Atelier Lodash

(orienté Lodex)

21 Mars 2024
Jean-Joffrey PARENTIN – INIST-CNRS – Service « Textes & Corpus - ISTEX »
Anaël KREMER – INIST-CNRS – Service « Appui au pilotage scientifique »

Sommaire

Introduction

Lodash est une bibliothèque JavaScript qui fournit des fonctions utilitaires pour les tâches de programmation courantes en utilisant le paradigme de la programmation fonctionnelle.

Programmation qui est donc centrée sur l'utilisation de fonctions (au sens mathématique), de valeurs et qui permet l'association de plusieurs fonctions (composition).

Lodash est un « fork » d'Underscore, c'est-à-dire qu'il est créé à partir du code source d'Underscore mais en devient indépendant afin de bénéficier de ses propres développements.

A l'origine, Underscore avait été conçu pour faciliter le développement en fournissant des utilitaires permettant de travailler plus facilement avec les tableaux, les nombres, les objets et les chaînes de caractères.

Lodash est créé comme sur-ensemble (superset) d'Underscore dans le sens où il apporte plus de fonctionnalités que ce dernier. Plus lourd qu'Underscore, Lodash est aussi plus rapide, mieux adapté à l'enchaînement de fonctions et surtout, il permet de gérer des objets imbriqués (nested).

Les deux bibliothèques tirent leur nom du caractère _ , qui est la variable globale qui contient toutes les fonctions utilitaires.

Une fonction Lodash classique

Une fonction (ou méthode) Lodash doit toujours déclarer des arguments dans un ordre spécifique.

- Le 1er argument déclaré ici est la chaîne de caractères à modifier, en l'occurrence "abc".
- Le 2ème argument est le motif à remplacer dans la chaîne de caractères, c'est aussi "abc".
- Le 3ème argument, "hello world", correspond à la valeur de remplacement du motif trouvé dans la chaîne de caractères.

Un enchaînement de fonctions Lodash

Afin d'effectuer plusieurs opérations sur des données, il convient de créer une chaîne de méthodes.

- Pour ce faire, il faut débuter par la méthode _***.chain()*** et y déclarer les données à modifier.
- On peut ensuite ajouter les différentes méthodes souhaitées.
- Puis on parachève le tout avec _***.value()***

Ecrire du Lodash dans Lodex

Dans Lodex, les scripts s'écrivent dans le mode enrichissement. Mais dans cet "éditeur", plusieurs petits programmes entrent en jeu afin de nous faciliter la tâche. Le chaînage des méthodes y est possible par défaut. Les méthodes _***.chain*** et _***.value*** sont actives mais pas visibles. Ce qui rend possible cet ensemble de transformations :

Autre différence notable, les données sont déclarées avec "path", "value" et ***'get'*** ici. Ce qui explique pourquoi il n'y a que 2 arguments déclarés à la suite de ***'replace'*** et non plus 3 comme vu plus haut.

Différences entre enrichissement et transformers

Dans Lodex les transformers portent sensiblement le même nom que les méthodes Lodash qu'ils exécutent. La différence principale réside dans le fait qu'un transformer modifie les données dans une ressource là où un enrichissement modifie directement les données du dataset et les stocke dans celui-ci.

L'autre différence est que bon nombre de transformers agissent de façon "magique".

Nous le verrons en détail plus tard, mais les méthodes Lodash fonctionnent avec un certain type de données. Certaines méthodes transforment des chaînes de caractères (string), d'autres agissent sur des tableaux (array) et d'autres enfin sur des objets.

Ainsi, sans que nous le voyons, un transformer va d'abord activer des tests pour savoir quel type de données on lui demande de traiter. Un script spécifique sera alors lancé en fonction des réponses renvoyées par les tests.

A titre d'exemple, nous souhaitons remplacer le rnsr "199512096Z" par "BLOB"

Opérations de transformation

REPLACE 199512096Z BLOB

["200112481S","199320514H","20122

["195922846R","BLOB","200012191F

["199712093M","201119577L","20012

["BLOB"]

Le transformer a bien remplacé les valeurs dans des tableaux. Or la méthode ***'replace'*** doit être utilisée sur des strings. Si nous essayons en enrichissement cela donne :

Mode avancé

```
1 [assign]
2 path=value
3 value=get("value.ApilRnsr").replace("199512096Z","BLOB")
```

ApilRnsr

"200112481S,199320514H,201220349W,2009184

"195922846R,BLOB,200012191F,201220716V,200

"199712093M,201119577L,200123308K,20011974

"BLOB"

La valeur a également été transformée, mais notre tableau a été "cassé". Nous avons désormais un seul string en lieu et place de notre tableau de valeurs initial.

Quelques précautions d'usage

- Attention donc à toujours vérifier à quel type de données s'applique la méthode que vous voulez utiliser. Référez-vous à la documentation Lodash le cas échéant. Au mieux la méthode ne fonctionnera pas, au pire vos données seront déstructurées.

- Autre point, un transformer peut être supprimé sans impacter les données d'origine. A l'inverse, si l'on supprime un enrichissement, la colonne et les données disparaissent avec ! Evitez donc, même si cela peut paraître plus simple, de nommer un enrichissement du même nom que la colonne sur laquelle il agit. En cas d'erreur dans le script, vous perdrez les données qui viennent du dataset de base.

- Enfin, et pour rester dans les bonnes pratiques, astreignez vous à nommer les enrichissements correctement. Evitez tout caractère spécial ! Les conventions Camel case ou Pascal case vous éviteront des problèmes lorsque vous devrez plus tard appeler ces colonnes.

```
[assign]
path=value
value=get("value.EnrichIstex.publicationDate").concat(self.value.EnrichConditor.host/publicationDate)
```

null

Quelques précautions d'usage

"Bad programmers worry about the code. Good programmers worry about data structures and their relationships."
Linus Torvalds, créateur de Linux et Git

- Comme dit précédemment, les fonctions Lodash s'appliquent à différents types de données dont on peut distinguer deux classes : les types primitifs et les types objets. Voici les principaux types :

- Types de données primitifs :
  - **Number** : Représente des nombres, qui peuvent être entiers (*integer*) ou à virgule (*floating*). Ex age = 42
  - **String** : ou chaîne de caractères représente une séquence de caractères, on les repère grâce aux guillemets placés en début et fin de séquence. Ex Nom = "John" ou encore "42" qui n'est donc ici plus un nombre.
  - **Boolean** : représente des valeurs booléennes (true ou false) qui rendent compte de la vérité ou non d'une expression ou condition. Ex is OA = false
  - **Null** : représente une valeur nulle ou l'absence de valeur. En général null est utilisé pour indiquer de façon intentionnelle et explicite que la variable n'a pas de valeur. Ex abstract = null (la notice n'a pas d'abstract)
  - **Undefined** : représente une valeur manquante ou vide. En pratique, il est déconseillé d'assigner soi-même "undefined" à une valeur en raison de sa proximité conceptuelle avec "null". JavaScript renvoi aussi "undefined" si l'on essaye d'accéder a une propriété qui n'existe pas. Si par exemple on veut extraire les valeurs associées à une clé, mais que cette dernière n'existe pas dans toutes les notices alors le résultat sera "undefined" pour les objets n'ayant pas cette clé.

Quelques précautions d'usage

- Les types objets sont des structures de données plus complexes qui permettent de regrouper des valeurs et des fonctionnalités connexes.
- Types de données objets :
  - **Object** : En JavaScript, un objet est une collection non ordonnée de paires clé-valeur. Chaque paire clé-valeur est une propriété de l'objet. Les clés (ou noms de propriété) sont des chaînes de caractères ou des symboles, et les valeurs peuvent être de n'importe quel type de données.

Cet objet par exemple a pour valeurs des chaînes de caractères, des nombres, un booléen, un objet (address) ainsi qu'un tableau (children).

  - **Array** : ou tableau représente une collection ordonnée d'éléments indexés numériquement. Chaque élément du tableau étant accessible par son indice numérique. Dans l'exemple ci-dessus "Bob" est le 1er élément et a donc l'index [0], "Alice" étant en second elle possède l'indice [1].

Ces types de structures peuvent s'imbriquer ! On peut avoir des tableaux dans des objets, des objets dans des tableaux (tableaux d'objets), des tableaux de tableaux (une matrice), des objets dans des objets dans des objets…

Avant de vous lancer

Cet atelier est dédié à Lodash, mais vous pouvez aussi utiliser des fonctions JavaScript natives qui ne sont pas dans la bibliothèque Lodash. Quelques fonctions EZS ont également été implémentées dans Lodex. Ainsi, au sein d'un même enchaînement de fonctions, il est possible de combiner du Lodash avec du JavaScript standard et du EZS.

Ce support a été conçu dans le but d'être réutilisé et compréhensible pour toute personne ayant suivi ou non l'atelier.

Afin d'être guidé au mieux dans les travaux pratiques de nombreux renvois sont présents vers une documentation traduite en français pour l'occasion, mais non-exhaustive bien sûr. Sont détaillées les catégories auxquelles appartiennent les fonctions, les fonctions elles-mêmes avec des explications, les opérateurs JavaScript qui serviront à écrire des fonctions de test et quelques instructions EZS.

Enfin, une section recensera les différentes ressources et outils qui ont servi à construire cet atelier et son support, et d'autres qui vous permettront d'aller plus loin.

Travaux Pratiques

Les exercices proposés s'appuient sur un mini-corpus de 8 DOIs, afin de pouvoir visualiser toutes les transformations.

Une fois les données chargées, il convient de réaliser deux enrichissements :

- ✓ "EnrichIstex" à partir du web-service *https://biblio-tools.services.istex.fr/v1/istex/works/expand*
- ✓ "EnrichConditor" à partir du web-service *https://biblio-tools.services.istex.fr/v1/conditor/works/expand*

10.3390/info10050178
10.4000/terminal.3665
10.1111/j.1467-9493.1986.tb00179.x
10.1111/evo.13045
10.1002/bies.201100025
10.1002/bbpc.19961000952
10.1007/978-3-030-35578-4_5
10.1007/978-3-030-35578-4_17

Pour tout exercice, une liste des fonctions utiles est donnée avec des renvois vers leurs définitions. Pour autant, il n'est pas nécessaire de toutes les appliquer pour trouver la solution, certaines sont présentes afin d'être comparées (***'first'*** ou ***'head'*** par exemple).

Et enfin, bien souvent, il peut exister plusieurs solutions à un même problème !

Exercice 1

extraction d'une chaîne de caractères, nettoyage et remplacement

Création d'une colonne "Title" depuis la colonne "EnrichConditor" et du champ "title/default".

On enlèvera les éventuels espaces superflus, les caractères spéciaux, on s'assurera que le 1er caractère est en majuscule et on effectuera un premier ***'replace'*** pour supprimer le caractère spécial uE01F.

Fonctions utiles :

- .isString()
- .trim()
- .deburr()
- .capitalize()
- .replace()

Voir la solution

Exercice 2

extraction d'une chaîne de caractères, nettoyage et suppression des champs vides

Création d'une colonne "Abstract" depuis la colonne "EnrichConditor" et du champ "abstract/default".

On remplacera les champs vides par "n/a", on enlèvera les éventuels espaces superflus et on s'assurera que le 1er caractère est bien en majuscule.

Fonctions utiles :

- .isString()
- .trim()
- .capitalize()
- .replace()

Voir la solution

Exercice 3

transformation d'une chaîne de caractères en tableau afin de le manipuler et d'en extraire une partie

Création d'une colonne "PublicationYearConditor" depuis la colonne "EnrichConditor" et du champ "host/publicationDate".

On découpera la chaîne de caractères obtenue de façon à ne garder que l'année de publication.

Fonctions utiles :

- .split()
- .first()
- .reverse()
- .last()

Voir la solution

Exercice 4

concaténation de deux valeurs et obtention d'un résultat basé sur la comparaison des deux

Création d'une colonne "PublicationYear" depuis la colonne "EnrichIstex" et du champ "publicationDate" (où ne figurent que les années).

On concaténera ensuite la colonne créée précédemment : "PublicationYearConditor".

On cherchera enfin à ne garder que l'année la plus ancienne des deux.

Fonctions utiles :

- .concat()
- .compact()
- .min()

Voir la solution

Exercice 5

Concaténation de tableaux, dédoublonnage, nettoyage, remplacement et filtrage

Création d'une colonne "Keywords" en concaténant les champs "EnrichConditor.keywords/en/author" et "EnrichIstex.keywords/teeft" (attention aux intitulés).

On supprimera ensuite les valeurs 'falsey', on mettra le 1er caractère de chaque chaîne en majuscule, on dédoublonnera, triera par ordre alphabétique, on remplacera "Istex" par "ISTEX" et enfin on retirera une valeur particulière ("Bonnefoy" qui apparaît en notice 7 et 8).

Fonctions utiles :

- .map() (2 slides pour savoir comment l'utiliser)
- .concat()
- .compact()
- .capitalize()
- .uniq()
- .sort()
- .pull()
- .replace()

Voir la solution

Exercice 6

création d'un tableau d'un seul niveau de profondeur à partir de tableaux imbriqués dans des objets

Création d'une colonne "Authors" depuis la colonne "EnrichConditor" et du champ "fullname" lui-même imbriqué dans le champ "authors".

Fonctions utiles :

- .isArray()
- .map()

Voir la solution

Exercice 7

création d'un tableau d'un seul niveau de profondeur à partir de tableaux imbriqués dans des objets

Création d'une colonne "Affiliations" depuis la colonne "EnrichIstex" et du champ "affiliations", lui-même imbriqué dans le champ "author".

On s'assurera qu'il n'y ait pas de doublons et que les valeurs contenant uniquement une adresse mail soient supprimées.

Fonctions utiles :

- .map()
- .flatten()
- .flatMap()
- .uniq()
- .filter()
- .startsWith()
- .remove()

Voir la solution

Exercice 8

création d'un tableau de valeurs, puis association d'une valeur dynamique à chaque valeur du tableau

Création d'une colonne "Rnsr" depuis la colonne "EnrichConditor" et du champ "rnsr" lui-même imbriqué dans le champ "authors".

Puis, création d'une colonne "RnsrYear" où l'on assemblera à chaque rnsr récupéré précédemment l'année de publication associée ("PublicationYearConditor").

Ceci afin de pouvoir utiliser ultérieurement le web-service *https://mapping-tools.services.istex.fr/v1/rnsr-year/instituts-cnrs* qui nécessite en entrée des données structurées de cette façon ["rnsr|year","rnsr|year"…]

Fonctions utiles :

- .flatMap()
- .compact()
- .uniq()
- .map()

Voir la solution

Exercice 9

création d'un tableau d'éléments regroupés, puis réduction du tableau aux seuls éléments répondant à une condition spécifiée

Création d'une colonne "WsInstituts" depuis la colonne "RnsrYear" avec utilisation du web-service.

Création d'une colonne "Instituts" afin de transformer chaque institut en une chaîne.

Puis création d'une colonne "RnsrInshs" dans laquelle on associera à chaque rnsr ses instituts d'appartenance (soit le résultat du ws) dans une matrice (un tableau de tableaux). On cherchera ensuite à ne conserver que les rnsr ayant pour institut principal l'INSHS (à titre d'exemple).

Fonctions utiles :

- .map()
- .split()
- .zip() (2 slides pour savoir comment l'utiliser)
- .flatten()
- .reduce() (5 slides pour savoir comment l'utiliser)
- .slice()
- .some()
- .push()

Voir la solution

Exercice 10

création d'un objet, puis création de valeurs spécifiques et fonction de la présence d'une ou de plusieurs clés dans l'objet

Création d'une colonne "Base" depuis les colonnes "EnrichIstex" et "EnrichConditor".

On cherchera à connaître la présence de notices dans les bases de données d'après leurs identifiants (respectivement "ark" et "sourceUidChain").

On attribuera donc à chaque combinaison existante une valeur : pour une seule présence dans Istex on attribuera la valeur "Istex", pour Conditor "Conditor", pour une présence dans les 2 bases "Istex & Conditor" et enfin pour aucune présence la valeur "Aucune Base".

Fonctions utiles :

- .fix()
- .reduce()

Voir la solution

Solution à l'exercice 1

```
[assign]
path=value
value=get("value.EnrichConditor.title/default").deburr().trim().capitalize().replace(/\uE01F/g," ")
```

- Pour accéder au champ 'title/default' il suffit de l'ajouter derrière le nom de la colonne avec un '.', il n'est pas nécessaire de faire plusieurs ***'get'***.

- On peut d'abord vérifier que l'on traite bien des chaînes de caractères avec la fonction ***'isString'***.

- On ajoute ensuite les différentes fonctions de nettoyage et de mise en forme et enfin, pour enlever le caractère spécial, il convient d'appliquer un ***'replace'***.

- Mais si l'on se contente d'encadrer ce caractère entre "" on constate que seul la 1ère occurrence de ce caractère est supprimée. Il faut donc utiliser une regex dans le replace, pour cela on place des '/' avant et après le motif à remplacer puis un 'g' pour indiquer une correspondance globale dans toute la chaîne.

Solution à l'exercice 2

```
[assign]
path=value
value=get("value.EnrichConditor.abstract/default").trim().capitalize().replace(/^$/,"n/a")
```

Comme dans l'exercice précédent on récupère le champ souhaité puis on applique les transformations demandées.

2 subtilités à noter ici :

- Il faut s'assurer du bon ordre des fonctions ! Si on débute par ***'replace'*** on obtiendra "N/a" et non "n/a". Un mauvais enchaînement de fonctions peut rendre une fonction caduque comme par exemple : "open access" > .capitalize() puis .replace("open access","acces ouvert") renverra "Open Access".

- Si une chaîne vide est représentée par "", on ne peut pas la capturer en l'écrivant de la sorte (testez pour voir le résultat). Il faut ici aussi appliquer une regex afin de capturer une chaîne vide.

Solution à l'exercice 3

```
[assign]
path=value
value=get("value.EnrichConditor.host/publicationDate").split("-").first()
```

- L'objectif de cet exercice est de voir comment, pour obtenir certaines valeurs, il peut être utile de basculer entre différents formats de données. Il est bien sûr possible d'appliquer une regex pour ne garder que les quatre 1ers caractères de la chaîne, mais ce n'est pas forcément l'idée la plus simple…

- Ici on va convertir la chaîne en tableau avec un ***'split'*** séparant ainsi les années, les mois et les jours.

- L'année étant toujours en index 0 du tableau, un '***first'*** ou '***head'*** suffit à ne conserver que cet élément et à le restituer en String.

Si les dates étaient renseignées sous format little endian (jj-mm-aaaa) on aurait dû appliquer la fonction '***last'*** pour extraire le dernier élément du tableau. Vous pouvez essayer ici en inversant le tableau avec la fonction '***reverse'*** puis avec '***last'***.

Solution à l'exercice 4

```
[assign]
path=value
value=get("value.EnrichIstex.publicationDate").concat(self.value.PublicationYearConditor).compact().min()
```

- Afin de pouvoir comparer les années de publication, il faut d'abord les réunir dans un tableau. On utilisera donc la fonction ***'concat'*** qui va créer un tableau avec les 2 chaînes de caractères souhaitées.

- Notons que dans cette fonction on souhaite récupérer deux valeurs situées à deux chemins différents de l'objet. Il faudrait donc ajouter un second ***'get'*** à l'intérieur de ***'concat'*** afin de récupérer "PublicationYearConditor". Pour simplifier l'écriture on utilisera ***'self'*** pour déclarer le chemin. Self faisant référence à l'objet courant.

- Logiquement, on serait tenté de convertir toutes les années en nombre afin de pouvoir les comparer avec la fonction mathématique ***'min'***. Mais ce n'est pas nécessaire, la fonction marche aussi sur des chaînes, et même sur des caractères alphabétiques.

- ***'compact'*** est nécessaire avant ***'min'***, enlevez-le, vous constaterez la différence entre `null` et `""`.

Solution à l'exercice 5

```
[assign]
path=value
value=get("value.EnrichConditor.keywords/en/author").concat(self.value.EnrichIstex["keywords/teeft"]).compact().map(item=>_.capitalize(String(item))).map(value=>value.replace("Istex","ISTEX")).uniq().sort().pull("Bonnefoy")
```

- On commence par un cas particulier, un ***'concat'*** habituel ne fonctionnera pas en raison des '/' dans le nom des champs comme vu ici. Il faut donc corriger la syntaxe et mettre le champ en question sous [""].

- Ensuite on cherche à utiliser des fonctions propres à des strings, ***'replace'*** et ***'capitalize'***, or nous sommes face à un tableau. Il faut donc appliquer les fonctions à chaque élément composant le tableau. Nous placerons donc ces fonctions à l'intérieur de la fonction ***'map'.*** Cette fonction récupère, par itération des valeurs sous forme de tableau.

- Pour ajouter une fonction à l'intérieur de ***'map'*** il faut créer une fonction dite anonyme ou fléchée qui est symbolisée par "=>". A titre d'exemple, la fonction `.map(item=>item.trim())` signifie: récupère un tableau de valeurs, où pour chaque valeur renvoyée seront retirés les blancs en début et fin de ligne.

Solution à l'exercice 5

```
[assign]
path=value
value=get("value.EnrichConditor.keywords/en/author").concat(self.value.EnrichIstex["keywords/teeft"]).compact().map(item=>_.capitalize(String(item))).map(value=>value.replace("Istex","ISTEX")).uniq().sort().pull("Bonnefoy")
```

- L'utilisation d'une fonction anonyme nous fait sortir du contexte Lodex (Lodash + EZS), cela signifie qu'après "=>" nous évoluons dans un contexte JavaScript.

- Mais, autre cas particulier ici, l'utilisation de ***'capitalize'*** dans une fonction anonyme ne fonctionnera pas sous cette forme : `.map(item=>item.capitalize())`

- La raison étant que, comme dit précédemment, nous sommes dans un contexte JavaScript et malheureusement ***'capitalize'*** est une fonction Lodash et pas une fonction Javascript native !

- Il faut donc invoquer explicitement Lodash avec un '_'. Et, toujours dans ce cas précis, il faut aussi déplacer l'argument (item) à l'intérieur des parenthèses et non plus avant la fonction.

Solution à l'exercice 5

```
[assign]
path=value
value=get("value.EnrichConditor.keywords/en/author").concat(self.value.EnrichIstex["keywords/teeft"]).compact().map(item=>_.capitalize(String(item))).map(value=>value.replace("Istex","ISTEX")).uniq().sort().pull("Bonnefoy")
```

- `_.capitalize(item)` fonctionnerait dans notre exemple mais, par sécurité, on s'assurera que chaque item devienne une chaîne de caractères en ajoutant la fonction ***'String'*** issue de Javascript.

A ce sujet, nous utiliserons les termes 'value' ou 'item' dans les fonctions fléchées des différents exercices pour ne pas semer de confusion avec des noms de champs et pour bien comprendre comment agissent les fonctions. La bonne pratique voudrait que l'on nomme notre fonction de la sorte : `.map(keyword=>_.capitalize(String(keyword)))`

- Enfin, la suppression de "Bonnefoy" pourrait se faire avec ***'filter'*** souvent utilisé dans Lodex : `.filter(x=>!"Bonnefoy".includes(x))`
  On lui préfèrera la fonction ***'pull'***, bien plus simple ici. La différence entre les 2 fonctions se situe au niveau des arguments qu'on peut leur attribuer. ***'pull'*** ne prenant que des valeurs là où ***'filter'*** utilise des conditions spécifiées dans une fonction de rappel (comme ***'startsWith'***).

Solution à l'exercice 6

```
[assign]
path=value
value=get("value.EnrichConditor.authors").map("fullname")
```

- La solution est pour le moins simple, mais permet de comprendre comment récupérer des valeurs d'un objet sous forme de tableau.

- Le champ "authors" étant un tableau d'objets, on ne peut récupérer ces valeurs en ajoutant simplement le nom du champ à la fin de ***'get'***. Il faut donc utiliser une méthode propre à "collection", en l'occurrence ***'map'***.

- Cette fonction va donc itérer sur le tableau d'auteurs et extraire les valeurs de la propriété "fullname" de chaque auteur, produisant un nouveau tableau contenant toutes les valeurs de "fullname".

Solution à l'exercice 7

```
[assign]
path=value
value=get("value.EnrichIstex.author").map("affiliations").flatten().map(value=>value.replace(/^E-mail:(.*)$/,"")).uniq().compact()
```

- Ici on souhaite extraire les multiples affiliations des auteurs de chaque notice, les données sont donc dans des tableaux, eux-mêmes compris dans l'objet "author". Il convient donc d'abord de réaliser un ***'map'*** du champ "affiliations" puis un ***'flatten'*** pour aplanir le tableau, car la propriété contient plusieurs valeurs.

- Ensuite pour enlever les valeurs commençant par "E-mail", on peut utiliser la fonction ***'replace'***, placée dans ***'map'*** car nous sommes dans un tableau, avec une regex pour capturer toutes les valeurs commençant par "E-mail". Ces valeurs étant désormais falsey, il faudra mettre un ***'compact'*** par après.

- Cependant ce script peut être simplifié, en réduisant le nombre de fonctions et en se passant de chercher une regex.

Solution à l'exercice 7

```
[assign]
path=value
value=get("value.EnrichIstex.author").flatMap("affiliations").filter(value=> !value.startsWith("E-mail")).uniq()
```

- Premièrement, en utilisant la fonction ***'flatMap'*** qui combine, comme son nom l'indique, les fonctions ***'flatten'*** et ***'map'***.

- Ensuite, on peut utiliser ***'filter'*** (ou ***'remove'***) associé à ***'startsWith'*** qui fonctionne comme la regex, capturant les valeurs débutant par "E-mail". Il n'y aura plus besoin d'ajouter ***'compact'*** puisque les valeurs capturées sont directement expurgées et non plus remplacées par "", qui doit être enlevé par la suite.

- Cela peut paraître contre-intuitif mais ***'filter'*** et ***'remove'*** ne renvoient que les arguments que l'on déclare. Il faut donc bien penser à ajouter l'opérateur de négation ***'!'*** dans la fonction. De sorte à ce que ne soient retournées que les valeurs ne commençant pas par "E-mail".

Solution à l'exercice 8

```
[assign]
path=value
value=get("value.EnrichConditor.authors").flatMap("rnsr").uniq().compact()
```

- Pour créer la colonne "Rnsr" on combine simplement les fonctions ***'flatMap'***, ***'uniq'*** et ***'compact'*** comme vu précédemment.

- Pour la colonne "RnsrYear" on souhaite qu'à chaque rnsr présent dans un tableau on accole l'année de publication. On ne peut donc utiliser ***'concat'*** qui ajoutera une seule fois l'année à chaque tableau de rnsr.

- Il faut utiliser ici une syntaxe dite de template strings (ou template literals) en Javascript. Cette syntaxe permet d'incorporer des variables ou expressions dans une chaîne de texte. La variable imbriquée dans la chaîne sera alors remplacée par la valeur dynamique souhaitée lors de l'exécution du code. Très utile pour constituer des modèles de blocs de texte prédéfinis et réutilisables.

Solution à l'exercice 8

```
[assign]
path=value
value=get("value.Rnsr").map(item => `${item}|${self.value.PublicationYearConditor}`)
```

- Les rnsr étant dans un tableau, il faut d'abord créer une fonction fléchée pour modifier chaque item du tableau.

- Pour créer le template string il faut l'encadrer dans des backquotes `` (cela permet de le distinguer d'une chaîne de caractères qui utilise les simple ou double quotes). La variable est ensuite insérée grâce à ${}

- Aussi la fonction suivante peut être explicitée comme suit : par itérations, on renverra pour chaque élément du tableau (c'est-à-dire un rnsr) : l'élément lui-même, un pipe comme séparateur, puis l'année de publication.

Solution à l'exercice 9

```
[assign]
path=value
value=get("value.WsInstituts").map(item=>item.split(";"))
```

- Après avoir lancé le web-service nous obtenons un tableau doté d'une structure particulière. Les résultats apparaissent sous forme de Strings où chaque élément correspond au résultat d'un rnsr. Mais dans un String apparaissent plusieurs instituts séparés par un ";".

- Pour obtenir le résultat souhaité, il est impératif que chaque institut soit sous forme de String, mais il faut aussi veiller à conserver une structure cohérente avec les résultats (1 rnsr = x instituts). Autrement dit, il faut créer une matrice, un tableau de tableaux, où chaque tableau regroupera les résultats d'un seul rnsr.

- Pour ce faire, on utilisera évidemment la fonction ***'split'*** mais pas sur le tableau. Il faut la faire fonctionner sur chaque String, donc chaque item du tableau. On utilise donc la fonction ***'map'*** afin d'itérer sur chaque item, dans laquelle on place notre ***'split'***. Chaque item doté de plusieurs instituts devenant ainsi un tableau où chaque institut est devenu une chaîne à part entière.

Solution à l'exercice 9

```
[assign]
path=value
value=zip(self.value.Rnsr,self.value.Instituts).map(item=>_.flatten(item)).reduce((result, item) =>{if
(item.slice(1).some(element => element === "INSHS (P)")) {result.push(item[0])}return result}, [])
```

- Pour savoir quels rnsr appartiennent à l'INSHS (P), il faut donc regrouper rnsr et instituts dans un même tableau. Pour ce faire, nous utiliserons la fonction 'zip' qui permet de créer un tableau en regroupant plusieurs tableaux de façon ordonnée. C'est-à-dire que le 1er élément contiendra les 1ers éléments de chaque tableau récupéré, le 2ème les 2èmes etc…

- Cependant, nous avons ici zippé un tableau de rnsr à une matrice d'instituts. Nous avons alors des tableaux sur 3 niveaux différents. Il faut donc de nouveau restructurer les données pour que chaque tableau soit composé d'un rnsr et d'un ou plusieurs instituts, et non plus d'un tableau d'instituts.

- Nous allons donc utiliser ***'flatten'*** pour aplanir les tableaux, mais à l'intérieur d'un ***'map'*** car nous voulons une matrice avec des tableaux d'un seul niveau de profondeur. Nous rassemblons donc les tableaux de niveau 3 et 2 en un seul.

Solution à l'exercice 9

```
[assign]
path=value
value=zip(self.value.Rnsr,self.value.Instituts).map(item=>_.flatten(item)).reduce((result, item) =>{if
(item.slice(1).some(element => element === "INSHS (P)")) {result.push(item[0])}return result}, [])
```

- Nous avons maintenant une matrice dans laquelle chaque tableau contient un rnsr avec le ou les instituts rattachés. Structure nécessaire pour réduire la matrice en un tableau contenant seulement les rnsr de l'INSHS (P). Nous utiliserons ensuite la fonction ***'reduce'*** qui pour chaque item testé renverra un résultat.

- Le test se décompose comme ceci : pour chaque tableau nous appliquons un ***'slice'***, qui va découper le tableau avec d'un côté le rnsr qui a pour index [0] et de l'autre côté tous les instituts avec l'index [1]. La fonction ***'some'*** va donc établir un test (que chaque élément soit strictement égal à "INSHS (P)") mais uniquement sur les instituts (précisé par `item.slice(1)`). Si le test renvoie `true` alors nous lui indiquons que le résultat à renvoyer est le rnsr par la fonction ***'push'*** avec comme argument (item[0]) qui correspond au rnsr.

- Si le test n'est pas validé, la fonction retournera un tableau vide, comme déclaré en toute fin de script.

Solution à l'exercice 10

```
[assign]
path=value
value=fix({ark:self.value.EnrichIstex.ark,sourceUidChain:self.value.EnrichConditor['business/sourceUidChain']})
.reduce((result, value, index, collection)=>{ if(collection.ark && !collection.sourceUidChain){return "Istex"}
if(collection.ark && collection.sourceUidChain){return  "Istex & Conditor"} if(!collection.ark &&
collection.sourceUidChain){return "Conditor"} return result},"Aucune Base")
```

- Nous allons utiliser ici la fonction ***'fix'*** qui permet d'attribuer une constante ou valeur fixe. Nous allons créer un objet avec deux clés en déclarant comme argument `{ark:, sourceUidChain:}`

- Nous ajoutons ensuite comme valeur à chaque clé une valeur dynamique avec 'self.value'. Nous venons ainsi de créer un objet littéral JavaScript avec les propriétés "ark" et "sourceUidChain". A chaque ligne sera donc renvoyé un objet avec soit les deux propriétés si elles existent, soit l'une, soit l'autre ou enfin un objet vide.

- Nous obtenons donc quatre cas de figure, que nous souhaitons remplacer par des valeurs spécifiques.

Solution à l'exercice 10

```
[assign]
path=value
value=fix({ark:self.value.EnrichIstex.ark,sourceUidChain:self.value.EnrichConditor['business/sourceUidChain']})
.reduce((result, value, index, collection)=>{ if(collection.ark && !collection.sourceUidChain){return "Istex"}
if(collection.ark && collection.sourceUidChain){return  "Istex & Conditor"} if(!collection.ark &&
collection.sourceUidChain){return "Conditor"} return result},"Aucune Base")
```

- Pour ce faire nous utiliserons la fonction ***'reduce'*** qui va itérer sur chaque élément de la collection (les propriétés donc) et réduira la collection à une seule valeur. La valeur étant le résultat accumulé de chaque itération.

- Il faut ensuite mettre en place des tests en définissant les valeurs souhaitées comme résultats. Ainsi, `if(collection.ark && !collection.sourceUidChain){return "istex"}` teste, si dans un objet, il y a présence de la propriété "ark" et qu'il y a absence de la propriété "sourceUidChain", si c'est le cas on réduit l'objet à un String : "Istex". Si non, on passe au prochain test…

Solution alternative à l'exercice 10

```
[assign]
path=value
value=get("value.EnrichIstex.ark").concat(self.value.EnrichConditor['business/sourceUidChain']).map((item)=
>!_.isEmpty(item)).reduce((result, value, index, collection) => { if (value === true && collection[index + 1] ===
false) {return "Istex"} if (value === false && collection[index + 1] === true) {return  "Conditor"} if (value ===
true && collection[index + 1] === true) {return  "Istex & Conditor"}return result}, "Aucune Base")
```

- On utilise tout d'abord un ***'concat'*** pour assembler les deux champs dans un tableau.

- On itère ensuite sur chaque élément du tableau auquel on applique la fonction ***'!.isEmpty'*** afin de savoir si la valeur testée est vide ou non. Cela nous renvoie un tableau de booléens avec quatre combinaisons possibles: `true,true` ou `true,false` ou `false,true` ou `false,false`

- Nous allons encore ici appliquer la fonction ***'reduce'*** mais en utilisant trois autres tests. Ce test spécifique: `if(value === true && collection[index + 1] === false) {return "Istex"}` signifie que si dans un tableau la valeur `true` est suivie de la valeur `false` alors le test est validé et l'on renvoie le résultat "Istex".

Solution alternative à l'exercice 10

```
[assign]
path=value
value=get("value.EnrichIstex.ark").concat(self.value.EnrichConditor['business/sourceUidChain']).map((item)=
>!_.isEmpty(item)).reduce((result, value, index, collection) => { if (value === true && collection[index + 1] ===
false) {return "Istex"} if (value === false && collection[index + 1] === true) {return  "Conditor"} if (value ===
true && collection[index + 1] === true) {return  "Istex & Conditor"}return result}, "Aucune Base")
```

- Nous poursuivons avec les autres cas de figure (`false,true` et `true,true`) selon les mêmes principes.

- Ayant précisé les conditions de trois cas de figure sur quatre, il n'est pas nécessaire d'écrire une condition pour le dernier, car si les trois tests ont échoué on est obligatoirement dans le dernier cas. On peut donc simplement déclarer "Aucune base" en fin de script après la virgule. C'est la valeur par défaut renvoyée en cas d'échec aux tests.

Les catégories

Les différentes méthodes sont regroupées en catégories en fonction de leur utilisation. Voici une brève explication de certaines de ces catégories :

1. **Array :** Cette catégorie regroupe les fonctions liées à la manipulation et à la transformation des tableaux. Elle comprend des opérations telles que le filtrage, le tri, la recherche, la modification et la création de tableaux.

2. **Collection :** La catégorie "Collection" englobe un ensemble plus large de fonctions pour travailler avec des collections de valeurs, pas seulement des tableaux. Cela inclut des objets, des tableaux, et d'autres structures de données.

3. **Function :** Les fonctions de la catégorie "Function" sont principalement liées à la manipulation et à la gestion des fonctions en JavaScript. Cela inclut des utilitaires pour la liaison de contexte, la composition de fonctions, etc.

4. **Lang :** Cette catégorie regroupe les fonctions liées à la manipulation et à l'inspection des valeurs en JavaScript. Elle comprend des opérations telles que la vérification du type, la comparaison, etc.

Les catégories

5. **Math :** La catégorie "Math" inclut des fonctions mathématiques et des utilitaires pour travailler avec des nombres.

6. **Number :** Cette catégorie est spécifiquement axée sur les opérations liées aux nombres, tels que l'arrondi, la normalisation, etc.

7. **Object :** Les fonctions de la catégorie "Object" sont dédiées à la manipulation et à la transformation d'objets en JavaScript. Cela inclut des opérations comme la fusion d'objets, la recherche de propriétés, etc.

8. **Seq :** La catégorie "Seq" concerne les opérations de séquences, notamment les chaînes de méthodes Lodash qui travaillent sur une séquence de valeurs.

9. **String :** Cette catégorie concerne les fonctions liées à la manipulation et à la transformation des chaînes de caractères.

"Array" Methods

.chunk	Crée un tableau d'éléments divisés en groupes de la taille spécifiée. Si le tableau ne peut pas être divisé de manière égale, le dernier morceau contiendra les éléments restants.
.compact	Crée un tableau en supprimant toutes les valeurs falsy. Les valeurs false, null, 0, "", undefined et NaN sont considérées comme falsy.
.concat	Crée un nouveau tableau en concaténant le tableau avec d'autres tableaux et/ou valeurs supplémentaires.
.flatten	Aplani un tableau d'un seul niveau de profondeur.
.first OR .head	Obtient le premier élément du tableau.
.join	Convertit tous les éléments du tableau en une seule chaîne de caractères séparée par le séparateur spécifié.
.last	Obtient le dernier élément du tableau.
.pull	Supprime toutes les valeurs spécifiées du tableau en utilisant SameValueZero pour les comparaisons d'égalité.
.remove	Supprime tous les éléments du tableau pour lesquels le prédicat renvoie une valeur truthy et renvoie un tableau des éléments supprimés. Le prédicat est invoqué avec trois arguments : (valeur, index, tableau).
.reverse	Inverse le tableau de telle sorte que le premier élément devienne le dernier, le deuxième élément devienne l'avant-dernier, et ainsi de suite.
.slice	renvoie un tableau, contenant une copie d'une portion du tableau d'origine, la portion est définie par un indice de début et un indice de fin (exclus).
.uniq	Crée une version du tableau sans doublons, en utilisant SameValueZero pour les comparaisons d'égalité, dans laquelle seule la première occurrence de chaque élément est conservée.
.zip	Crée un tableau d'éléments regroupés, le premier contenant les premiers éléments des tableaux donnés, le deuxième contenant les deuxièmes éléments des tableaux donnés, et ainsi de suite...

"Collection" Methods

.filter	Parcourt les éléments de la collection et renvoie un tableau de tous les éléments pour lesquels le prédicat renvoie une valeur truthy. Le prédicat est invoqué avec trois arguments : (valeur, index|clé, collection).
.flatMap	Crée un tableau aplani de valeurs en exécutant chaque élément de la collection à travers l'itérateur et en aplatissant les résultats obtenus. L'itérateur est invoqué avec trois arguments : (valeur, index|clé, collection).
.forEach OR .each	Parcourt les éléments de la collection et invoque l'itérateur pour chaque élément. L'itérateur est appelé avec trois arguments : (valeur, index|clé, collection). Les fonctions d'itérateur peuvent arrêter l'itération prématurément en retournant explicitement `false`.
.includes	Vérifie si la valeur se trouve dans la collection. Si la collection est une chaîne de caractères, elle est vérifiée pour une sous-chaîne correspondant à la valeur. Sinon, SameValueZero est utilisé pour les comparaisons d'égalité. Si fromIndex est négatif, il est utilisé comme décalage à partir de la fin de la collection.
.map	Crée un tableau de valeurs en exécutant chaque élément de la collection à travers l'itérateur. L'itérateur est appelé avec trois arguments : (valeur, index|clé, collection). De nombreuses méthodes de lodash sont sécurisées pour fonctionner en tant qu'itérateurs pour des méthodes telles que _.every, _.filter, _.map, _.mapValues, _.reject et _.some.
.reduce	Réduit la collection à une valeur qui est le résultat accumulé de l'exécution de chaque élément de la collection à travers l'itérateur, où chaque invocation successive reçoit la valeur de retour de l'itération précédente. Si un accumulateur n'est pas fourni, le premier élément de la collection est utilisé comme valeur initiale. L'itérateur est appelé avec quatre arguments : (accumulateur, valeur, index|clé, collection). De nombreuses méthodes de lodash sont sécurisées pour fonctionner en tant qu'itérateurs pour des méthodes telles que _.reduce, _.reduceRight et _.transform. Les méthodes sécurisées sont : assign, defaults, defaultsDeep, includes, merge, orderBy et sortBy.
.size	Obtient la taille de la collection en renvoyant sa longueur pour les valeurs de type tableau ou le nombre de propriétés de clé de chaîne énumérables propres pour les objets.
.some	Teste si au moins un élément du tableau passe le test implémenté par la fonction fournie. Elle renvoie un booléen indiquant le résultat du test.

"Lang" Methods

.castArray	Transforme la valeur en un tableau si ce n'est pas déjà le cas.
.isArray	Vérifie si la valeur est classifiée comme un objet de type tableau (Array).
.isBoolean	Vérifie si la valeur est classifiée comme un type primitif booléen ou un objet booléen.
.isEmpty	Vérifie si la valeur est un objet vide, une collection, une map ou un ensemble vide. Les objets sont considérés comme vides s'ils n'ont aucune propriété propre énumérable de type chaîne. Les valeurs de type tableau telles que les objets arguments, les tableaux, les buffers, les chaînes de caractères ou les collections de type jQuery sont considérées comme vides si leur longueur est de 0. De même, les maps et les sets sont considérés comme vides s'ils ont une taille de 0.
.isEqual	Effectue une comparaison en profondeur entre deux valeurs pour déterminer si elles sont équivalentes.
.isNumber	Vérifie si la valeur est classifiée en tant que nombre (primitive ou objet).
.isObject	Vérifie si la valeur appartient au type de langage "Object". Cela inclut les tableaux, les fonctions, les objets, les expressions régulières, `new Number(0)`, et `new String('')`.
.isString	vérifie si la valeur est classée comme une chaîne de caractères primitive ou un objet.
.toArray	Convertit la valeur en un tableau.
.toNumber	Convertit la valeur en un nombre.
.words	Divise la chaîne en un tableau de ses mots.

"Math" Methods

.add	Additionne deux nombres.
.ceil	Calcule le nombre arrondi vers le haut à la précision spécifiée.
.divide	Divise deux nombres.
.floor	Calcule le nombre arrondi vers le bas à la précision spécifiée.
.max	Calcule la valeur maximale d'un tableau. Si le tableau est vide ou falsy, undefined est renvoyé.
.mean	Calcule la moyenne des valeurs d'un tableau.
.min	Calcule la valeur minimale d'un tableau. Si le tableau est vide ou falsy, undefined est renvoyé.
.multiply	Multiplie deux nombres.
.round	Calcule le nombre arrondi à la précision spécifiée.
.substract	Soustrait deux nombres.
.sum	Calcule la somme des valeurs dans un tableau.

"Object" Methods

.get	Récupère la valeur située au chemin spécifié dans l'objet. Si la valeur résolue est indéfinie (undefined), la valeur par défaut (defaultValue) est retournée à sa place.
.mapValues	Crée un objet avec les mêmes clés que l'objet donné et des valeurs générées en exécutant chaque propriété énumérable propre de type chaîne de l'objet à travers l'itérateur. L'itérateur est invoqué avec trois arguments : (valeur, clé, objet).
.omit	L'opposé de _.pick ; cette méthode crée un objet composé des chemins de propriété énumérables propres et héritées de l'objet qui ne sont pas omis.
.pick	Crée un objet composé des propriétés choisies de l'objet.
.transform	Une alternative à _.reduce ; cette méthode transforme un objet en un nouvel objet d'accumulateur qui est le résultat de l'exécution de chacune de ses propriétés énumérables propres de type chaîne à travers l'itérateur. Chaque invocation peut potentiellement muter l'objet d'accumulateur. Si l'accumulateur n'est pas fourni, un nouvel objet avec le même [[Prototype]] sera utilisé. L'itérateur est invoqué avec quatre arguments : (accumulateur, valeur, clé, objet). Les fonctions d'itérateur peuvent arrêter l'itération prématurément en retournant explicitement `false`.

"String" Methods

.capitalize	Convertit le premier caractère de la chaîne en majuscule et le reste en minuscules.
.deburr	Retire les marques diacritiques en convertissant les lettres de l'ensemble Latin-1 Supplement et Latin Extended-A en lettres de base de l'alphabet latin.
.lowerCase	Convertit la chaîne en minuscules, en considérant les mots séparés par des espaces.
.replace	Remplace les correspondances du motif dans la chaîne par la valeur de remplacement.
.split	Divise la chaîne en utilisant le séparateur spécifié.
.startsWith	Vérifie si la chaîne commence par la chaîne cible donnée.
.toLower	Convertit la chaîne entière en minuscules, de la même manière que la méthode `String#toLowerCase()`.
.toUpper	Convertit la chaîne entière en majuscules, de la même manière que la méthode `String#toUpperCase()`.
.trim	Supprime les espaces ou les caractères spécifiés en début et en fin de chaîne.
.unescape	Cette méthode convertit les entités HTML &, <, >, " et ' dans la chaîne en leurs caractères correspondants.
.upperCase	Convertit les mots de la chaîne, séparés par des espaces, en majuscules.
.words	Divise la chaîne en un tableau de ses mots.

Utiliser un map

La fonction .map() sert à créer un tableau de valeurs en parcourant chaque élément d'une collection donnée, ici 'users'.

```
users = [
  { 'user': 'barney' },
  { 'user': 'fred' },
  { 'user': 'bob' }
]
```

➡ .map('user') ➡ ['barney', 'fred', 'bob']

Dans ce cas, la fonction .map() va "simplement" parcourir les éléments du tableau d'objet 'users', récupérer toutes les valeurs associées à la clé que l'on a déclaré ('user') et restituer toutes ces valeurs dans un tableau.

Utiliser un map

La fonction ·map() itérant sur tous les éléments d'une collection, on peut lui associer des fonctions pour transformer tous les éléments de la collection

["A", "N", "n/a", "C", "n/a", "A"]

·map(item=>item·replace("n/a","Inconnu"))

Ici ·map() va parcourir tous les élements, où pour chaque élément rencontré correspondant à "n/a" il effectuera un ·replace() et enfin il retournera tous les éléments dans ce tableau :

["A", "N", "Inconnu", "C", "Inconnu", "A"]

Retour à l'exercice

Faire un zip de tableaux

TableauA : ["A","B","C","D","E"]

TableauB : [1,2,1,9,0]

·zip(self·value·TableauA, self·value·TableauB)

Renvoie une matrice où les éléments sont associés en fonction de leur position dans le tableau :

[["A", 1], ["B", 2], ["C", 1], ["D", 9], ["E", 0]]

Faire un zip de tableaux

! Attention !

Evitez de passer des ·uniq() ou ·compact() sur l'un des tableaux·

Si les tableaux n'ont pas le même nombre d'éléments la matrice sera erronée·

Si on applique ·uniq() sur TableauB: [1,2,1,9,0] celui-ci n'aura plus que quatre éléments: [1,2,9,0] alors ·zip() renverra :

[["A", 1], ["B", 2], ["C", 9], ["D", 0], ["E", undefined]]

À la place de :

[["A", 1], ["B", 2], ["C", 1], ["D", 9], ["E", 0]]

Retour à l'exercice

Utiliser la fonction reduce

La fonction .reduce() est utilisée pour réduire une collection (un tableau ou un objet) à une seule valeur en appliquant une fonction de réduction à chaque élément de la collection. Cette fonction de réduction prend généralement deux arguments : l'accumulateur et la valeur courante, et renvoie une nouvelle valeur qui devient l'accumulateur pour l'itération suivante.

```
.reduce((result, value) => {return result + value;}, 0)
```

Dans cette fonction l'accumulateur est "result", il représente la valeur accumulée.
Le second argument "value" sera la valeur courante à chaque étape de l'iteration.
Ceci est la fonction de réduction, qui signifie qu'à chaque iteration on additionne le résultat accumulé avec la valeur courante.
0 est la valeur initiale de notre accumulateur.

Utiliser la fonction reduce

Voyons étape par étape comment fonctionne notre ·reduce(), avec comme données un tableau de quatre éléments : [2,3,5,7] qui donnera donc lieu à quatre itérations·

- 1ère itération : la valeur initiale de l'accumulateur est 0, on lui ajoute la valeur du 1er élément du tableau (2), soit 0 + 2 = 2·
  2 est retourné comme nouvelle valeur accumulée (result)·

- 2ème itération : 2 est désormais la valeur accumulée actuelle, on lui ajoute la valeur courante de la 2ème itération, en l'occurrence 3· Ce qui nous renvoie l'accumulateur de la prochaine itération : 2 + 3 = 5

- 3ème itération : on additionne la valeur accumulée 5 à la valeur courante, 5 aussi· 10 est le résultat retourné : 5 + 5 = 10

- 4ème et dernière itération : la valeur accumulée actuelle 10 est additionnée avec la valeur courante 7· 17 est donc retourné· Le résultat final de la fonction est donc une valeur unique, 17, en lieu et place d'un tableau·

Utiliser la fonction reduce

Voyons en détail un autre cas d'utilisation de .reduce()

```
.reduce((result, item) =>{if (item.slice(1).some(element => element === "INSHS (P)")) {result.push(item[0])} return result}, [])
```

"result" représente toujours la valeur accumulée ici.

Le second argument sera la valeur courante à chaque étape de l'iteration, nous l'avons nommé item. Comme nous appliquons notre .reduce() à une matrice, les valeurs courantes seront des tableaux.

La fonction de réduction, basée sur une condition et que nous allons détailler.

La valeur initiale de notre accumulateur qui est un tableau vide et qui sera utilisé pour stocker les éléments qui satisfont la condition.

Utiliser la fonction reduce

Décomposons cette fonction :

```
=>{if (item·slice(1)·some(element => element === "INSHS (P)"))
{result·push(item[0])} return result}, []
```

On avait vu dans l'exemple précédent que l'accumulateur était modifié à chaque itération· Or ici {if... {result·push(item[0])} return result} signifie qu'une nouvelle valeur accumulée sera retournée si la condition est satisfaite· Sinon, on retourne l'accumulateur tel quel sans modification·

La condition signifie qu'à chaque itération, la valeur courante - qui sera un tableau - sera découpée en 2 portions par invocation de ·slice(1) (un rnsr d'un côté, les instituts de l'autre)· La seconde partie du tableau est ensuite parcourue par ·some() qui va vérifier si un des éléments est "INSHS (P)"· S'il en trouve un ou plus alors la condition est vraie·

Le rnsr (item[0]) est alors injecté dans notre tableau vide par ·push()·

Utiliser la fonction reduce

Pour [["198822446E","DGD-S (P)"]] :

1 seule itération se fera. On démarre donc avec l'accumulateur [], le tableau est découpé, la condition n'est pas satisfaite, on renvoie l'accumulateur []

Pour [["201822728F","INSHS (P)"],["199612367P","INSHS (P)","INEE (S)"]]

On démarre la première itération avec l'accumulateur [], le tableau est slicé, on parcourt la deuxième partie où le seul élément présent satisfait la condition. Le premier élément de la valeur courante est alors injecté dans []. La nouvelle valeur accumulée devenant ["201822728F"]

Dernière itération, avec la valeur accumulée ["201822728F"], on teste le tableau qui vérifie lui aussi la condition. "199612367P" est injecté dans le tableau et le résultat final est renvoyé : ["201822728F","199612367P"]

Retour à l'exercice

Les opérateurs JavaScript

=	affectation	a = 3
==	égalité	3 == "3" // true
===	égalité stricte	3 === "3" // false
!	NON logique (NOT)	!true // false
!=	inégalité	3 != "3" // false
!==	inégalité stricte	3 !== "3" // true
&&	ET logique (AND)	Si a=true et b=false alors a &&b // false
||	OU logique (OR)	a||b
?	conditionnel (IF-ELSE)	Si age = 3; age >= 18 ? "majeur" : "mineur" // "mineur"
>	supérieur à	Si a = 3 et b = 5; a > b // false
>=	supérieur ou égal à	a >= b // false
<	inférieur à	a < b // true
<=	inférieur ou égal à	a <= b // true

EZS

EZS est un cadre logiciel spécifiquement conçu pour Lodex et qui permet de traiter d'importants flux de données, morceau par morceau. Traitant des objets JavaScript (JSON), il propose une foule d'instructions permettant de croiser, compter, trier, comparer, supprimer ces mêmes flux.

Sujet vaste, nous évoquerons ici les seules fonctions que l'on peut écrire dans un enchaînement de fonctions avec la syntaxe JavaScript dans le mode enrichissement.

- ***'append'*** : Ajoute un suffixe dans une chaîne. `("lodex").append(".fr")` renvoie "lodex.fr".

- ***'compute'*** : Compile des expressions de filtrage. Ex `` `age < 18 and qi > 130` `` vérifie si l'âge est inférieur à 18 et le QI supérieur à 130 et renvoie un booléen.

EZS

- ***'fix'*** : Créé une constante permettant de déclarer en arguments des valeurs fixes et/ou dynamiques.

  - `fix("ok")` renvoie "ok"

  - `fix({doi: self.value.DOI, titre: self.value.Title})` renvoie un objet littéral JavaScript avec comme propriétés "doi" et "titre" auxquelles sont attribuées les valeurs dynamiques souhaitées.

- ***'prepend'*** : Ajoute un préfixe dans une chaîne. `("lodex.fr").prepend("www.")` renvoie "www.lodex.fr"

- ***'self'*** : Renvoie tel quel l'argument déclaré.

Pour aller plus loin

- La documentation de Lodex et notamment des recettes d'enrichissements en Lodash.
- La documentation officielle de Lodash.
- Le language EZS et les différentes instructions que l'on peut utiliser.
- Une série de six livres écrits par Kyle Simpson et qui constituent des précis sur les différents aspects du langage Javascript.
- L'outil EZS Playground qui permet de tester ses scripts.
- Un Playground Lodash qui permet aussi de tester ses scripts, mais cette fois en dehors du contexte Lodex/EZS.
- Devdocs.io : site de référence pour JavaScript et Lodash et de nombreux autres langages.