- session: Manages user-specific settings.

Reactivity in Shiny allows dynamic output changes based on inputs through reactive dependencies, which include:

- Automatic tracking of input-output relationships.
- Outputs update only when necessary, minimizing redundant computations.
- Shiny recalculates components only upon input changes.
-

Reactive expressions (reactive({...})) enhance optimization by:

- Storing computed results for efficient updates.
- Avoiding code duplication for multiple outputs relying on the same computation.
- Improving performance by caching values.

Shiny Workflow

Folder Structure Summary:

- dataset: Contains all data sources (e.g., .csv files).
- moduls: Stores UI and server modules for different application features (e.g., statplayer_module.R).
- helper: Includes utility scripts for data cleaning and custom metrics.
- www: Holds static resources like images, logos, fonts, and CSS.
- MainApp.R: Main application file that loads data, sources modules, and defines the UI and server.

Modularization involves breaking an application into self-contained modules, each handling specific functions. Benefits include:

- Reusability across projects
- Improved maintainability through easier debugging and testing
- Enhanced scalability for adding features without disruption

Helper Scripts

- Helper scripts (e.g., prep_player_dashboard_helper.R) are used for data preprocessing and calculations, such as:
- Cleaning and transforming raw datasets.
- Computing statistics like percentages, totals, or rankings.
- Preparing reactive objects to be used inside modules.

Introduction to our Data

Tokyo OG2020

For our web app, we used readily available data coming from Olympic Games Tokyo 2020 Men's Volleyball Competition.

What kind of data?

Scout data from live matches (original data is in dvw extension, which is the typical output file of volleyball scouting files)

- 38 individual matches scout files were grouped together.
- Data processing and manipulation using datavolley and dplyr libraries.

↓

Our final dataset on which we have developed our Shiny web app was in csv format and contains detailed information on each individual match.

More in depth

Each row in the dataset represents a touch-id, so we have play-by-play data.
The variables we were most interested in were:

- Match-id data.
- Team and Player data: they include names for all teams and individual players.
- Skill and Skill Evaluation data: they report skill's names performed by each player (Attack, Serve, Reception, Block, Dig) and their evaluation (Positive, Negative, Error, etc)
- Coordinate data about the ball position: namely, positional data tracking where the ball was hit on the court.
- Scoring data, showing the number of points scored.

Web Scraping Techniques

We integrated additional information for each player (image, birthdate, height, position, and nationality) through some web scraping methods and via the usage of Wikipedia API, as follows:

Wikipedia & Wikidata functions

We defined a number of custom functions to retrieve player-related information:

- get_wikipedia_title(): finds the Wikipedia page title for a given player name using the Wikipedia API.
- get_wikipedia_image(): retrieves the player's thumbnail image URL.
- get_wikidata_id(): retrieves the Wikidata entity ID for a player from their Wikipedia page.
- get_wikidata_label(): takes Wikidata entity IDs and retrieves the human-readable labels (names) in English from Wikidata.
- scrape_wikipedia_info(): scrapes fallback info (especially birthdate and height) from the player's Wikipedia infobox if not available from Wikidata.
- get_player_info(): combines Wikidata info and scrapes Wikipedia if necessary to fill in missing data.

Subsequent steps

- We used our dataset with player and team information (in csv format) and cleaned it for this purpose, selecting only relevant columns (like player_name and few others) and applying corrections for name mismatches (using case_when() to adjust player names to correct Wikipedia titles).
- We iterated through each player name, in order to retrieve Wikipedia title, image URL, Wikidata ID, and player details.
- Finally, we build the code so that it loops again through each player, detects the image format, and downloads each player's photo to a specific folder, naming each file after the player.

R Shiny: Key Points and Alternatives

Main Features of R Shiny

- Reactive programming: automatically updates outputs based on user input.
- Customizable UI: supports HTML, CSS, and JavaScript for a tailored interface design.
- Powerful data visualization: integrates with libraries like ggplot2 and plotly for interactive charts.
- Seamless integration with R packages: works with dplyr, tidyr, and shinyWidgets for advanced analytics.

Alternatives to R Shiny

- Dash (Python): a Python-based alternative with similar functionalities.
- Bokeh (Python): primarily for interactive data visualizations, can be combined with web frameworks.
- JavaScript frameworks (React, Vue.js): more flexible but require deeper programming knowledge.

Deploying our Application Online

Finally, we deployed the app online using shinyapps.io, a cloud-based service by Posit.

What to deploy

When we deployed, we needed to include in the same folder:

- the app.R file: the main Shiny app logic.
- modules/ and helpers/ folders: where we inserted the files for defining reusable functions and modular UI/server components.
- www/ folders: namely, the static assets (like images) the app uses.
- all dependencies: custom functions and local packages that the app sources (besides those from CRAN - Comprehensive R Archive Network, the primary package repository in the R community).

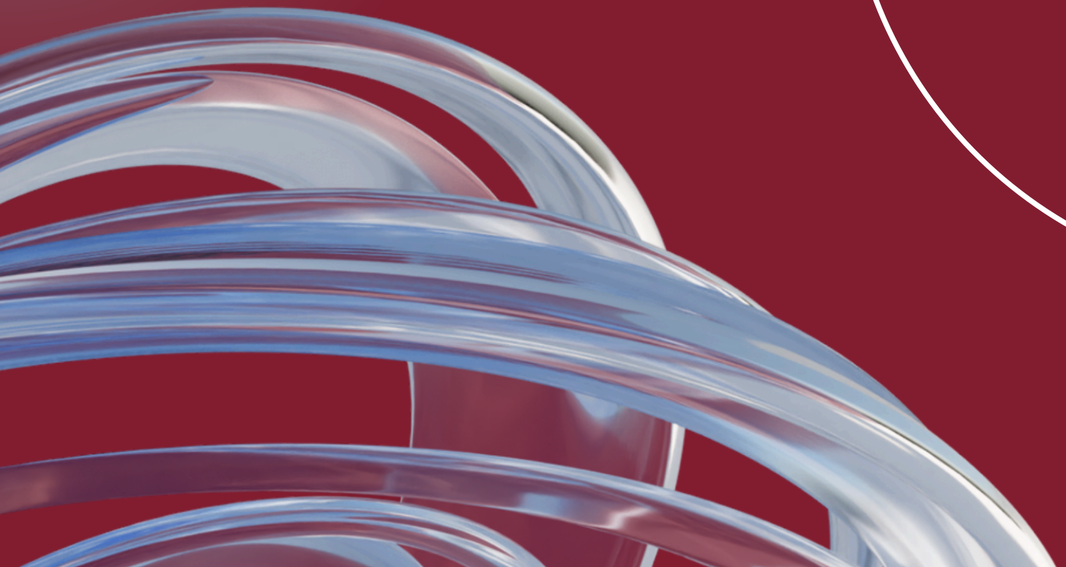
Managing NON-CRAN packages

In our project, we used, among others, two packages (ovlytics and datavolley) which are not on CRAN but are GitHub packages. So, we handled this aspect by using renv: it is an R package that creates isolated and reproducible environments for R projects by capturing all package versions (including local and GitHub ones) in a renv.lock file so they can be restored anywhere, including on shinyapps.io servers.

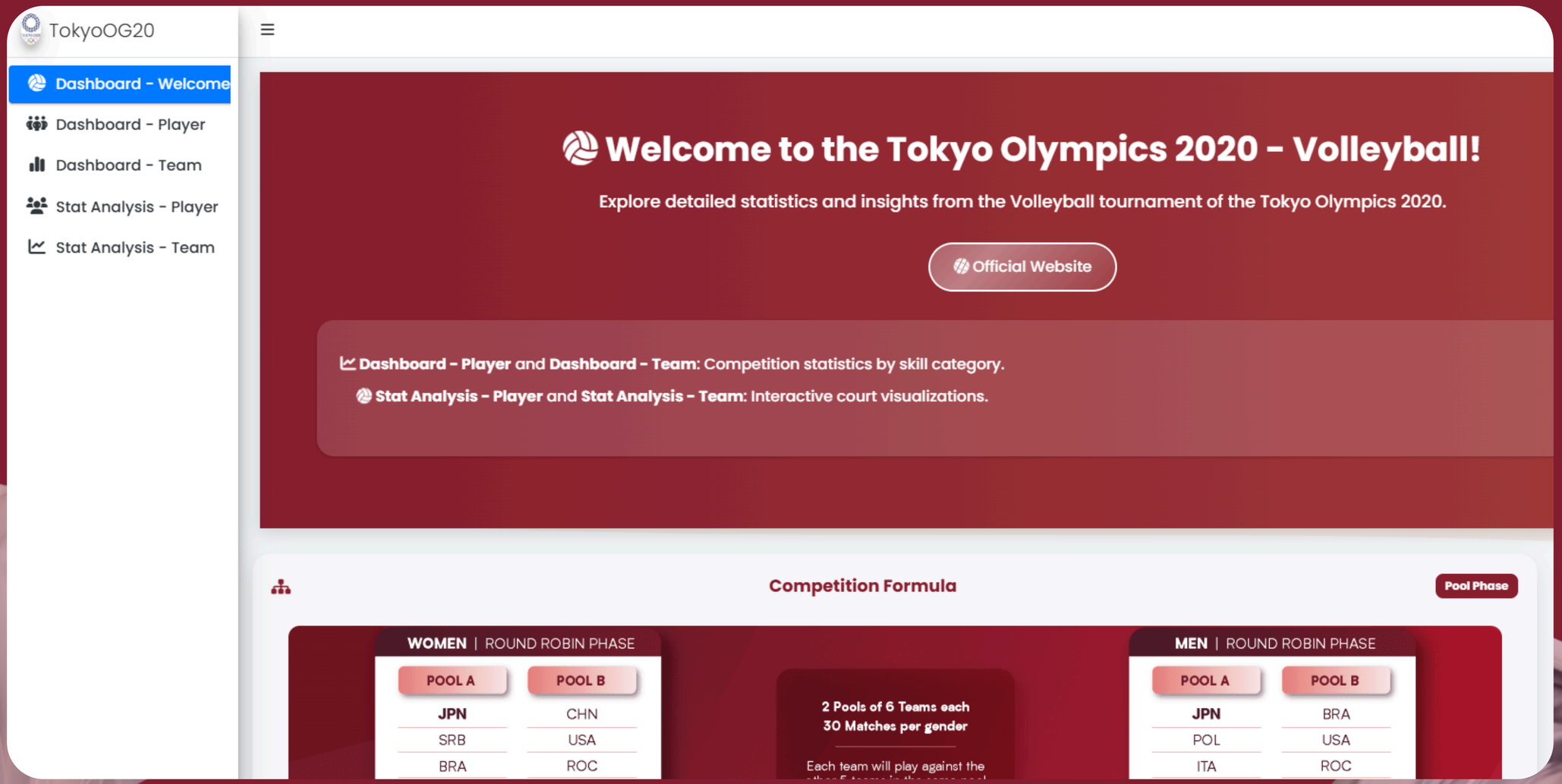
Therefore, we proceeded as follows:

- we initialized renv and installed all required packages inside the renv environment
- we snapshotted the environment and tested it locally
- we deployed on shinyapps.io with rsconnect + renv

PART 2



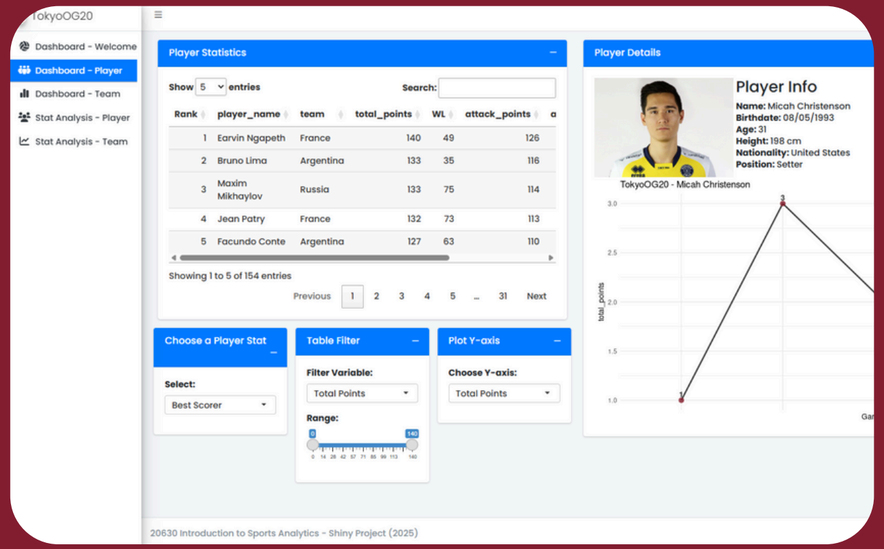
TOKYO OG20 MEN'S VOLLEYBALL

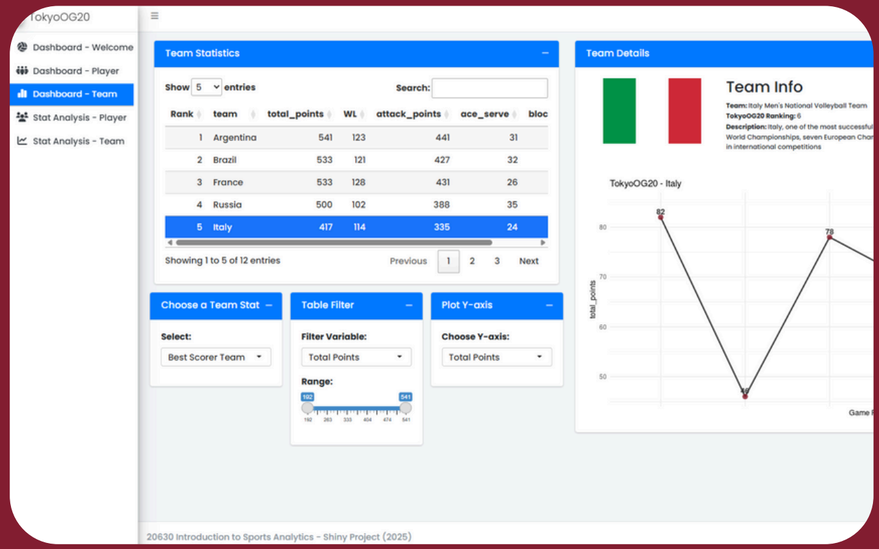


TOKYO OG20 MEN'S VOLLEYBALL

Dashboard Player
Dashboard Team

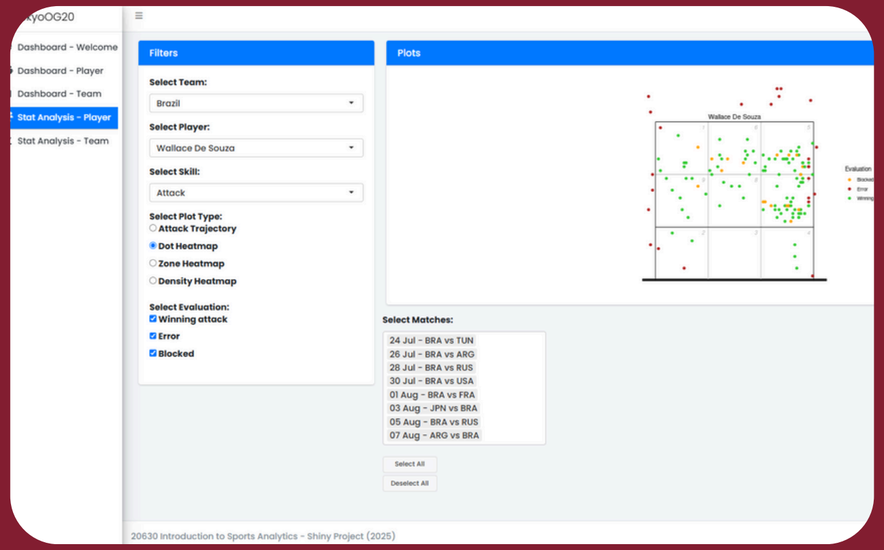
- Player dataset
- Player info box
- Player plot
- Filters
  - dataset
  - table filter
  - plot y-axis

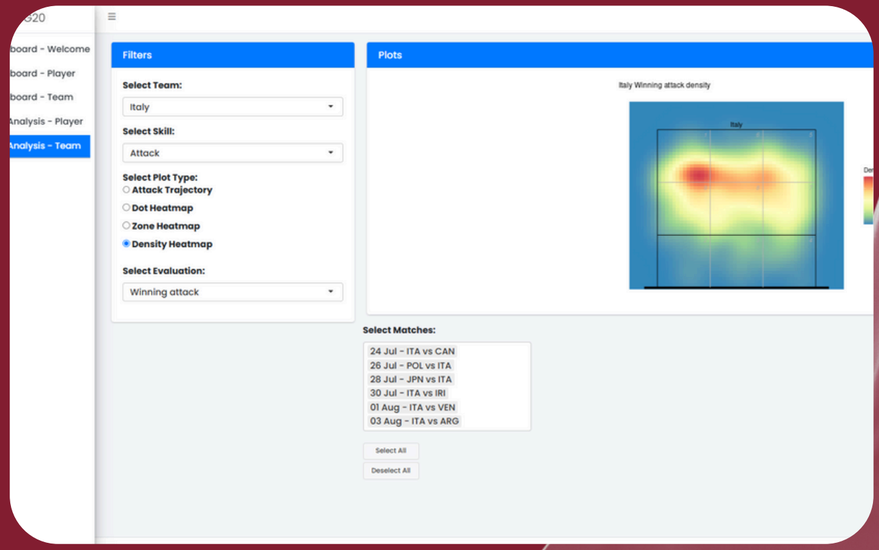




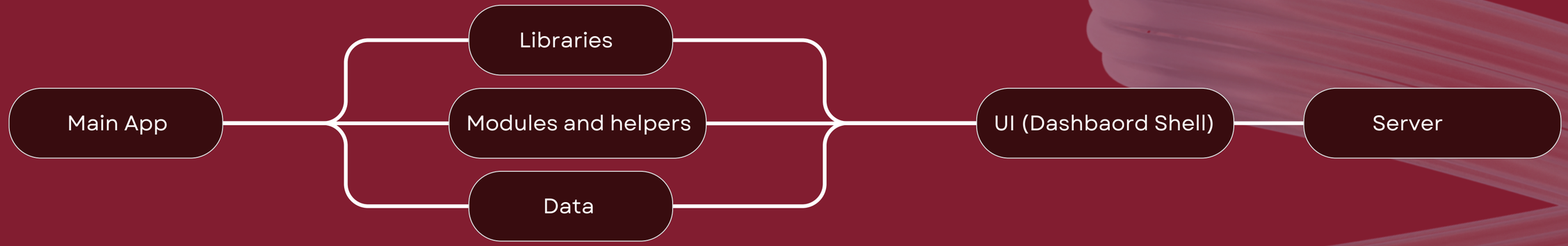
Stat Analysis Player
Stat Anaysis Team

- Plot
- Filters
  - team,player
  - skill, evaluation
  - plot type
  - match filter





MAIN APP STRUCTURE - CODE SNIPPETS




```r
library(shiny)
library(bs4Dash)
library(dplyr)
library(DT)
library(ggplot2)
library(ovlytics)
library(datavolley)
library(shinycssloaders)

source("modules/dashwelcome_module.R")
source("modules/statplayer_module.R")
source("modules/statteam_module.R")
source("modules/playerdashboard_module.R")
source("modules/teamdashboard_module.R")
source("helpers/prep_player_dashboard_helper.R")
source("helpers/prep_team_dashboard_helper.R")

OL21_plays <- data.frame(read.csv('data/OL21_df_plays_cleaned.csv', header = TRUE))
full_player_data <- data.frame(read.csv("data/full_player_data.csv"))
team_description <- data.frame(read.csv("data/team_description.csv"))
```

Libraries:

- Shiny related (shiny, bs4Dash,shinycssloaders,DT).
- R data manipulation and plotting: dyplr, ggplot2.
- Volleyball specific: datavolley, ovlytics.

Sources:

- Retrieving the module and helper files.

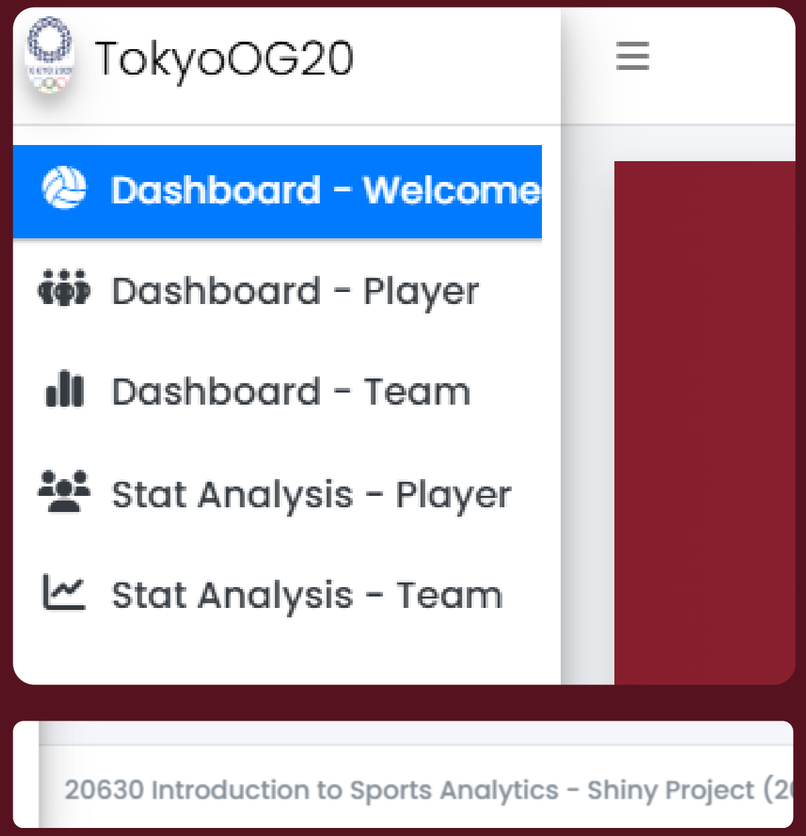
Data loading:

- Player photo and information (age, height, position etc.).
- Team flag and description.
- Volleyball play data (skill, coordinate data, etc.)

MAIN APP UI - CODE SNIPPETS

Simplified UI structure of Main App (Dashboard Shall)
Based on package {bs4Dash} for a modern UI

```
ui <- dashboardPage(
  header = dashboardHeader(title = "TokyoOG20"),
  sidebar = dashboardSidebar(
    sidebarMenu(
      menuItem("Dashboard - Welcome",tabName = "dashwelcome",
               icon = icon("volleyball"),selected = TRUE),
    )),
  footer = dashboardFooter(
    left = "20630 Introduction to Sports Analytics - Shiny Project (2025)"
  ),
  body = dashboardBody(
    tabItems(
      tabItem(tabName = "dashwelcome",
              dashwelcomeUI("welcome")),
    )
)
```



Key elements:

- Header: customizable with title and image.
- Sidebar: navigation menu for different sections (icon, selected, tabName).
- Footer: additional information, credits, or copyright.
- Body: the main content area where outputs and interactive elements appear (tabName, dashWelcomeUI).

MAIN APP SERVER - CODE SNIPPETS

Main App's Server logic
(The backend functionality that processes data and connects it to the UI)

```
server <- function(input, output, session) {
  # Preprocess the data reactively (prep_player_dashboard_helper.R)
  processed_data <- reactive({preprocess_data(OL21_plays)})
  match_data <- reactive({preprocess_match_data(OL21_plays)})

  # Preprocess the data reactively (prep_team_dashboard_helper.R)
  processed_data_team <- reactive({preprocess_data_team(OL21_plays)})
  match_data_team <- reactive({preprocess_match_data_team(OL21_plays)})

  dashwelcomeServer("welcome")
  playerdashboardServer("playerdashboard", processed_data,
                        match_data,full_player_data)
  teamdashboardServer("teamdashboard",processed_data_team,
                      match_data_team,team_description)
  statplayerServer("statplayer", OL21_plays)
  statteamServer("statteam",OL21_plays)
}

shinyApp(ui, server)
```

The server function handles:

- Data processing
- Reactive programming
- Module integration

Why reactive expressions?

- Avoid **redundant** computations
- Automatically **update outputs** when data changes

Key points:

- Passing pre-processed data
- Passing relevant data to server modules Server

HELPERS - CODE SNIPPETS

```r
preprocess_data <- function(data) {
  df <- data %>%
    group_by(team, player_name) %>%
    summarize(
      # Total points
      total_serve = sum(ifelse(skill == "Serve", 1, 0), na.rm = TRUE),
      error_serve = sum(ifelse(skill == 'Serve' & evaluation_code == '=', 1, 0), na.
      bad_serve = sum(ifelse(skill == 'Serve' & evaluation_code %in% c("-", "!"), 1,
      positive_serve = sum(ifelse(skill == 'Serve' & evaluation_code %in% c("+", "/"
      ace_serve = sum(ifelse(skill == 'Serve' & evaluation_code == '#', 1, 0), na.rm
```

```r
  ) %>%
  mutate(
    # Serve percentages and efficiency
    serve_point_percentage = round((ace_serve / total_serve) * 100, 2),
    serve_error_percentage = round((error_serve / total_serve) * 100, 2),
    serve_efficiency = round(((ace_serve - error_serve) / total_serve) * 100
```

```r
      dig_efficiency = round(((perfect_dig + good_dig - error_dig) / s
    ) %>%
    mutate(across(-c(player_name, team, file_name), as.numeric))
}
```

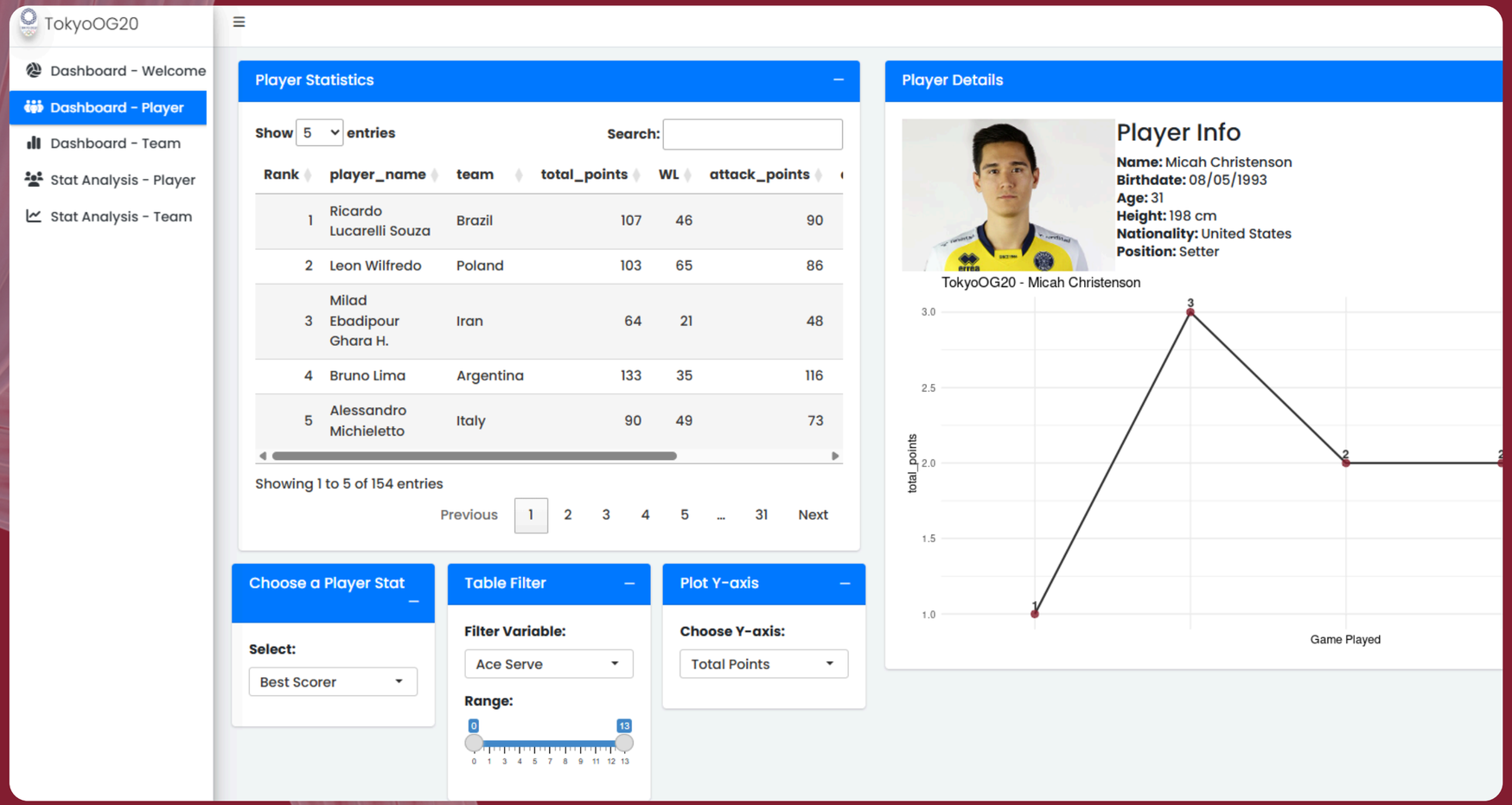
prep_player_dashboard_helper.R and prep_team_dashboard_helper.R are the helper files.

They contain functions that, given the data, generate dataframes that will be used in the dashboard pages.

The main difference is in the grouping (by team and player vs by team only).

PLAYER DASHBOARD OVERVIEW

This module allows for an interactive dashboard for volleyball player statistics analysis. We provide possibility to generate interactive tables and visualizations, along with a player info box.



Dynamic UI
Possibility of dynamic data filtering and sorting

Player Info Box
Displaying player photo and player details

Performance trend visualization
Dynamic Y-axis selector for the plot

PLAYER DASHBOARD UI - CODE SNIPPETS

```r
playerdashboardUI <- function(id) {
  ns <- NS(id)
  tagList(
    fluidRow(

      column(width = 5,
             box(width = 12, title = "Player Statistics", status = "primary", solidHeader = TRUE,
                 DTOutput(ns("table")) %>% withSpinner()),

             fluidRow(
               box(width = 4, title = "Choose a Player Stat", status = "primary", solidHeader = TRUE,
                   selectInput(ns("dataset"), "Select:", choices = c(...))
               ),
               box(width = 4, title = "Table Filter", status = "primary", solidHeader = TRUE,
                   selectInput(ns("tablevariable"), "Filter Variable:", choices = NULL),
                   sliderInput(ns("num"), "Range:", min = 1, max = 100, value = c(1, 100))
               ),
               box(width = 4, title = "Plot Y-axis", status = "primary", solidHeader = TRUE,
                   selectInput(ns("y_axis"), "Choose Y-axis:", choices = NULL)
               )
             )
      ),
```

```r
      # Right Panel
      column(width = 7,
             box(width = 12, title = "Player Details", status = "primary", solidHeader = TRUE,
                 fluidRow(
                   column(3,
                          div(style = "text-align: center; height: 180px;",
                              uiOutput(ns("player_photo")))),
                   column(9,
                          uiOutput(ns("player_info")) %>% withSpinner())),
                 fluidRow(
                   column(12,
                          plotOutput(ns("player_plot"), height = "450px") %>% withSpinner()
```

TWO COLUMNS LAYOUT

Left Panel Components

Player Statistics Table:

- Displays filtered player rankings
- Interactive row selection
- Auto-scrolling and pagination

Control Boxes:

- Dataset selector (attackers, blockers, etc.)
- Variable filter with dynamic range slider
- Y-axis selector for the plot

Right Panel Components

Player Profile

- Photo display
- Detailed player information (name, height, position etc.)

Performance Plot

- Line chart showing performance across matches
- Dynamic Y-axis based on selection

PLAYER DASHBOARD SERVER - CODE SNIPPETS

```
5
6     # Reactive data selection
7     current_dataset <- reactive({
8       req(input$dataset)
9       processed_data()[[input$dataset]]
10    })
11
12    # Dynamic UI updates
13    observe({
14      req(current_dataset())
15      numeric_cols <- names(Filter(is.numeric, current_dataset()))
16      updateSelectInput(session, "tablevariable", choices = numeri
17    })
18
19    # Filtered data
20    filtered_dataset <- reactive({
21      req(current_dataset(), input$tablevariable, input$num)
22      current_dataset() %>% filter(.data[[input$tablevariable]] >=
23    })
24
25    # Outputs
```

Server Logic

Reactive Data Flow:

current_dataset <- reactive({ ... })

- Selects data based on input$dataset (from UI dropdown) and reacts to changes in input$dataset

observe()

- Watches current_dataset() for changes
- Updates selectInput("tablevariable") with numeric columns from the dataset

filtered_dataset <- reactive({ ... })

- Filters data based on user-selected variable and range

PLAYER DASHBOARD SERVER - CODE SNIPPETS

```
    })

    # Outputs
    output$table <- renderDT({ ... })
    output$player_plot <- renderPlot({ ... })
    output$player_photo <- renderUI({ ... })
  })
}
```

Component	**Purpose and key features**
output$table <- renderDT({ ... })	Displays the filtered data table
output$player_plot <- renderPlot({ ... })	Shows performance trends for the selected player
output$player_photo <- renderUI({ ... })	Dynamically renders player photos

Server Logic

Player Profile System:

- Dynamic photo loading
- Info display

Plot Generation:

- Updates when new player selected (row)
- Y-axis responds to control selection

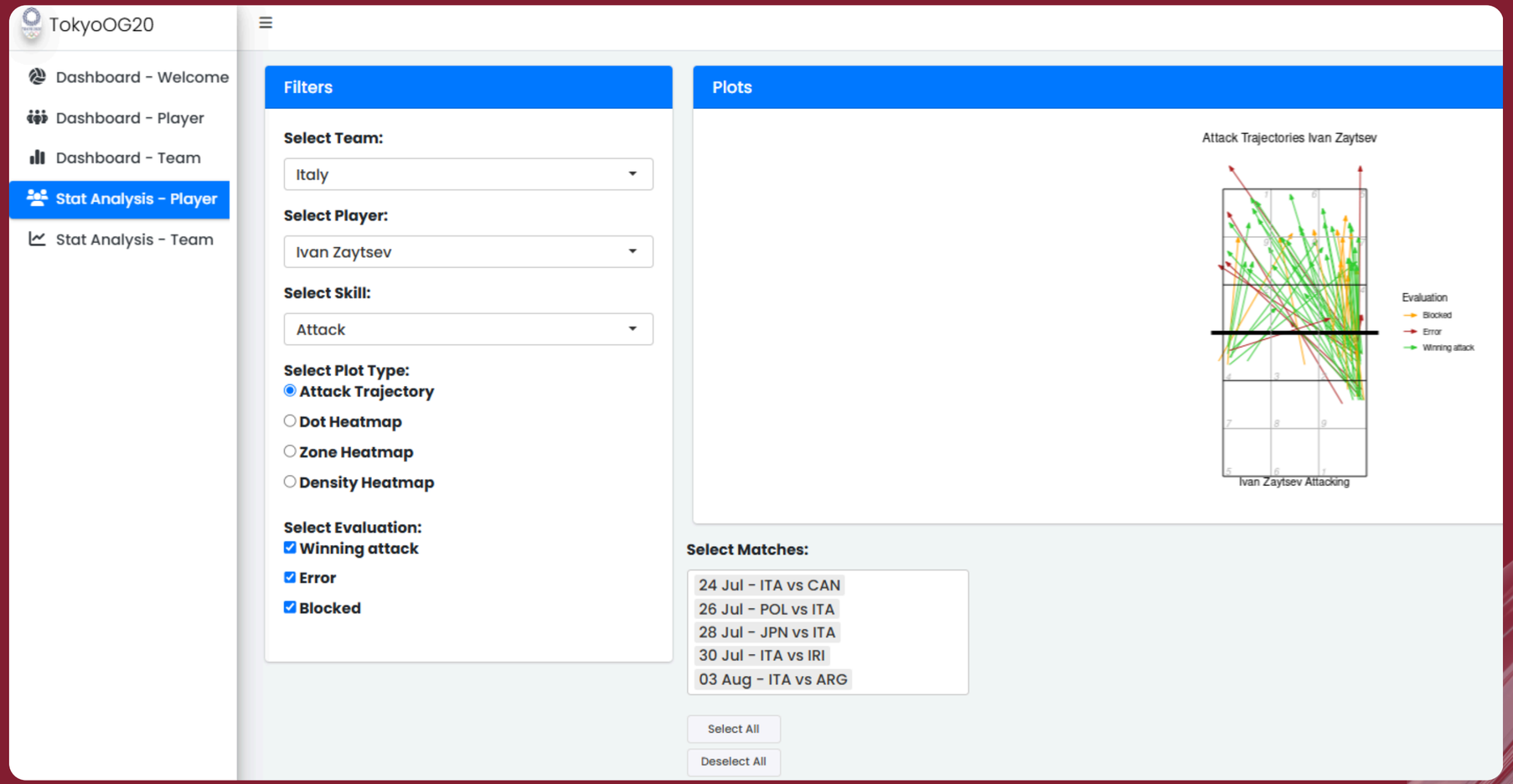
Data Flow Architecture

1. User selects dataset → Updates table variables
2. User filters data → Updates table and plot
3. User selects row → Updates player profile
4. User changes y-axis → Updates plot

Team Dashboard UI and Server function are designed with a focus at team level rather than individual player one. Therefore, the two modules files share similar content.

PLAYER STAT OVERVIEW

This module allows for an interactive analysis of player performance across different volleyball skills (Attack, Serve, Reception, Block, Dig). We provide possibility to generate interactive heatmaps, trajectory plots, and density maps for tactical analysis, based on user selections.



Dynamic UI
Filters update based on user selections (team, player, skill, and matches).

Multiple Plot Types
Zone heatmaps, dot plots, density maps, and trajectory plots.

Interactive Controls:
Conditional panels for skill-specific options.

PLAYER STAT UI - CODE SNIPPETS

```r
statplayerUI <- function(id) {
  ns <- NS(id)  # Namespace function
  tagList(
    fluidRow(
      # Column for filters
      column(
        width = 3,
        box(
          title = "Filters",
          selectInput(ns("team"), "Select Team:", choices = NULL),
          selectInput(ns("player"), "Select Player:", choices = NULL),
          selectInput(ns("skill"), "Select Skill:",
                      choices = c("Attack", "Serve", "Reception", "Block",
```

```r
# Serve Panels
conditionalPanel(
  condition = paste0("input['", ns("skill"), "'] == 'Serve'"),
  radioButtons(ns("plot_type_serve"), "Select Plot Type:",
               choices = c("Serve Trajectory", "Dot Heatmap",
                           "Zone Heatmap", "Density Heatmap")),

  conditionalPanel(
    condition = paste0("input['", ns("plot_type_serve"), "'] == 'Serve Trajectory'"),
    checkboxGroupInput(ns("evaluation_trajectory_serve"), "Select Evaluation:",
                       choices = c("serve_points", "serve_errors",
                                   "serve_bad", "serve_positive"),
                       selected = c("serve_points", "serve_errors",
                                    "serve_bad", "serve_positive"))
  ),
```

UI Components

Team Dropdown:

- selectInput("team"): Dynamically populated with unique teams from the dataset.

Player Dropdown:

- selectInput("player"): Updates based on the selected team.

Skill Dropdown:

- selectInput("skill"): Fixed choices (Attack, Serve, etc.). Triggers conditional panels for skill-specific options.

Skill-Specific Controls:

- Example: For "Serve," users can choose between "Serve Trajectory," "Dot Heatmap," etc.
- Each plot type has further inputs (e.g., evaluation metrics for trajectories or heatmaps).

PLAYER STAT UI - CODE SNIPPETS

```r
fluidRow(
  # Add Match Selection (Dropdown)
  div(
    selectizeInput(ns("matches"), "Select Matches:", choices = NULL, mu
    style = "display: flex; flex-direction: column; align-items: flex-s
    actionButton(ns("select_all"), "Select All", style = "width: 100px
    actionButton(ns("deselect_all"), "Deselect All", style = "width: 1(
  )
)
```

```r
conditionalPanel(
  condition = paste0("input['", ns("skill"), "'] == 'Attack'"),
  plotOutput(ns("attackPlot"))
),
conditionalPanel(
  condition = paste0("input['", ns("skill"), "'] == 'Serve'"),
  plotOutput(ns("servePlot"))
),
conditionalPanel(
  condition = paste0("input['", ns("skill"), "'] == 'Reception'"),
  plotOutput(ns("receptionPlot"))
),
conditionalPanel(
  condition = paste0("input['", ns("skill"), "'] == 'Block'"),
  plotOutput(ns("blockPlot"))
)
```

UI Components

Match Selection:

- selectizeInput("matches"): Dynamically populated with matches involving the selected player.
- Includes "Select All/Deselect All" buttons for convenience.

Visualization Output:

- plotOutput() for each skill (e.g., attackPlot, servePlot), rendered conditionally based on the selected skill

Conditional UI:

Panels for plot types and evaluations appear only when the relevant skill is selected.

- Example: "Serve Trajectory" options only show for skill == "Serve”

PLAYER STAT SERVER - CODE SNIPPETS

```r
# Module Server Function
statplayerServer <- function(id, data) {

  moduleServer(id, function(input, output, session) {
    ns <- session$ns

    # 1. Update team choices dynamically
    observe({
      teams <- unique(data$team)
      updateSelectInput(session, "team", choices = teams)
    })
```

```r
    # 2. Observe team selection -> update players
    observeEvent(input$team, {
      players <- unique(data$player_name[data$team == input$team])
      updateSelectInput(session, "player", choices = players)
    })
```

```r
# 3. Observe player selection -> update matches
observeEvent(input$player, {
  req(input$player)

  matches <- unique(data$file_name[data$player_name == input$player])
```

Server Logic

Dynamic Data Updates (Team/Player/Match):

- updates teams → observeEvent(input$team)
- updates players → observeEvent(input$player)
- updates matches → observeEvent(input$player)
- req() to ensure dependecies are met before execution

UI-Server Workflow:

- User selects a team
- Server updates player list
- User selects player
- Server updates matches
- User selects matches/skill
- Server generates plots.

PLAYER STAT SERVER - CODE SNIPPETS

```r
filtered_data <- reactive({
  req(input$player, input$skill, input$matches)
  data %>% filter(player_name == input$player,
                  skill == input$skill,
                  file_name %in% input$matches)
})

zone_attack <- reactive({
  if (input$skill == "Attack" && input$plot_type_attack == "Zone Heatmap") {
    filtered_data() %>%
      group_by(start_zone) %>%
      summarize(attack_points = sum(evaluation_code == '#'), ...)
  }
})
```

```r
output$attackPlot <- renderPlot({
  if (input$plot_type_attack == "Zone Heatmap") {
    ggplot(zone_attack(), aes(x, y, fill = attack_efficiency)) +
      geom_tile() + ggcourt(...)
  } else if (input$plot_type_attack == "Dot Heatmap") {
    ggplot(heatmap_attack(), aes(end_coordinate_x, end_coordinate_y,
                                 color = evaluation)) +
      geom_point() + ggcourt(...)
  }
})
```

SERVER Logic

Reactive Data Processing:

- filtered_data(): Filters data based on player, skill, and selected matches.
- Generating skill-specific reactive dataframes
  - Example: zone_attack(), heatmap_serve()
  - Calculate metrics (e.g., efficiency, percentages) and transform coordinates for plotting.

Plot Rendering:

- Conditional Rendering:
  - Example: For "Attack," checks input$plot_type_attack to render Zone/Dot/Density plots or trajectories.
- ggplot2 with ggcourt for volleyball court overlays.
- Dynamic Labels:
  - Titles and labels include the player's name and selected evaluation metric.

Team Stat UI and Server function are designed with a focus at team level rather than individual player one. Therefore, the two modules files share similar content.

BEFORE MOVING TO SHOWCASE

USER INTERFACE & INTERACTIVITY ENHANCEMENT

- In our wep app, significant effort was made to improve the user interface and user experience through both visual design and usability optimizations (such as custom icons, custom CSS, HTML tags, bs4Dash etc.).
- Loading states were made more intuitive using the withSpinner() wrapper, which displays loading animations while plots or tables are being generated, improving perceived performance.
- Conditional UI rendering with conditionalPanel() to show or hide elements based on user choices (e.g., plot type or skill).

MAIN CHALLENGES

- Deploy process (non-CRAN libraries): solved using renv package.
- Modularization process.
- Preparing the data for the plots (e.g., Zone Plots vs Heatmap) + spatial/coordinate data.
- Project coordination and time availability.

FUTURE IMPROVEMENTS

- Higher level of overall generalization.
- Match comparison Module.
- Graphics improvement of the web app.
- Integrating data from different competitions.

THANK YOU

https://zhuohao13.shinyapps.io/20630ShinyApp_TokyoOG20/