Dashboards com R — Módulo II

O pacote Golem

Agosto de 2022

Motivação

O destino final de aplicativos Shiny costuma ser um ambiente de produção diferente do ambiente de desenvolvimento. Seja um servidor próprio, uma máquina na nuvem ou shinyapps.io, o nosso app precisa funcionar nesses ambientes, e não apenas na nossa máquina.

Uma vez no ambiente de produção, aplicativos Shiny costumam ficar lá por um bom tempo, gerando a necessidade de manutenção e/ou atualizações periódicas.

A depender de como o app foi desenvolvido, essas tarefas podem ficar muito mais trabalhosas. Nesse sentido, seria interessante ter um framework de desenvolvimento que facilitasse a organização e documentação do código e o controle das dependências.

Solução

O `golem` é um *framework* para desenvolvimento de aplicativos Shiny prontos para serem colocados em produção.

Vantagens

- Padroniza a organização dos códigos
- Integra com pacotes que aceleram o desenvolvimento
- Motiva e facilita a documentação do código
- Facilita o compartilhamento e a reutilização de códigos em outros projetos e com outras pessoas

Premissas do `golem`

- Um aplicativo `golem` é construído como um pacote R.
- Divida seu aplicativo em módulos!

Antes de continuar falando do `golem`, vamos fazer uma breve introdução ao desenvolvimentos de pacotes em R.

Pacotes

Um pacote do R é uma forma específica de organizar seus código, seguindo o protocolo descrito pela R Foundation:

> *Pacotes são a unidade fundamental de código R reprodutível.*
>
> — Wickham & Bryan

- Pacotes incluem:
  - Funções em R
  - Documentação sobre como usá-las
  - Testes
  - Dados de exemplo

Criando um pacote

Para criar um pacote, usamos a função `usethis::create_package()`

- Você deve passar um caminho como `~/Documents/meupacote` e uma nova pasta chamada `meupacote` será criada dentro da pasta `Documents`. Essa pasta será tanto um `Rproj` quanto um pacote, ambos chamados `meupacote`
- **Dica geral:** não adicione acentos, caracteres especiais e espaços no nome dos caminhos, arquivos, funções, pacotes, etc.

```
# Saída no próximo slide
usethis::create_package("~/Documents/meupacote")
```

Estrutura básica do pacote

Essa é a estrutura criada quando usamos a função `usethis::create_package()`:

- `meupacote.Rproj`: este arquivo faz com que este diretório seja um projeto no RStudio (RStudio Project)
- `LICENSE`: especifica os termos de uso e distribuição do seu pacote
- `DESCRIPTION`: define o nome, descrição, versão, licença, dependências e outras caracaterísticas do seu pacote
- `R/`: aqui ficam as funções desenvolvidas em R
- `.Rbuildignore`: Lista arquivos que não devem ser incluídos ao compilar o pacote R a partir do código-fonte
- `NAMESPACE`: Não devemos editar este arquivo manualmente. Ele declara as funções que o pacote exporta e as funções que seu pacote importa de outros pacotes

A pasta R/

Dentro de um pacote, a pasta `R/` só pode ter arquivos R com funções.

- Uma função é responsável por executar uma tarefa pequena, mas muito bem.
- A ideia da pasta `R/` é guardar em um local comum tudo aquilo que nós utilizamos como ferramenta interna para nossas análises, bem como aquilo que queremos que outras pessoas possam usar no futuro.
- Podemos usar `usethis::use_r("nome-do-arquivo")` para que um arquivo seja criado antes começarmos a escrever uma função.
- Assim que escrevermos/modificarmos alguma função, podemos carregá-las e testá-las manualmente com `devtools::load_all()`

Criando a sua própria função

Quando estamos desenvolvendo pacotes, iremos criar funções para executar as tarefas necessárias:

```
nome_da_funcao <- function(argumento_1, argumento_2 = valor_padrao_2) {
  # Código que a função irá executar
}
```

Um exemplo: vamos criar uma função que soma dois números.

```
minha_soma <- function(x, y) { # minha_soma é uma função de x e y
  soma <- x + y # soma é igual a x + y
  soma # resultado retornado
}

minha_soma(2, 2)
```

```
#> [1] 4
```

Se quisermos dar valores padrões para os argumentos, basta colocar um = na definição da função.

Dependências

Sem os inúmeros pacotes criados pela comunidade, o R provavelmente já estaria no porão da Ciência de Dados. Por isso, é a primeira coisa que escrevemos nos nossos *scripts* quase sempre é `library(algumPacoteLegal)`.

Quando lidamos com pacotes, a função `library()` não pode ser utilizada, e todas as funções devem ter seus pacotes de origem explicitamente referenciados pelo operador `::`.

- O código, no total, executa um pouco mais rápido porque são carregadas menos funções no ambiente global (isso é especialmente importante em aplicações interativas feitas em Shiny).
- As dependências do código estão sempre atualizadas porque elas estão diretamente atreladas às próprias funções sendo utilizadas.
    - `usethis::use_package()`: adiciona pacotes que foram instalados via CRAN
    - `usethis::use_dev_package()`: adiciona pacotes que não foram instalados via CRAN
- Para escrever `dplyr::`, por exemplo, basta digitar `d`, `p`, `l` e apertar TAB uma vez. Com os `::`, as sugestões passarão a ser somente de funções daquele pacote.

Recomendações

Algumas recomendações sobre como organizar seu código:

- Evite usar `.` no nome das suas funções (hoje em dia usar _ é muito mais comum)
- Use nomes descritivos para as funções, pois isso facilita a manutenção e o uso do pacote
- Tente se limitar a 80 caracteres por linha porque isso permite que seu código caiba confortavelmente em qualquer tela
- Não use `library()` ou `require()`, pois isso vai causar problemas (use a notação `pacote::função()`)
- **Nunca** use `source()`, todo o código já será carregado automaticamente com `devtools::load_all()`
- Não usar "metapackages" (como o tidyverse)

Dados

Se você quiser inserir dados ao seu pacote, você pode utilizar a função `usethis::use_data()`.

Ela criará uma pasta `data/` na raiz do seu pacote, caso ela não exista ainda, e salvará nela o objeto `meus_dados` em formato `.rda`.

Arquivos `.rda` são extremamente estáveis, compactos e podem ser carregados rapidamente pelo R, tornando este formato o principal meio de guardar dados de um pacote.

Manipulando dados crus

Se a base que você quiser colocar no pacote for o resultado de um processo de manipulação de uma base crua, você pode salvar a base crua e o código desse processo na pasta `data-raw`.

Para isso, utilize a função `usethis::use_data_raw("meus_dados")`. Ela criará uma pasta `data-raw/` na raiz do seu pacote, caso ela não exista ainda, e um arquivo `meus_dados.R` onde você colocará o código de manipulação da base crua.

Qual a diferença entre R/ e data-raw/?

`data-raw`

- A pasta `data-raw/` é sua caixa de areia
- Apesar de existirem formas razoáveis de organizar seus pacotes aqui, nessa parte você será livre

`R/`

- Já a pasta `R/` conterá funções bem organizadas e documentadas
- Por exemplo, uma função que ajusta um modelo estatístico, outra que arruma um texto de um jeito patronizado, ou uma que contém seu tema customizado do `{ggplot2}`
- Dentro dessa pasta você não deve carregar outros pacotes com `library()`, mas sim usar o operador `::`

Voltando ao `golem`

Para instalar o pacote direto do CRAN, rode o código abaixo:

```
install.packages("golem")
```

Já a versão de desenvolvimento pode ser instalada com o código a seguir:

```
remotes::install_github("Thinkr-open/golem")
```

Para criar um aplicativo golem, utilizamos a função `golem::create_golem()`. Essa função cria um projeto do RStudio (uma pasta no computador) com a estrutura de um pacote de R e arquivos auxiliares que facilitam o início do desenvolvimento do app.

```
golem::create_golem(path = "caminho/para/o/pacote")
```

Estrutura

A pasta criada terá a seguinte estrutura:

```
#> ├── DESCRIPTION
#> ├── NAMESPACE
#> ├── R
#> │   ├── app_config.R
#> │   ├── app_server.R
#> │   ├── app_ui.R
#> │   └── run_app.R
#> ├── dev
#> │   ├── 01_start.R
#> │   ├── 02_dev.R
#> │   ├── 03_deploy.R
#> │   └── run_dev.R
#> ├── inst
#> │   ├── app
#> │   │   └── www
#> │   │       └── favicon.ico
#> │   └── golem-config.yml
#> └── man
#>     └── run_app.Rd
```

Estrutura

- Arquivos `DESCRIPTION` e `NAMESPACE`: metadados do pacote.
- `R/app_config.R`: usado para ler o arquivo de configuração do `golem` localizado em `inst/golem-config.yml`.
- `R/app_server.R` e `R/app_ui.R`: arquivos onde vamos desenvolver a UI e o servidor do app.
- `R/run_app.R`: função para configurar e rodar o app.
- `dev/`: scripts do `golem` que podem ser utilizados ao longo do desenvolvimento do app. Eles contêm uma lista de funções úteis que ajudam a configurar diversos aspectos do aplicativo.
- `inst/app/www`: pasta onde adicionaremos os recursos externos do aplicativo (imagens, arquivos CSS, fontes etc).
- `man`: documentação do pacote, a ser gerado pelo R e pelo `roxygen2`.

Exercício

Aplique o framework Golem no app disponibilizado neste link. Ele utiliza a base de dados Pokemon, que pode ser baixada clicando aqui.

- Transforme o shinydashboard em Bs4Dash: construa ao menos a UI do zero.
- Modularize o app (cada página do dashboard deve ser um módulo diferente).
- Refaça os gráficos utilizando alguma biblioteca javascript (plotly, echarts, highcharts etc)
- Faça o deploy do app para o shinyapps.io

Referências

- Zen do R, livro em desenvolvimento pela Curso-R.
- R Packages, livro aprofundado sobre desenvolvimento de pacotes.
- R for Data Science - capítulo sobre Funcões
- Tutorial do `golem`
- Engineering Production-Grade Shiny Apps
- rhino: alternativa ao golem.

Leitura extra

Documentação de funções

Se quisermos adicionar documentação ao nosso pacote (as instruções que aparecem quando vamos usar uma função ou o documento mostrado quando rodamos `?função()`) precisamos usar um comentário especial: `#'`

```
#' Título da função
#'
#' Descrição da função
#'
#' @param a primeiro parâmetro
#' @param b segundo parâmetro
#'
#' @return descrição do resultado
#'
#' @export
fun <- function(a, b) {
  a + b
}
```

Documentação de funções

- O parâmetro `@export` indica que a função ficará disponível quando rodarmos `library(meupacote)`. Não se esqueça de exportar todas (e somente) as funções públicas!
- O RStudio disponibiliza um atalho para criar a estrutura da documentação de uma função. No menu superior, clique em `Code` -> `Insert Roxygen Skeleton`.
- Para deixar a documentação das suas funções acessível (no help do R), use a função `devtools::document()` (**CTRL + SHIFT + D**).
- Ao executar `devtools::check()`, a documentação já é atualizada e disponibilizada de brinde

Documentação de bases de dados

```
#' Título da base
#'
#' Descrição da base
#'
#' @format Uma lista que descreve as colunas:
#' \describe{
#'   \item{col1}{Descrição da coluna 1}
#'   \item{col2}{Descrição da coluna 2}
#'   ...
#' }
#' @source Origem dos dados
"base"
```

- `@format` descreve o formato da base (número de colunas, linhas, etc.) e pode conter uma lista que explica o significado de cada coluna
- `@source` é a fonte, muitas vezes um `\url{}`
- **Nunca** coloque `@export` em uma base de dados

Sobre licenças de código aberto

- O pacote `{usethis}` possui algumas funcões para nos ajudar a declarar uma licenças como, por exemplo:

```
usethis::use_cc0_license()
```

- CC0: essa licença, muitas vezes chamada de "sem direitos reservados", permite que o trabalho seja colocado em domínio público. Qualquer pessoa pode usar, modificar, distribuir e vender o seu trabalho sem nenhuma restrição de direitos autorais

```
usethis::use_mit_license()
```

- MIT: curta e permissiva, a licença MIT exige apenas manutenção dos direitos autorais. Modificações e trabalhos maiores podem ser distribuídos sob outros termos de uso

```
usethis::use_gpl3_license()
```

- GPLv3: essa licença exige que o código-fonte dos derivados seja *open source* sob a mesma licença. Direitos autorais devem ser preservados.

- O help é útil: `?usethis::use_mit_license`, e também o site Choose a License.

Acentos, encoding e variáveis globais

Prefira manter os arquivos em inglês para que seu pacote possa ser submetido ao CRAN.

- Se quiser fazer um pacote com documentação em português, tente escrever sem acentos ou escapar strings (veja `abjutils::escape_unicode()`). O `devtools::check()` vai te alertar caso essa regra seja violada

O *encoding* (codificação) dos arquivos deve ser **sempre** UTF-8 para evitar problemas entre plataformas.

- Se tiver problemas com isso, tente **File > Reopen with Encoding...**, ou **File > Save with Encoding...**, ou **Tools > Project Options... > Code Editing > Text encoding**

Variáveis globais são normalmente uma má prática em código R, então a `devtools::check()` vai reclamar se encontrar algo do tipo; o problema é que as colunas modificadas em funções do `{dplyr}` são caracterizadas como globais.

- A solução é criar um arquivo com uma linha como a abaixo contendo todas as variáveis que fizerem a `devtools::check()`reclamar

```
utils::globalVariables(c("variavel1", "variavel2"))
```

Boas práticas no desenvolvimento

- Não rode as funções diretamente. Utilize sempre a função `devtools::load_all()`. Ela carrega todas as funções da pasta `R/` e as bases salvas na pasta `data/`. Isso diminuirá a chance de elas estarem sendo afetadas por valores externos que estão no seu *Environment*.
- Limpe o seu *Environment* sempre que possível. Um atalho útil: **CTRL + SHIFT + F10**.
- Para deixar a documentação das suas funções acessível (no help do R), use a função `devtools::document()`.
- Se você precisar instalar o seu pacote (equivalente ao que fazemos com pacotes do CRAN quando rodamos `install.packages()`), use a função `devtools::install()`. Ela deve ser utilizada quando o seu pacote estiver pronto (ou pelo menos alguma versão dele).