Dashboards com R

Reatividade (parte 2)

Junho de 2022

observers

Já vimos que os observers são os pontos finais de um diagrama de reatividade de um shiny app e que, sem eles, a reatividade não é ativada. Também vimos que as funções `render*()`, que geram os nossos outputs, são o tipo mais comum de observer. Mas eles não são os únicos.

Muitas vezes queremos usar a reatividade para disparar ações que não estão ligadas a geração de outputs, como o registro de informações em bases de dados ou a atualização de elementos da UI.

Nesses casos, podemos utilizar as funções `observe()` e `oberveEvent()`. Elas funcionam de maneira similar às funções `reactive()` e `eventReactive()`, mas em vez de criar expressões reativas, elas são observers.

A função `observe()`

A função `observe({codigo})` monitora os valores e expressões reativas que estão dentro dela e roda seu código quando algum desses valores são modificados.

A diferença do `observe()` para a função `reactive()` é que a primeira não gera expressões reativas, servindo apenas para códigos que têm efeitos colaterais.

Essa função é muito utilizada com as funções da família `update*()`, que servem para atualizar valores de um input na UI.

Veja um exemplo no próximo slide.

A função `observe()`

Queremos, na segunda caixa de seleção, selecionar apenas os filmes do(a) diretor(a) que selecionamos na primeira. Veja que usamos o texto `Carregando...` como um *placeholder* para o segundo `selectInput()`.

```
ui <- fluidPage(
  selectInput(
    "dir",
    "Selecione um(a) diretor(a)",
    choices = sort(unique(imdb$diretor))
  ),
  selectInput(
    "filme",
    "Selecione um filme",
    choices = "Carregando..."
  )
)
```

A função `observe()`

No `server`, atualizamos as escolhas da segunda caixa de seleção com a função `updateSelectInput()`. Veja que, como essa função está dentro de um observer, esse código será rodado novamente sempre que o valor de `input$dir` mudar.

```
server <- function(input, output, session) {

  observe({
    opcoes <- imdb |>
      dplyr::filter(diretor == input$dir) |>
      dplyr::pull(titulo)
    updateSelectInput(
      session,
      inputId = "filme",
      choices = opcoes
    )
  })
}
```

A função `observeEvent()`

A função `observeEvent()` funciona assim como a `observe()`, mas ela escuta apenas um valor ou expressão reativa, que é definido em seu primeiro argumento, assim como na função `eventReactive()`.

Ela é muito utiliza para disparar ações, como gravar informações em uma base de dados, a partir de botões.

Veja um exemplo no slide a seguir.

A função `observeEvent()`

Queremos salvar o e-mail de uma pessoa quando ela clicar no botão "Enviar dados". A função `observeEvent()` roda o código definido dentro dela quando o botão é clicado, salvando o e-mail em um arquivo.

```
ui <- fluidPage(
  textInput("email", "Informe seu e-mail"),
  actionButton("enviar", "Enviar dados")
)

server <- function(input, output, session) {

  observeEvent(input$enviar, {
    write(input$email, "emails.txt", append = TRUE)
  })
}
```

Atividade

Vamos treinar a utilização dessas funções.

Ao RStudio: 19-observers.R e Ao RStudio: 20-observeEvent.R

Valores reativos

Também vimos que os valores reativos são o início do diagrama de reatividade e que os valores da lista `input` são o principal tipo de valor reativo em um shiny app.

Em alguns casos, no entanto, vamos precisar de valores reativos que não são inputs, isto é, não estão associados a ações vindas da UI. A lista `input` possui a restrição de não podermos escrever nela, o que nos impede de criar um valor que disparasse o fluxo reativo e que só existisse dentro do server.

Para criar valores reativos com essa característica, podemos usar as funções `reactiveVal()` e `reactiveValues()`. A única diferença entre as duas é que a primeira cria apenas um valor reativo enquanto a segunda permite criar uma lista de valores reativos.

A função `reactiveVal()`

Para criar um valor reativo:

```
valor_reativo <- reactiveVal(1)
```

Para acessar o valor atual:

```
valor_reativo()
#> 1
```

Para atualizar o valor:

```
valor_reativo(2)
```

A função `reactiveValues()`

Para criar um valor reativo:

```
rv <- reactiveValues(a = 1, b = 2)
```

Para acessar o valor atual:

```
rv$a
#> 1

rv$b
#> 2
```

Para atualizar o valor:

```
rv$a <- 3
rv$b <- 4
```

Atividade

Vamos usar valores reativos para simular a atualização de uma base.

Ao RStudio: 21-reactvalues.R

Validação

O pacote `shiny` possui algumas funções que nos ajudam a validar valores reativos antes de rodarmos um código que gera uma visualização (output). Isso impede que mensagens de erros internas do R apareçam para o usuário, envia mensagens e avisos personalizados quando o usuário faz algo que não devia, controla pontas soltas de reatividade e deixa o nosso aplicativo mais eficiente.

- `isTruthy(x)`: teste se x é válido. `FALSE`, `NULL`, `""`, entre outros, são considerados valores inválidos. Veja `help(isTruthy)` para ver a lista de valores considerados inválidos.

- `req(x)`: retorna um erro silencioso caso x seja inválido.

- `validate()`: transforma uma mensagem personalizada em uma mensagem de erro para o usuário. Geralmente utilizada junto da função `need()`.

- `need(teste, mensagem)`: retorna uma mensagem personalizada caso o resultado do teste seja falso.

Função req()

Veja um exemplo de utilização da função `req()`. No código abaixo, a infoBox só será criada se o valor reativo `input$filme` tiver um valor válido (no caso, uma string não nula). Caso o valor seja inválido, a infoBox não será mostrada no app. Nenhuma mensagem de erro ou aviso será retornado ao usuário.

```
#server
ouput$orcamento <- renderInfoBox({

  req(input$filme)

  orcamento <- imdb %>%
    filter(titulo == input$filme) %>%
    pull(orcamento)

  infoBox(
    title = "Orçamento",
    value = orcamento
  )

})
```

Mensagem de erro personalizada

Neste caso, além de o aplicativo não mostrar a infoBox, uma mensagem é enviada ao usuário explicando o porquê. No código, utilizamos `isTruthy(input$filme)` para testar se `input$filme` é válido, retornamos a mensagem `"Nenhum filme selecionado."` caso ele não seja e usamos a função `validate()` para parar a execução e retornar essa mensagem ao usuário.

```
#server
ouput$orcamento <- renderInfoBox({
  validate(
    need(isTruthy(input$filme), message = "Nenhum filme selec
  )
  orcamento <- imdb %>% filter(titulo == input$filme) %>% pull
  infoBox(
    title = "Orçamento",
    value = orcamento
  )
})
```

Exercícios

1) Usando a base da CETESB, construa um shiny app que possua um filtro de poluente que depende de um filtro de estação de monitoramento. Como output, faça uma série temporal com a média diária da concentração do poluente.

2) Usando a base da SSP, construa um shiny app que possua um filtro de delegacia que depende de um filtro de município que depende de um filtro de região. Como output, faça uma série temporal do número de furtos.

3) Faça um app que contenha um formulário de cadastro com os campos "nome", "e-mail", "idade" e "cidade" e um botão de salvar dados que faça o app salvar as informações em uma planilha no computador. O app também deve mostrar a tabela mais atualizada de pessoas cadastradas.

Atividade

Vamos usar essas funções para melhorar a experiência de uso do nosso app.

Ao RStudio: 19-observers.R

Referências e material extra

Tutoriais

- Tutorial de Shiny do Garrett Grolemund
- Tutorial do Shinydashboard

Material avançado

- Mais sobre reatividade
- Mais sobre debug
- Shiny Server