Fairtree Code Guide

Menzi Njakazi

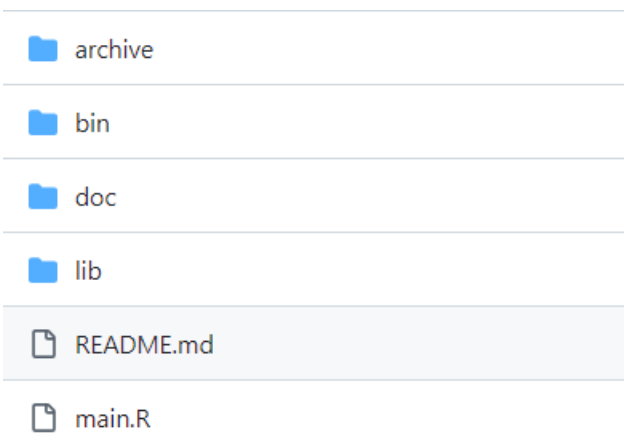
This document outlines the coding standards and best practices to be followed by the team.

Project Structure

There are two types of project structures to be followed one for the standard R Project and the other for Shiny applications.

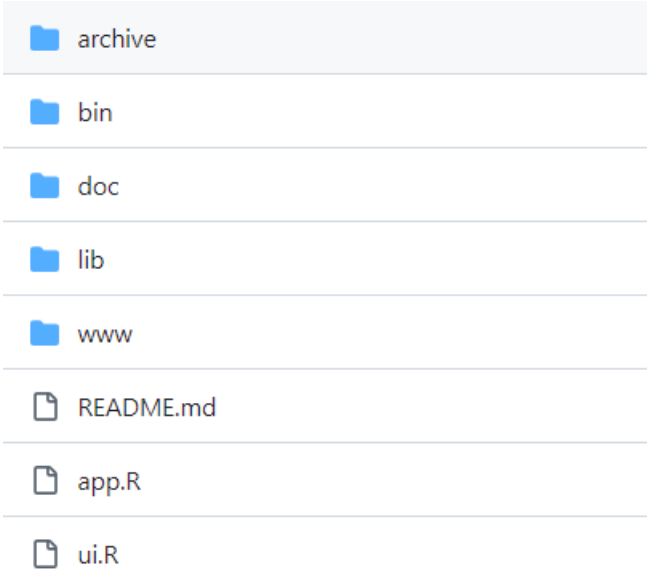
1. Standard R Project

- **archive:** Store old but still relevant documents/scripts.
- **bin:** Directory for the project's output files.
- **doc:** Contains documentation related to the project.
- **lib:** Holds function scripts.
- **main.R:** The main entry point for the project.
- **README.md:** Includes project details, dependencies, and instructions on how to run the project.



2. Shiny Project

- **archive:** Store old but still relevant documents/scripts.
- **bin:** Directory for the project's output files.
- **doc:** Contains documentation related to the project.
- **lib:** Holds function scripts.
- **www:** Stores assets like images, icons, and stylesheets.
- **ui.R:** Defines the user interface of the application.
- **app.R:** Contains the server logic or main application code.
- **README.md:** Includes project details, dependencies, and instructions on how to run the application.



Variables

1. **Use snake_casing for variable names**
   - Improves readability by using underscores to separate words.

```
number_of_days <- 30  # Snake case for variable names
```

2. **Use meaningful names**
   - Variable names should clearly indicate their purpose. Avoid vague names like temp or var.

3. **Avoid overwriting variables**
   - For debugging and code clarity, create new variables instead of overwriting existing ones.

```
# Instead of overwriting, use new variable names
final_price <- 500
discounted_price <- final_price * 0.9
```

4. **Avoid reseverd names and keywords**
   - Do not use names that conflict with R's base functions or keywords (e.g., sum, mean, data).

Functions

Follow the DRY principle: "Don't Repeat Yourself."

1. Use PascalCase for function names

- Function names should follow pascal casing to distinguish them from variables.

```r
# Pascal case function name
CalculateMean <- function(numbers) {
  return(mean(numbers))
}
```

2. Functions should do precisely one thing.

- Each function should perform a single, well-defined task.

```r
# Single responsibility functions
CalculateMean <- function(numbers) {
  return(mean(numbers))
}

# Separate function for standard deviation
CalculateSD <- function(numbers) {
  return(sd(numbers))
}
```

3. Pass all parameters explicitly

- Functions should depend only on their arguments, not external variables.

```r
# Define the function
CalculateDiscount <- function(price, discount_rate) {
  discounted_price <- price * (1 - discount_rate)
  return(discounted_price)
}

# Explicitly pass all arguments when calling the function
price <- 100  # Example price
discount_rate <- 0.2  # Example discount rate

# Call the function with the explicitly provided arguments
final_price <- CalculateDiscount(price = price, discount_rate = discount_rate)
```

4. Keep Functions concise

- Functions should ideally be 10–20 lines. Break longer functions into smaller, manageable ones.

```r
CleanData <- function(data) {
  # Data cleaning steps
}

SummarizeData <- function(data) {
  # Data summarization steps
}

# Example of breaking a long function into smaller ones
```

```r
ProcessData <- function(data) {
  cleaned_data <- CleanData(data)
  summarized_data <- SummarizeData(cleaned_data)
  return(summarized_data)
}
```

5. Document your functions

- All functions should include comments explaining the function and use.
- Use comments and the roxygen2 package to document functions..

```r
#' Calculate the average of a numeric vector
#' @param numbers A numeric vector
#' @return The mean of the numbers
CalculateMean <- function(numbers) {
  return(mean(numbers))
}
```

6. Source Functions at the top of the script

- Always source your functions at the top of the script for better organization.

```r
# Source functions at the top of the script
source("lib/calculate_mean.R")
source("lib/clean_data.R")
```

Refactoring Tips:

- Refactoring improves existing code without changing its behavior. First, write working code, then refine it.
- Make the code more readable and cleaner whenever you modify it.
- **Remove dead code:** Eliminate unused functions and commented-out code.
- Avoid leaving .x or .y suffix when joining data frames.

Remember to **Commit regularly** with descriptive commit messages.