Workshop: Web Scraping em R

C. Lente + F. Corrêa + Curso-R

2018-03-10

Prólogo

O termo mais apropriado para tratar de *web scraping* em portugês é "raspagem web", ou seja, a extração de dados provenientes de uma página online.

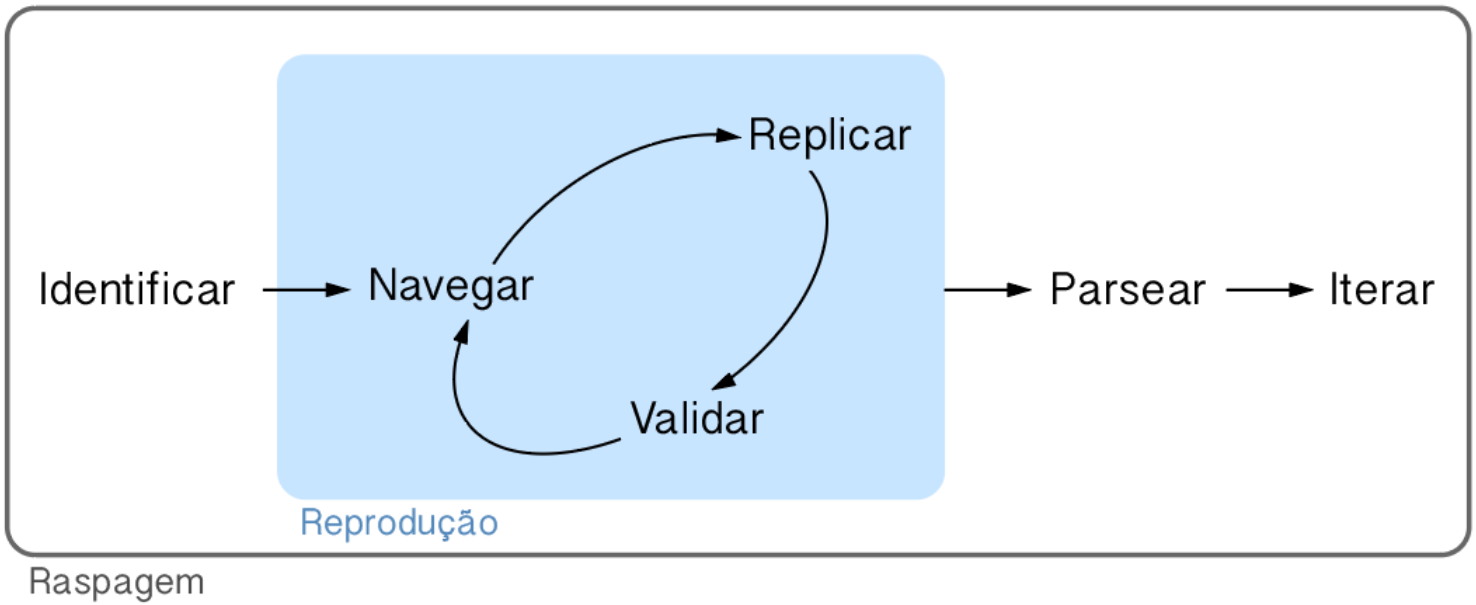
Quando se trata de web scraping, a primeira coisa que precisamos fazer é criar um plano de ação. Raspar dados online não é uma ciência exata, então se não nos planejarmos com antecedência é bem provável que no final nosso código fique completamente incompreensível e irreprodutível.

Expectativa vs. Realidade

```
dados <- raspar("site.com")
```

Prólogo (cont.)

Por isso recomendamos sempre tentar seguir o **fluxo do web scraping**:



Nós achamos que o fluxo do web scraping é a melhor forma de abordar esse conteúdo porque, com ele, a tarefa muitas vezes complicada e sem objetivo definido de raspar dados da web se torna um processo iterativo e bem delimitado que podemos utilizar nas mais diversas situações.

Conteúdo

Através de exposição, exemplos do mundo real e exercícios veremos cada passo do fluxo:

- Como **identificar** o objeto de interesse
- Como funciona uma página HTML e como **navegar** por seus elementos
- O protocolo HTTP e como **replicar** seu funcionamento
- Como **parsear** as páginas baixadas
- Como **iterar** ao longo de muitas páginas e como salvar esses resultados
- Como **validar** resultados e evitar os problemas mais comuns do web scraping
- Outras tecnlogias associadas ao web scraping moderno
- Captchas (quando quebrá-los e como quebrá-los)

IDENTIFICAR

Quando usar/não-usar web scraping

Quando usar:

- Quando precisamos coletar um volume grande de dados da internet

Quando não usar:

- Quando temos uma forma mais simples de obter os dados (API, base de dados, etc.)
- Quando os termos de uso do site não nos permitem fazer isso
- Quando o `robots.txt` do site não nos permite fazer isso
- Quando houver risco de derrubar ou comprometer a estabilidade do site
- Quando as informações do site não são públicas

Encontrar o que você quer

Imagine que você precisa extair alguma informação de um site, seja ela o título de várias páginas da Wikipédia, os comentários de um post do Reddit ou mesmo os dados de países contidos no famosíssimo *Example Web Scraping Website* (http://example.webscraping.com/).

Se tivermos verificado que de fato não temos nenhuma outra opção e que o site nos permite raspá-lo, então podemos começar.

O primeiro passo do fluxo consiste em **identificar** os elementos que queremos extrair e observar se eles se comportam da mesma forma em todas as situações possíveis.

Example Web Scraping Website

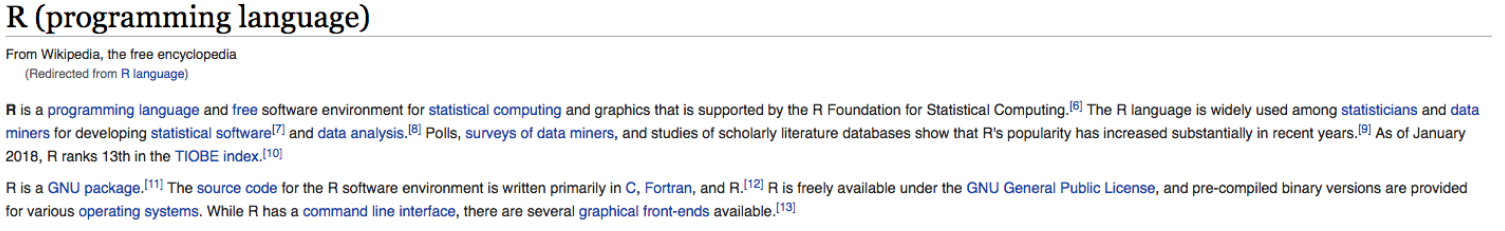
O nosso site de exemplo será o `http://example.webscraping.com`. Para que possamos continuar, precisamos salvar manualmente uma página de interesse desse site (no caso, a home já basta).

Outros casos interessantes

Extração de textos

Talvez uma das tarefas mais simples de se cumprir em web scraping é extrair texto bem-estruturado de uma página. Os artigos da Wikipédia são ótimos para treinar um primeiro scraper, mas eles acabam sendo uma excessão no mundo do web scraping justamente por serem tão bem estruturados.

Além disso, a Wikipédia possui uma API.



R (programming language)

From Wikipedia, the free encyclopedia
(Redirected from R language)

R is a programming language and free software environment for statistical computing and graphics that is supported by the R Foundation for Statistical Computing.[6] The R language is widely used among statisticians and data miners for developing statistical software[7] and data analysis.[8] Polls, surveys of data miners, and studies of scholarly literature databases show that R's popularity has increased substantially in recent years.[9] As of January 2018, R ranks 13th in the TIOBE index.[10]

R is a GNU package.[11] The source code for the R software environment is written primarily in C, Fortran, and R.[12] R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems. While R has a command line interface, there are several graphical front-ends available.[13]

Outros casos interessantes (cont.)

Extação de imagens

Baixar imagens já uma tarefa um pouco mais complexa porque elas geralmente estão menos organizadas que texto e ainda por cima ocupam mais espaço.

A imagem abaixo é do Reddit, que também possui uma API.

Outros casos interessantes (cont.)

Extração de PDFs

Extrair PDFs é um pouco mais complicado do que extrair imagens porque eles geralmente não estão endereçados na página da mesma forma que o resto do conteúdo. Além disso, eles são uma das formas mais pesadas de mídia que podemos raspar.


Instruções Curso de Ciência da Computação e sequenciais
Faculdade Integrada da Grande Fortaleza
Modelo de TCC

1. Introdução

2. Referencial Teórico


Titre du rapport éventuellement en plusieurs lignes


Titre long de la présentation

17 avril 2016

Outros casos interessantes (cont.)

Extação de áudio

Dificilmente temos a necessidade de baixar arquivos de áudio, mas, quando esse é o caso, o fluxo não muda muito em relação aos PDFs. A dificuldade aqui é em como tratar esses arquivos porque geralmente são escassos os métodos e bibliotecas de processamento de som.

Work	Date of performance	Noisy version	Cleaned version
Carmen	1903	▶ 0:00 CC MENU	▶ 0:00 CC MENU

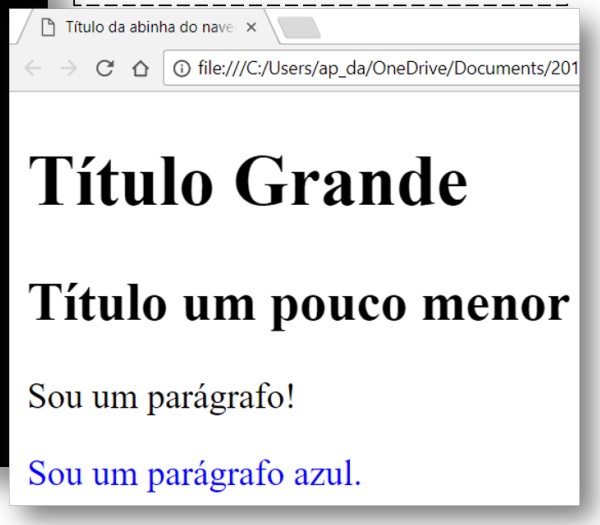
NAVEGAR

Introdução ao HTML

Exemplo.html no editor de texto

```
1  <!DOCTYPE html>
2
3    <head>
4      <meta charset = latin1>
5      <title>Título da abinha do navegador</title>
6    </head>
7
8    <body>
9      <h1>Título Grande</h1>
10
11     <h2>Título um pouco menor</h2>
12
13     <p>Sou um parágrafo!</p>
14
15     <p style='color: blue;'>Sou um parágrafo azul.</p>
16
17   </body>
18 </html>
```

Exemplo.html no navegador



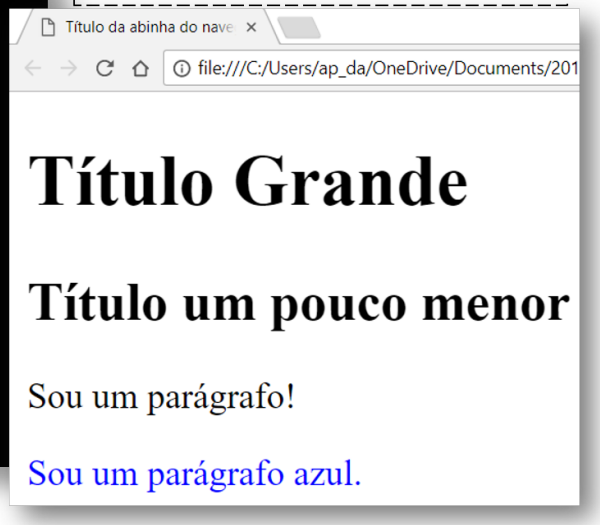
HTML (*Hypertext Markup Language*) é uma linguagem de markup cujo uso é a criação de páginas web. Por trás de todo site há pelo menos um arquivo `.html`.

Introdução ao HTML (cont.)

Exemplo.html no editor de texto

```
1  <!DOCTYPE html>
2
3  <head>
4    <meta charset = latin1>
5    <title>Título da abinha do navegador</title>
6  </head>
7
8  <body>
9    <h1>Título Grande</h1>
10
11   <h2>Título um pouco menor</h2>
12
13   <p>Sou um parágrafo!</p>
14
15   <p style='color: blue;'>Sou um parágrafo azul.</p>
16
17 </body>
18 </html>
```

Exemplo.html no navegador



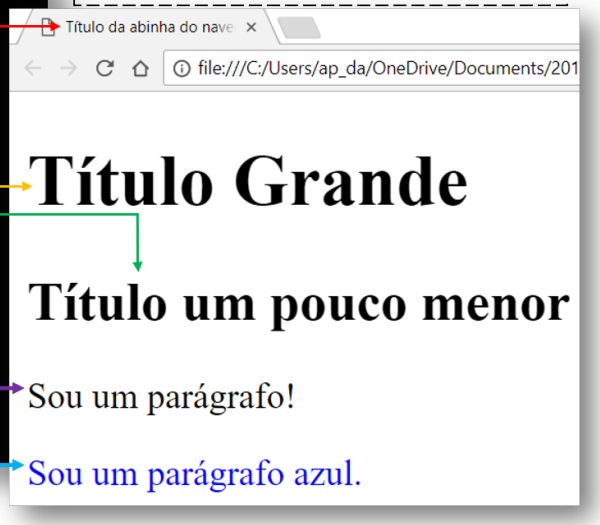
Todo arquivo HTML pode ser dividido em seções que definirão diferentes aspectos da página. `head` descreve "metadados", enquanto `body` é o corpo da página.

Introdução ao HTML (cont.)

Exemplo.html no editor de texto

```
1  <!DOCTYPE html>
2
3    <head>
4      <meta charset = latin1>
5      <title>Título da abinha do navegador</title>
6    </head>
7
8    <body>
9      <h1>Título Grande</h1>
10
11     <h2>Título um pouco menor</h2>
12
13     <p>Sou um parágrafo!</p>
14
15     <p style='color: blue;'>Sou um parágrafo azul.</p>
16
17   </body>
18 </html>
```

Exemplo.html no navegador



Tags (`head`, `body`, `h1`, `p`, ...) demarcam as seções e sub-seções, enquanto atributos (`charset`, `style`, ...) mudam como essas seções são renderizadas pelo navegador.

Introdução ao HTML (cont.)

Um pouco de teoria:

1) Todo HTML se transforma em um **DOM** (document object model) dentro do navegador.

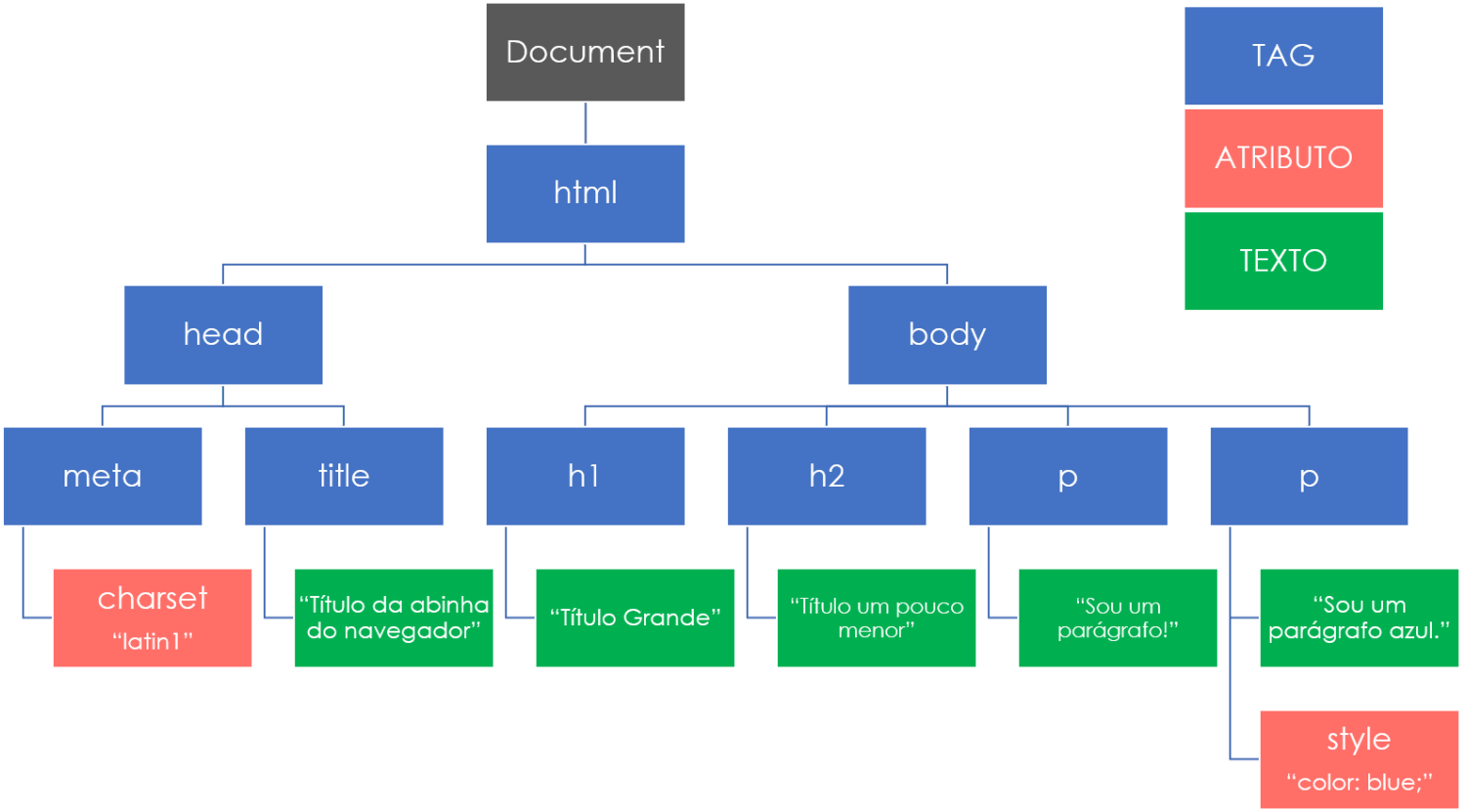
2) Um DOM pode ser representado como uma árvore em que cada *node* é:

- ou um **atributo**
- ou um **texto**
- ou uma **tag**
- ou um **comentário**

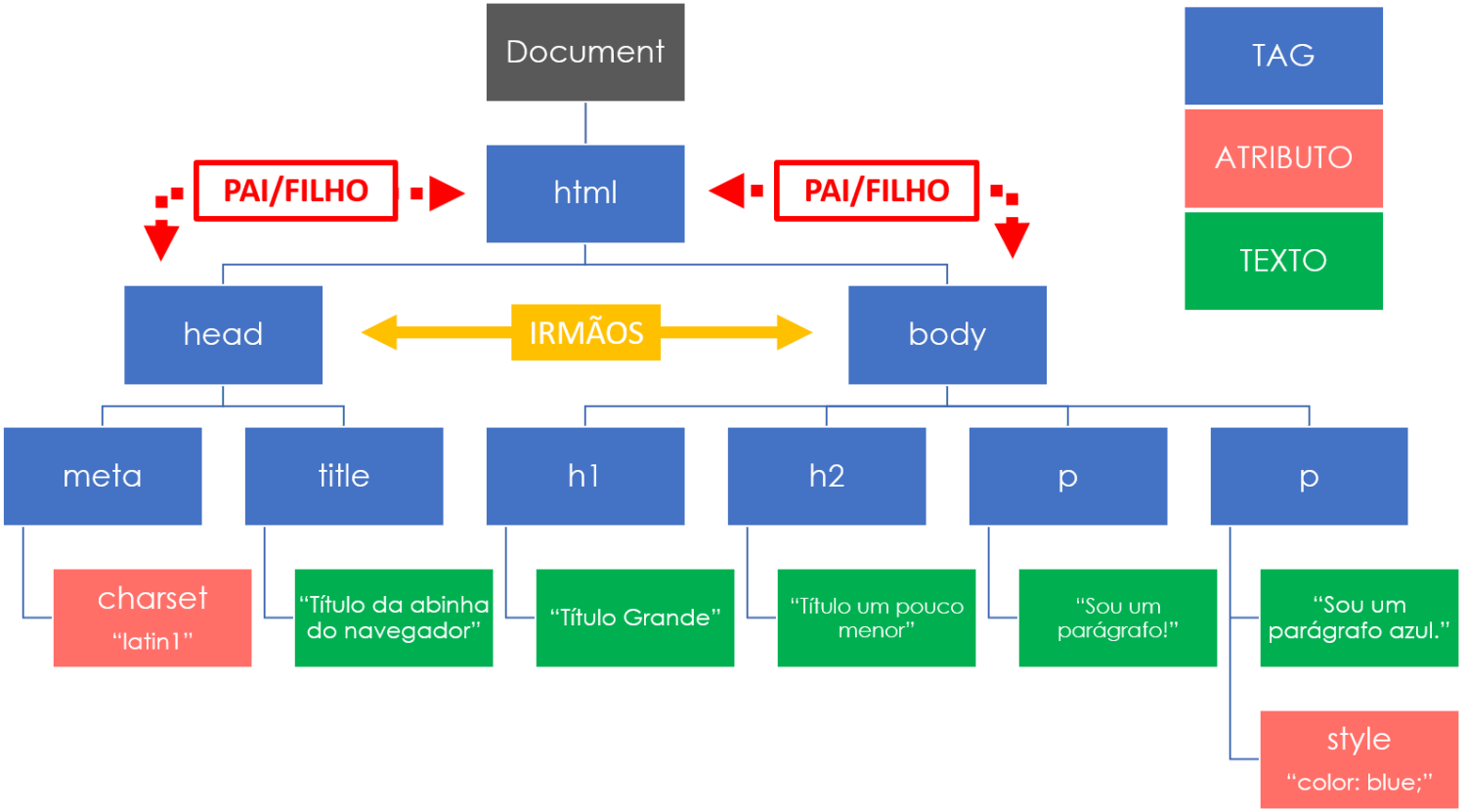
3) Utiliza-se a relação de pai/filho/irmão entre os nós.

4) Para descrever a estrutura de um DOM, usa-se uma linguagem de markup chamada XML (*Extensible Markup Language*) que define regras para a codificação de um documento.

Introdução ao HTML (cont.)



Introdução ao HTML (cont.)



XPath - XML Path Language

Exemplo: coletando todas as tags <p> (parágrafos)

```
library(xml2)

# Ler o HTML
html <- read_html("webscraping/html_exemplo.html")

# Coletar todos os nodes com a tag <p>
nodes <- xml_find_all(html, "//p")
nodes
```

```
#> {xml_nodeset (2)}
#> [1] <p>Sou um parágrafo!</p>
#> [2] <p style="color: blue;">Sou um parágrafo azul.</p>
```

```
# Extrair o texto contido em cada um dos nodes
text <- xml_text(nodes)
text
```

```
#> [1] "Sou um parágrafo!"      "Sou um parágrafo azul."
```

XPath - XML Path Language (cont.)

Com `xml_attrs()` podemos extrair todos os atributos de um node:

```
xml_attrs(nodes)
```

```
#> [[1]]
#> named character(0)
#>
#> [[2]]
#>          style
#> "color: blue;"
```

Já com `xml_children()`, `xml_parents()` e `xml_siblings()` podemos acessar a estrutura de parentesco dos nós:

```
body <- xml_find_all(html, "body")
xml_siblings(body)
```

```
#> {xml_nodeset (1)}
#> [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset= .
```

CSS

CSS (Cascading Style Sheets) descrevem como os elementos HTML devem se apresentar na tela. Ele é responsável pela aparência da página.

```
<p style='color: blue;'>Sou um parágrafo azul.</p>
```

O atributo `style` é uma das maneiras de mexer na aparência utilizando CSS. No exemplo,

- `color` é uma **property** do CSS e
- `blue` é um **value** do CSS.

Para associar esses pares **properties/values** aos elementos de um DOM, existe uma ferramenta chamada **CSS selectors**. Assim como fazemos com XML, podemos usar esses seletores (através do pacote `rvest`) para extrair os nós de uma página HTML.

CSS (cont.)

Abaixo vemos um `.html` e um `.css` que é usado para estilizar o primeiro. Se os nós indicados forem encontrados pelos seletores do CSS, então eles sofrerão as mudanças indicadas.

Seletores CSS vs. XPath

> "OK, legal, podemos usar o CSS e o XPath para encontrar nós em um página, mas por que usar um ou outro"

A grande vantagem do XPath é permitir que acessemos os filhos, pais e irmãos de um nó. De fato os seletores CSS são mais simples, mas eles também são mais limitados.

O bom é que se tivermos os seletores CSS, podemos transformá-los sem muita dificuldade em um query XPath:

- Seletor de tag: `p` = `//p`
- Seletor de classe: `.azul` = `//*[@class='azul']`
- Seletor de id: `#meu-p-favorito` = `//*[@id='meu-p-favorito']`

Além disso, a maior parte das ferramentas que utilizaremos ao longo do processo trabalham preferencialmente com XPath.

Seletores CSS vs. XPath (cont.)

```
html <- read_html("webscraping/html_exemplo_css_a_parte.html")
xml_find_all(html, "//p")
```

```
#> {xml_nodeset (3)}
#> [1] <p>Sou um parágrafo normal.</p>
#> [2] <p class="azul">Sou um parágrafo azul.</p>
#> [3] <p id="meu-p-favorito" class="azul">Sou um parágrafo azul e negrito. .
```

```
xml_find_all(html, "//*[@class='azul']")
```

```
#> {xml_nodeset (2)}
#> [1] <p class="azul">Sou um parágrafo azul.</p>
#> [2] <p id="meu-p-favorito" class="azul">Sou um parágrafo azul e negrito. .
```

Note que `//p` indica que estamos fazendo uma busca na tag `p`, enquanto `//*` indica que estamos fazendo uma busca em qualquer tag.

Encontrando o XPath

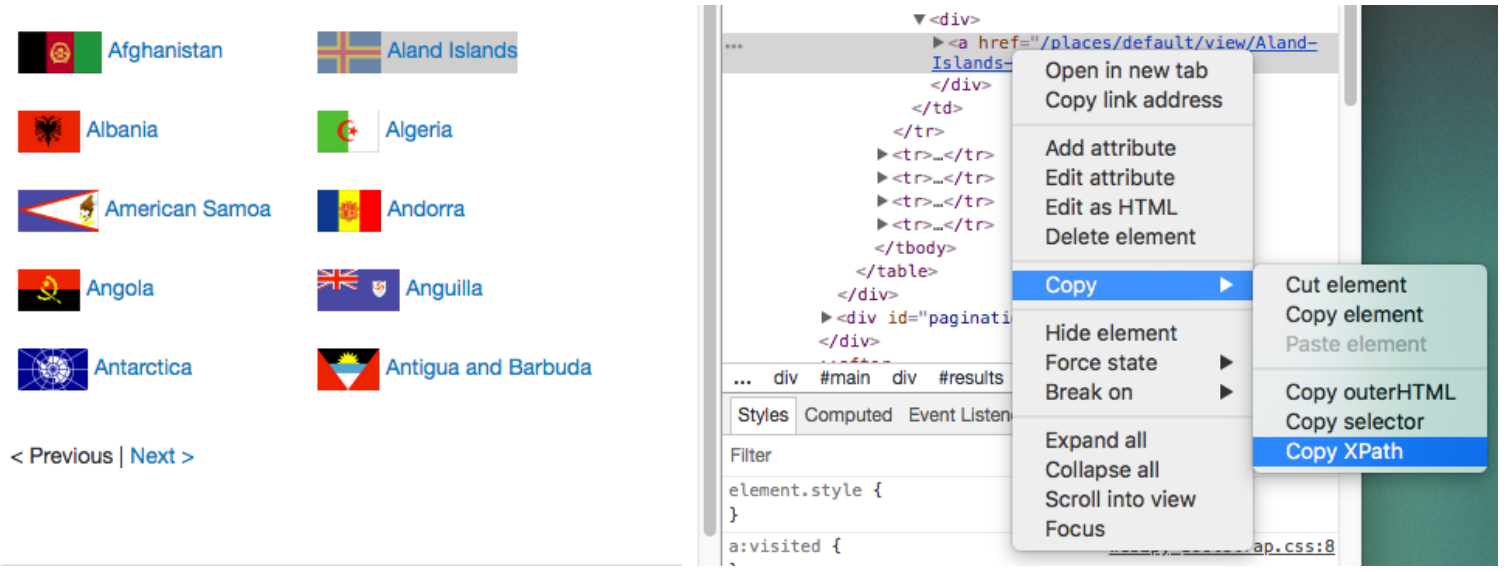
Para os iniciantes no mundo do web scraping, existem ferramentas muito convenientes que nos ajudam a encontrar o XPath (ou CSS Selector) de uma estrutura da página.

A primeira delas se chama **CSS Selector Gadget**, enquanto a segunda está embutida no próprio Chrome.

Vamos ao Chrome...

Inspect

Como acabamos de ver, a ferramenta de inspeção do Chrome (ou de qualquer outro navegador moderno) é nossa grande aliada na hora de encontrar os elementos que desejamos extrair quando raspando uma página.



REPLICAR

Web

Até agora já aprendemos que um site não passa de uma coleção de arquivos que os navegadores são capazes de renderizar. Entendemos também como é a arquitetura básica desses arquivos e como podemos extrair informações destas estruturas.

Agora nos resta entender como funciona a rede que nos traz tais arquivos. Veremos que não é necessário ter um navegador para acessar um site, característica da qual tiraremos vantagem para poder coletar todas as páginas das quais precisamos.

Protocolo HTTP

O HTTP (Hypertext Transfer Protocol) é um protocolo de aplicações para sistemas de informação distribuídos e colaborativos. Este protocolo é, no fundo, a base de toda a web* pois ele permite a troca e transferência de arquivos hipertexto.

Métodos HTTP

O protocolo HTTP na verdade é composto por diversos **métodos** que chamam páginas páginas hipertexto de servidores web. Os métodos mais comuns são `GET` e `POST`

Pacote httr

Em R, o pacote `httr` é o que faz a ponte de comunicação entre nossas requisições e as respectivas respostas.

- Serve para mexer com requisições/respostas HTTP e URLs.
- As principais funções são os métodos do protocolo `http`: GET, POST, content, cookies, write_disk, status_code.

GET

O método `GET` é o mais simples. Ele permite chamar uma página da internet e especificar todos os parâmetros de seu URL.

```r
library(httr)

# GET sem nenhum parâmetro
get_1 <- GET("https://httpbin.org/get")

# GET equivalente a https://httpbin.org/cookies/set?meu-cookie=1
params <- list("meu-cookie" = 1)
get_2 <- GET("https://httpbin.org/cookies/set", query = params)
```

Métodos HTTP (cont.)

POST

O método `POST` é muito semelhante ao `GET`, mas ele permite enviar pacotes carregados de informações para a página (ao invés de apenas adicionar parâmetros à chamada).

```
library(httr)

# POST sem nenhum form
post_1 <- POST("https://httpbin.org/post")

# POST equivalente a https://httpbin.org/get?show_env=1
form <- list(
  "um-numero" = 1,
  "uma-letra" = "a",
  "uma-data" = "2018-01-01")
post_2 <- POST("https://httpbin.org/post", body = form)
```

Responses

Nos últimos slides vimos chamadas HTTP, mas não vimos os seus resultados. Cada uma destas chamadas retorna aquilo que chamamos de **response**, uma resposta que contém metadados importantes e (mais relevante no nosso caso) o código HTML para renderizar a página requisitada.

Status

Status é um verificador simples de se a chamada deu certo. Um status igual a 200 geralmente indica que não houve nenhum erro ("OK"); 401 indica "Não Autorizado", 404 indica "Não Encontrado", 500 indica "Erro Interno Do Servidor" e assim por diante.

418 indica "Sou Um Bule De Chá".

```
get_1$status_code
```

```
#> [1] 200
```

Responses (cont.)

Cookies

Cookies são pequenos rastreadores que informam ao servidor que você passou por uma determinada parte do site. Eles se tornam importantes quando queremos fingir para o site que o nosso scraper já cumpriu um certo passo da navegação.

```
get_2$cookies
```

```
#>        domain  flag path secure expiration       name value
#> 1 httpbin.org FALSE    /  FALSE       <NA> meu-cookie     1
```

Responses (cont.)

Content

Este é o nosso objeto de interesse. Ele é o conteúdo HTML da página requisitada!

Pacote xml2

Uma vez com o HTML em mãos, é usando o pacote `xml2` que extraímos as informações que precisamos.

- Esse pacote fornece uma interface simples e consistente para trabalhar com HTML/XML.
- Suas principais funções são read_html, xml_find_all, xml_attrs, xml_text, xml_children, xml_parents, xml_siblings

Ler um html

Se usarmos `read_html()` e as funções de XML que vimos anteriormente, podemos começar a extrair importantes dados da página.

```
read_html(post_1)
```

```
#> {xml_document}
#> <html>
#> [1] <body><p>{\n  "args": {}, \n  "data": "", \n  "files": {}, \n  "form .
```

```
read_html(post_2)
```

```
#> {xml_document}
#> <html>
#> [1] <body><p>{\n  "args": {}, \n  "data": "", \n  "files": {}, \n  "form .
```

Content

Talvez mais fácil do que tentar ler o HTML retornado na response seja salvar o HTML em um arquivo e aí abrir este arquivo. Isso nos dá uma noção muito melhor do que recebemos de volta e é uma parte essencial do processo do web scraping.

```
post_2 <- POST(
  "https://httpbin.org/post", body = form,
  write_disk("webscraping/post.html", overwrite = TRUE))
read_html("webscraping/post.html")
```

```
#> {xml_document}
#> <html>
#> [1] <body><p>{\n  "args": {}, \n  "data": "", \n  "files": {}, \n  "form .
```

Clicando duas vezes no arquivo ou rodando a função `BROWSE()` abrimos ele em um navegador para que possamos investigar de perto se está tudo em ordem.

```
BROWSE("webscraping/post.html")
```

Content

Vale a pena saber...

O `content` de uma `response` pode ser virtualmente qualquer tipo de arquivo. Geralmente é um HTML e por isso usamos a função `read_html()` para acessar o conteúdo, mas usa-se a função `httr::content()` para acessar os contents de modo geral.

```
httr::content(post_1)
```

Network

A última ferramenta que demonstraremos do Chrome é a função de análise de **networking**. Com ela podemos acompanhar todas as chamadas HTTP que são feitas pela página, nos permitindo analisá-las de modo a reproduzí-las através do R.

Vamos ao Chrome...

PARSEAR

Parsear

- Definição de "Parsear"

Parsear

Exercício 1

O site http://example.webscraping.com/ contém uma série de links que possuem informações sobre países.

Construa um `data.frame` com as colunas `pais` e `link` dos dez primeiros países que aparecem na primeira página.

Parsear

Exercício 1 - GABARITO

O site http://example.webscraping.com/ contém uma série de links que possuem informações sobre países.

Construa um `data.frame` com as colunas `pais` e `link` dos dez primeiros países que aparecem na primeira página.

```
library(dplyr)
url <- "http://example.webscraping.com/"
paises <- url %>%
  GET %>%
  read_html %>%
  xml_find_all('//table//a') %>%
  tibble(node = .) %>%
  mutate(pais = xml_text(node),
         attrs = xml_attr(node, "href"))
```

Parsear

Exercício 2

A partir do objeto `paises` gerado no exercício 1 crie uma coluna `img_src` que guarde o atributo `src` das tags `<img>` (ele é local onde a imagem da bandeira está disponível).

Parsear

Exercício 2 - GABARITO

A partir do objeto `paises` gerado no exercício 1 crie uma coluna `img_src` que guarde o atributo `src` das tags `<img>` (ele é local onde a imagem da bandeira está disponível).

```
paises <- paises %>%
  mutate(img_src = xml_find_all(node, "img") %>% xml_attr("src"))
```

Parsear

Exercício 3

No navegador, inspecione o http://example.webscraping.com/ e identifique uma tabela no corpo do site. Em seguida, utilize a função `html_table()` do pacote `rvest` e compare o resultado com o observado no inspetor. Qual conteúdo a função devolveu: tag, texto ou atributos?

Parsear

Exercício 3 - GABARITO

No navegador, inspecione o http://example.webscraping.com/ e identifique uma tabela no corpo do site. Em seguida, utilize a função `html_table()` do pacote `rvest` e compare o resultado com o observado no inspetor. Qual conteúdo a função devolveu: tag, texto ou atributos?

```
library(rvest)
```

```
#> Registered S3 method overwritten by 'rvest':
#>   method            from
#>   read_xml.response xml2
```

```
tabela <- url %>%
  GET() %>%
  read_html %>%
  html_table()
```

O `html_table()` retorna o texto das células.

Download

Dentro da sua request (POST ou GET) use a função `write_disk()`. Por meio desta função você especifica que você gostaria de guardar o conteúdo (`content`) da resposta (`response`) em um arquivo externo.

Exemplo: baixando a imagem de uma bandeira.

```
url_img <- paste0(url, paises$img_src[1])
url_img
```

```
#> [1] "http://example.webscraping.com//places/static/images/flags/af.png"
```

```
img <- GET(url_img, write_disk("teste.png", overwrite = TRUE))
```

Download

Também há a possibilidade de salvar um `content` a qualquer momento.

```
img <- GET(url_img)

img %>% httr::content() %>% png::writePNG("teste2.png")
```

A função `content` retorna o conteúdo de uma dada `response`. OBS: No caso o conteúdo era um `png`, por isso utilizamos o `writePNG()` pra guardar o conteúdo.

Tidy Output

Até o momento apenas extraímos o conteúdo das páginas e aprendemos a baixa-las, mas saber deixar esse conteúdo arrumadinho também é muito importante. Uma tabela em `html` extraída com `html_table` é muito legal, mas existem jeitos mais legais de guardar informação.

Jeitos de guardar a mesma informação

Jeito 1

```
preg <- read.csv("preg.csv", stringsAsFactors = FALSE)
preg
#>           nome      reais    dolares
#> 1         João         NA         18
#> 2        Maria          4          1
#> 3        Pedro          6          7
```

Jeito 2

```
#>       moeda       João    Maria        Pedro
#> 1     reais         NA        4            6
#> 2   dolares         18        1            7
```

Tidy data

Toda tabela é um conjunto de **valores** e, por sua vez, cada **valor** pertence a uma **variável** e a uma **observação**.

1. **observações** são as entidades de onde se tomam medidas, como por exemplo as pessoas da nossa tabela.
2. **variáveis** são tipos de medidas que podem ser extraídas de uma observação

```
#>           nome     moeda  quantidade
#> 1        Maria     reais           4
#> 2        Maria     dolares         1
#> 3         João     reais          NA
#> 4         João     dolares        18
#> 5        Pedro     reais           6
#> 6        Pedro     dolares         7
```

Tidy data

Uma base é dita `tidy` quando:

1. Cada **variável** forma uma coluna.
2. Cada **observação** forma uma linha.
3. Cada tipo de observação forma uma tabela

Tipo comum de base bagunçada

```
#>    religion      `<$10k` `$10-20k` `$20-30k` `$30-40k` `$40-50k`
#>    <chr>           <int>     <int>     <int>     <int>     <int>
#>  1 Agnostic           27        34        60        81        76
#>  2 Atheist            12        27        37        52        35
#>  3 Buddhist           27        21        30        34        33
#>  4 Catholic          418       617       732       670       638
#>  6 Evangelical …     575       869      1064       982       881
#> # ... with 8 more rows, and 4 more variables: `$75-100k` <int>,
#> #   `$100-150k` <int>, `>150k` <int>, `Don't know/refused` <int>
```

Tidy data

Versão arrumada:

```
#>    religion                income frequency
#>    <chr>                   <chr>      <int>
#>  1 Agnostic                <$10k         27
#>  2 Atheist                 <$10k         12
#>  3 Buddhist                <$10k         27
#>  4 Catholic                <$10k        418
#>  5 Don't know/refused      <$10k         15
#>  6 Evangelical Prot        <$10k        575
```

BREAK (15 minutos)

Mão na Massa

Exercícios:

1. Crie uma conta manualmente e depois construa uma função para se logar.
2. Faça uma requisição que baixa a página de Andorra.
3. Extraia os dados de andorra numa tabela `tidy`.

ITERAR

O que significa "iterar"

Iterar nada mais é que do que repetir um comportamento várias vezes para parâmetros ligeiramente diferentes. Em computação geralmente falamos em "iterar em uma lista" por exemplo, mas nesse caso vamos estar iterando ao longo das páginas as quais desejamos baixar.

```
i <- 1
v <- 1:15
while (i <= length(v)) {
  v[i] <- v[i] + 5
  i <- i + 1
}
v
```

```
#>  [1]  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Precisamos fazer isso porque geralmente vamos querer que o scraper puxe várias páginas semelhantes de um mesmo site, logo, precisamos automatizar esse processo iterativo.

Salvar Tidy output

Vale a pena extrair as coisas no format `tidy` por vários motivos.

1. Vai ser mais fácil analisar os dados.
2. Vai ser mais fácil juntar os dados.

Mas também existem outros cuidados que você pode ter na hora de repetir uma requisição várias vezes:

1. Salve resultados parciais.
2. Deixe tudo o mais `tidy` o possível.
3. Use a função `carefully`.

Iteração - Básico

Existem algumas maneiras de iterar uma função que faz *scraping*

1. Funções como `purrr::walk`, `sapply`, etc
    - Não monta outra função, mas também não paraleliza nem trata erros
2. `for` e `while`
    - Lida um pouco melhor com erros, mas vai precisar de outros pacotes para paralelizar

Exemplo de for

```
for(i in 1:10){
  httr::GET(
    paste0('http://example.webscraping.com/places/default/view/',i)
  )
}
```

O loop acima não tem muito mistério. Para cada `i` estamos estamos aplicando a função `GET`, mudando o URL da requisição; o resultado é que em cada passada do loop, vamos estar pegando uma página do índice do site.

carefully

Web scraping é difícil porque, lá no fundo, estamos fazendo uma engenharia reversa...

...e fazendo isso **MUITA** coisa pode dar errado.

Para remediar isso, criamos uma função que faz três coisas:

1. Itera o mesmo resultado várias vezes (vetoriza)
2. Não trava a função se der erro
3. Paraleliza

carefully (cont.)

```r
library(abjutils)

pad <- function(str, b = "", a = "") { paste0(b, str, a) }

# Create wrapped version of pad() that executes over 4 cores,
# captures errors, and prints its current iteration with a
# probability of 50%
new_pad <- carefully(pad, p = 0.5, cores = 4)

# Execute new_pad() with some sample data
new_pad(c("asdf", "poiu", "qwer"), b = "0", a = "1")
```

Exercício

Baixe as páginas de todos os países

VALIDAR

Validação

Um dos passos mais importantes do web scraping é verificar se a requisição que fizemos retornou exatamente o que queríamos.

Entretanto, existem algumas características comuns a muitos websites que dificultam essa parte do trabalho

Alguns sites bloqueiam IP's, enquanto outros, construídos em `.NET` ou renderizando parte do conteúdo em `javascript`, precisam de mais do que um simples `POST` para chegar onde queremos.

Bloqueio de IP

Web scraping, por fazer um monte de requisições automáticas, é um inimigo comum dos administradores de websites. Existem várias formas de *desincentivar* o acesso automático, e o mais comum deles é o bloqueio de IP.

A solução mais simples para o bloqueio de IP, que resolve um problema tanto do usuário quanto do servidor, é inserir uma pausa entre um download e outro. Em R, isso é feito usando a função `sleep`

```
for(i in 1:10){
  httr::GET(
    paste0('http://example.webscraping.com/places/default/view/',i)
  )

  sleep(1)
}
```

Event validation e view state

Sites feitos em `.NET` (os `.aspx` espalhados pela internet) muitas vezes usam parâmetros esquisitos nas requisições `POST`

Esses parâmetros não significam nada em particular, mas são importantes para que as requisições funcionem corretamente.

Isso acontece, por exemplo neste site.

```
view_state <- r %>%
  xml2::read_html() %>%
  rvest::html_nodes("input[name='__VIEWSTATE']") %>%
  rvest::html_attr("value")

event_validation <- r %>%
  xml2::read_html() %>%
  rvest::html_nodes("input[name='__EVENTVALIDATION']") %>%
  rvest::html_attr("value")
```

MISCELÂNEA

Paralelização e Distribuição

Paralelização

Já aprendemos anteriormente a função `carefully()`, que permite a execução em paralelo de múltiplos downloads. Isso é possível porque, na realidade, o computador passa muito tempo da raspagem esperando o servidor entregar os arquivos necessários pra ele; esse tempo pode ser utilizado para começar a raspagem de outras páginas.

O motor por trás da carefully é a função `mcmapply()` do pacote `parallel`:

```
#> Unit: seconds
#>          expr      min     mean   median      max
#>  com_paralelo 1.831117 1.913419 1.906115 2.012591
#>  sem_paralelo 3.249223 3.374113 3.375318 3.498316
```

Para tirar mais vantagem disso, podemos por exemplo alugar máquinas na GCP ou na AWS que tenham mais núcleos, possibilitando um grau maior de paralelismo.

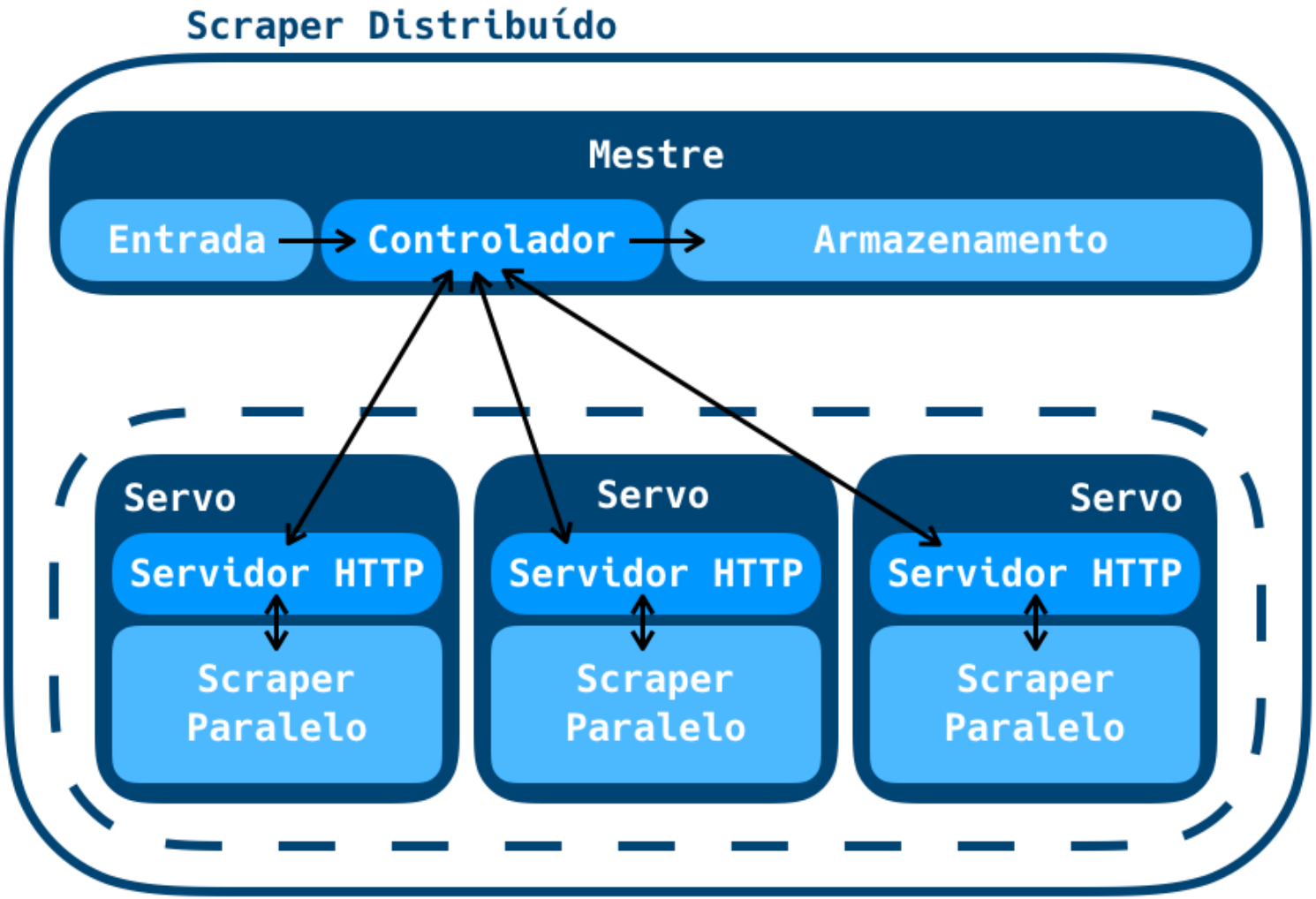
Paralelização e Distribuição (cont.)

Distribuição

... No entando, uma única máquina com muitos núcleos de processamento terá um gargalo no acesso a disco, ou seja, ela não conseguirá escrever todos os arquivos que queremos salvar em seu disco rígido com eficiência suficiente.

Por isso, podemos sincronizar várias máquinas virtuais para criar o que chamamos de *web scraper distribuído*.

Distribuição (cont.)



CAPTCHAs

Quebrar CAPTCHAs muitas vezes se faz necessário quando raspando uma página da web. Existem muitas formas de fazer isso, desde pagar acessos para o *Death By Captcha* até criar os seus próprios modelos de visão computacional; aqui vamos falar de algumas das formas mais simples e eficazes.

Lendo a imagem

```
pag_signup %>%
  GET() %>%
  read_html() %>%
  xml_find_first("//*[@id='recaptcha']//img") %>%
  xml_attr("src") %>%
  str_remove("") %>%
  base64decode() %>%
  readPNG() %>%
  writePNG("webscraping/captcha.png")
```

CATPCHAs (cont.)

Tratando imagens com magick


```
"webscraping/captcha.png" %>%
  image_read() %>%
  image_transparent("green", fuzz = 20) %>%
  image_transparent("blue", fuzz = 50) %>%
  image_transparent("red", fuzz = 50) %>%
  image_transparent("white", fuzz = 50) %>%
  image_quantize(2) %>%
  image_write("webscraping/captcha_clean.png")
```

CATPCHAs (cont.)

Lendo o texto com tesseract

```
"webscraping/captcha_clean.png" %>%
  image_read() %>%
  image_background("white") %>%
  image_ocr()
```

```
#> [1] "garden\n"
```

Muitas vezes pode ser necessário tratar ainda mais a imagem antes de passar o OCR. Para melhorar o acerto, tente aumentar, diminuir, filtrar as cores, borrar, desborrar, reduzir o barulho...

CATPCHAs (cont.)

Decryptr

Quando temos que lidar com imagens mais complicadas que as dos slides anteriores (principalmente quando as letras desejadas não estão em preto), pode ser que precisemos criar um modelo de *deep learning* para quebrar os CAPTCHAs.

Ensinar modelos desse tipo está fora do escopo da aula, mas nós criamos um pacote que já vem carregado com alguns modelos pré-prontos. Ele também pode te ajudar a criar seus próprios modelos.

```
decrypt(arq, model = "rfb")
```

OBRIGADO