Certificat Data Scientist

JavaScript Object Notation (JSON)

NicolasKlutchnikoff

Octobre 2021

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.2     v dplyr   1.0.7
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   2.0.0     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()

##
## Attaching package: 'jsonlite'

## The following object is masked from 'package:purrr':
##
##     flatten
```

Introduction

Des données variées

Les données ne sont pas toujours stockées dans des formats tabulaires. Il existe bien d'autres façons de les conserver et d'y accéder :

- Bases de données relationnelles/SQL ;
- Fichiers structurés : XML, YAML ou JSON ;
- Bases de données NoSQL ;
- WEB

JSON

C’est un format de plus en plus utilisé :

- MongoDB (NoSQL)
- API (interface de programmation applicative) web.
- Open Data : données publiques de l’État et des administrations.

Structure des fichiers JSON

Objectifs

Lisible par :

- des humains ;
- des machines.

Articulé autour d'une syntaxe simple et de types prédéfinis et non extensibles :

- 2 types structurés ;
- plusieurs types simples.

Types structurés - les objets JSON

Ils sont proches des **dictionnaires Python** ou des **listes** de R.

Syntaxe :

```
{
  "clé1": valeur1,
  "clé2": valeur2,
  ...
}
```

Principe clé/valeur avec des " " autour des clés.

Types structurés - les tableaux

Ils sont proches des **listes Python** ou des **vecteurs** de **R**.

```
[
  valeur1,
  valeur2,
  ...
]
```

Les listes permettent simplement de structurer des données de **façon ordonnée**.

Types simples (1)

Chaîne de caractères : elle est entre `"..."`. Donnons un exemple :

```
{
  "prénom": "Jean-Sébastien",
  "nom": "Bach"
}
```

Nombre : on l'écrit directement, comme dans l'exemple ci-dessous :

```
{
  "prénom": "Jean-Sébastien",
  "nom": "Bach",
  "nombre_enfants": 20
}
```

Types simples (2)

`true` ou `false` : il faut noter (surtout quand on utilise R) que les valeurs boo- léennes sont écrites en minuscules. Exemple :

```
{
  "prénom": "Jean-Sébastien",
  "nom": "Bach",
  "compositeur": true
}
```

`null` : si l'on ne souhaite pas donner de valeur à une clé.

```
{
  "prénom": "Jean-Sébastien",
  "particule": null,
  "nom": "Bach"
}
```

Imbrication

Pour le types structurés, les différentes valeurs peuvent être de n'importe quel type, y compris un autre type structuré.

Pour vérifier la validité d'un fichiers JSON, on peut visiter ce site.

```
{
  "prénom": "Jean-Sébastien",
  "nom": "Bach",
  "épouses": [
    {
      "prénom": ["Maria", "Barbara"],
      "nom": "Bach"
    },
    {
      "prénom": ["Anna", "Magdalena"],
      "nom": "Wilcke"
    }
  ]
}
```

Utilisation basique

Les principales fonctions

Il y en a quatre :

- `fromJSON()` : import ;
- `toJSON()` : export ;
- `stream_in()` : import ;
- `stream_out()` : export ;

Import et jeu de données bidon.

```r
library(jsonlite)
```

```r
df <- tibble(
  x = c(0, pi),
  y = cos(x)
)
```

JSON et data-frame (1)

```
toJSON(df)
```

```
## [{"x":0,"y":1},{"x":3.1416,"y":-1}]
```

Remarques :

- le résultat est un chaînes de caractère !
- il y a un tableau JSON ;
- paradigme individus/variables respectés.
- perte de précision...

JSON et data-frame (2)

```
df2 <- df %>% toJSON() %>% fromJSON()
```

La fonction `toJSON()` prend une chaîne de caractère représentant un fichier JSON bien formaté en l'importe si possible sous la forme d'un data-frame

Remarques :

- même structure ;
- confirmation de la perte de précision.

```
df2
```

```
##        x  y
## 1 0.0000  1
## 2 3.1416 -1
```

JSON et data-frame (3)

```r
fromJSON('[{"x":1},{"y":2}]')
```

```
##    x  y
## 1  1 NA
## 2 NA  2
```

Remarques :

- la situation est plus délicate ;
- on utilise deux quotes différentes...

JSON et data-frame (4)

```
df1 <- fromJSON('[{"x":[1, 2, 3]},{"x":4}]')
```

```
class(df1)
```

```
## [1] "data.frame"
```

Un data-frame... inhabituel.

```
df1
```

```
##         x
## 1 1, 2, 3
## 2       4
```

```
df1$x
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] 4
```

JSON et data-frame (5)

Par défaut, la fonction `fromJSON()` tente de simplifier les vecteurs. On peut forcer le comportement inverse :

```
fromJSON('[{"x":[1, 2, 3]},{"x":4}]', simplifyVector = FALSE)
```

```
## [[1]]
## [[1]]$x
## [[1]]$x[[1]]
## [1] 1
##
## [[1]]$x[[2]]
## [1] 2
##
## [[1]]$x[[3]]
## [1] 3
##
##
##
##
## [[2]]
## [[2]]$x
## [1] 4
```

JSON et data-frame (6)

Lequel choisr ?

```
fromJSON('[{"x":{"xa":1, "xb":2}},{"x":3}]')
```

```
##      x
## 1 1, 2
## 2    3
```

```
fromJSON('[{"x":{"xa":1, "xb":2}},{"x":3}]', simplifyDataFrame = FALSE)
```

```
## [[1]]
## [[1]]$x
## [[1]]$x$xa
## [1] 1
##
## [[1]]$x$xb
## [1] 2
##
##
##
## [[2]]
## [[2]]$x
## [1] 3
```

LE vélo STAR

Qu’est-ce ?

Nous allons présenter l’API qui permet, en temps réel, d’obtenir des données sur le service **LE vélo STAR**, le Vélib rennais...

Il faut commencer par suivre ce lien pour avoir plus d’informations.

Deux bases vont nous intéresser :

- l’état des stations ;
- la topographie des stations.

API

On peut voir que la requête prend une forme spécifique :

- le nom de domaine : `https://data.rennesmetropole.fr/` ;
- le chemin d'accès à l'API : `api/records/1.0/search/` ;
- une indication sur le jeu de données : `?dataset=etat-des-stations-le-velo-star-en-temps-reel` ;
- séparées par des esperluettes `&`, une liste de facettes à inclure dans les résul- tats. Par exemple `&facet=nom&facet=etat`.

Comment utiliser ça dans `R` : `jsonlite` !

En avant!

```
url <- paste0(
  "https://data.rennesmetropole.fr/api/records/1.0/search/",
  "?dataset=etat-des-stations-le-velo-star-en-temps-reel",
  "&rows=100",
  "&facet=nom",
  "&facet=etat",
  "&facet=nombreemplacementsactuels",
  "&facet=nombreemplacementsdisponibles",
  "&facet=nombrevelosdisponibles"
)
ll <- fromJSON(url)
```

Finalisation

Il faut enfin aller récupérer les données dans la bonne liste !

```
df <- ll$records$fields
glimpse(df)
```

```
## Rows: 57
## Columns: 8
## $ etat                          <chr> "En fonctionnement", "En fonctionnement"~
## $ lastupdate                    <chr> "2021-10-04T05:45:08+00:00", "2021-
10-04~
## $ nombrevelosdisponibles        <int> 11, 10, 23, 3, 15, 10, 17, 16, 0, 7, 25,~
## $ nombreemplacementsactuels     <int> 16, 24, 24, 19, 24, 18, 29, 30, 24, 19, ~
## $ nom                           <chr> "Musée Beaux-Arts", "Champs Libres", "Br~
## $ nombreemplacementsdisponibles <int> 5, 14, 1, 16, 9, 8, 12, 14, 24, 12, 0, 2~
## $ idstation                     <chr> "5510", "5516", "5534", "5538", "5546", ~
## $ coordonnees                   <list> <48.10960, -1.67408>, <48.105537, -
1.67~
```

Le tour est joué !

Pour aller plus loin

Le format ndjson

Pour Newline Delimited JSON !

Très souvent les API, ou les requêtes MongoDB (on y reviendra), renvoient des fichiers JSON non valides. Ces fichiers ont en fait la structure typique suivante :

```
{"x":1,"y":2}
{"x":3,"y":4}
...
```

Chaque ligne est un fichier JSON valide qu'on peut lire à l'aide de la fonction `stream_in()`

stream_in()

```
url <- "http://jeroen.github.io/data/diamonds.json"
diamonds <- jsonlite::stream_in(url(url))
```

```
## opening url input connection.
```

```
## closing url input connection.
```

```
head(diamonds,3)
```

```
##   carat     cut color clarity depth table price    x    y    z
## 1  0.23   Ideal     E     SI2  61.5    55   326 3.95 3.98 2.43
## 2  0.21 Premium     E     SI1  59.8    61   326 3.89 3.84 2.31
## 3  0.23    Good     E     VS1  56.9    65   327 4.05 4.07 2.31
```