# RFC 8610 : Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 13 juin 2019

Date de publication du RFC : Juin 2019

https://www.bortzmeyer.org/8610.html

---

Le format de données binaire CBOR, normalisé dans le RFC 8949 [1], commence à avoir un certain succès. Il lui manquait juste un langage de schéma, permettant de décrire les données acceptables (comme Relax NG ou XML Schema pour XML, ou comme le projet - abandonné - JCR (≪ "*JSON Content Rules*" ≫) pour JSON). C'est désormais fait dans ce RFC, qui normalise le langage CDDL, ≪ "*Concise Data Definition Language*" ≫.

La section 1 de notre RFC résume le cahier des charges : CDDL doit permettre de décrire sans ambiguïté un fichier CBOR acceptable pour un usage donné, il doit être lisible et rédigeable par un humain, tout en étant analysable par un programme, et doit permettre la validation automatique d'un fichier CBOR. Autrement dit, étant donné une description en CDDL en `schema.cddl` et un fichier CBOR en `data.cbor`, il faut qu'on puisse développer un outil `validator` qui permettra de lancer la commande `validator data.cbor schema.cddl` et qui dira si le fichier CBOR est conforme au schéma ou pas. (Un tel outil existe effectivement, il est présenté à la fin de cet article.) Comme CBOR utilise un modèle de données très proche de celui de JSON, CDDL peut (même si ce n'est pas son but principal) être utilisé pour décrire des fichiers JSON, ce que détaille l'annexe E du RFC, consacrée à l'utilisation de CDDL avec JSON (il y a quelques subtilités à respecter).

(Attention, il ne faut pas confondre notre CDDL avec la licence ayant le même acronyme.)

La section 2 de notre RFC explique les éléments de base d'un schéma CDDL. On y trouve les classiques nombres, booléens, chaînes de caractères, correspondant aux éléments identiques en CBOR. Pour

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc8949.txt`

les structures plus compliquées (tableaux et "*maps*", c'est-à-dire dictionnaires, ce qu'on nomme objets en JSON), CDDL ne fournit qu'un seul mécanisme, le **groupe**. Un groupe est une liste de doublets {nom, valeur}, le nom pouvant être omis si on décrit un tableau. Avec ce concept de groupe, CDDL permet également de décrire ce que dans d'autres langages, on appelerait struct ou enregistrement.

La liste est encadrée par des parenthèses. Chaque donnée décrite en CDDL a un type, par exemple `bool` pour un booléen, `uint` pour un entier non signé ou `tstr` pour une chaîne de caractères. La définition indique également quel type majeur CBOR (RFC 8949, section 3.1) va être utilisé pour ce type CDDL. `uint` est évidemment le type majeur 0, `bool` est le type majeur 7, etc. (D'ailleurs, vous pouvez aussi créer des types en indiquant le type majeur CBOR, ce qui donne une grande liberté, et la possibilité d'influencer la sérialisation.) Une liste des types et valeurs prédéfinies (comme `false` et `true`) figure dans l'annexe D de notre RFC.

Voici un groupe à qui on donne le nom `pii` :

```
pii = (
       age: uint,
       name: tstr,
       employer: tstr
)
```

Et ici une donnée `person` est définie avec ce groupe :

```
person = {
        pii
}
```

Comme `person` est défini avec des accolades, ce sera un dictionnaire ("*map*"). Le même groupe `pii` aurait pu être utilisé pour définir un tableau, en mettant entre crochets (et, dans ce cas, les noms seraient ignorés, seule la position compte).

On peut définir une donnée en utilisant un groupe et d'autres informations, ici, `person` et `dog` ont les attributs de `identity` et quelques uns en plus :

```
person = {
     identity,
     employer: tstr
}

dog = {
     identity,
     leash-length: float
}

identity = (
    age: 0..120, ; Ou "uint" mais, ici, on utilise les intervalles
    name: tstr
)
```

La syntaxe `nom: valeur` est en fait un cas particulier. La notation la plus générale est `clé => valeur`. Comme CBOR (contrairement à JSON) permet d'avoir des clés qui ne sont pas des chaînes de caractères, la notation avec le deux-points est là pour ce cas particulier, mais courant, où la clé est une chaîne de caractères. (`age: int` et `"age" => int` sont donc équivalents.)

Un autre exemple permet d'illustrer le fait que l'encodage CBOR en tableau ou en dictionnaire va dépendre de la syntaxe utilisée en CDDL (avec en prime les commentaires, précédés d'un point-virgule) :

```
Geography = [
           city           : tstr,
           gpsCoordinates : GpsCoordinates,
]

GpsCoordinates = {
           longitude      : uint,            ; multiplied by 10^7
           latitude       : uint,            ; multiplied by 10^7
}
```

Dans le fichier CBOR, `GpsCoordinates` sera un dictionnaire ("*map*") en raison de l'utilisation des accolades, et `Geography` sera un tableau (les noms `city` et `gpsCoordinates` seront donc ignorés).

Un champ d'un groupe peut être facultatif, en le faisant précéder d'un point d'interrogation, ou bien répété (avec une astérisque ou un plus) :

```
apartment = {
     kitchen: size,
     + bedroom: size,
     ? bathroom: size
   }

size = float
```

Dans cet appartement, il y a exactement une cuisine, au moins une chambre et peut-être une salle de bains. Notez que l'outil cddl, présenté plus loin, ne créera pas d'appartements avec plusieurs chambres. C'est parce que CBOR, contrairement à JSON (mais pas à I-JSON, cf. RFC 7493, section 2.3), ne permet pas de clés répétées dans une "*map*". On a ici un exemple du fait que CDDL peut décrire des cas qui ne pourront pas être sérialisés dans un format cible donné.

Revenons aux types. On a également le droit aux énumérations, les valeurs étant séparées par une barre oblique :

```
attire = "bow tie" / "necktie" / "Internet attire"

protocol = 6 / 17
```

C'est d'ailleurs ainsi qu'est défini le type booléen (c'est prédéfini, vous n'avez pas à taper cela) :

```
bool = false / true
```

On peut aussi choisir entre groupes (et pas seulement entre types), avec deux barres obliques.

Et l'élement racine, on le reconnait comment ? C'est simplement le premier défini dans le schéma. À part cette règle, CDDL n'impose pas d'ordre aux définitions. Le RFC préfère partir des structures de plus haut niveau pour les détailler ensuite, mais on peut faire différemment, selon ses goûts personnels.

Pour les gens qui travaillent avec des protocoles réseau, il est souvent nécessaire de pouvoir fixer exactement la représentation des données. CDDL a la notion de contrôle, un contrôle étant une directive donnée à CDDL. Elle commence par un point. Ainsi, le contrôle `.size` indique la taille que doit prendre la donnée. Par exemple (`bstr` étant une chaîne d'octets) :

```
ip4 = bstr .size 4
ip6 = bstr .size 16
```

Un autre contrôle, `.bits`, permet de placer les bits exactement, ici pour l'en-tête TCP :

```
tcpflagbytes = bstr .bits flags
                      flags = &(
                        fin: 8,
                        syn: 9,
                        rst: 10,
                        psh: 11,
                        ack: 12,
                        urg: 13,
                        ece: 14,
                        cwr: 15,
                        ns: 0,
) / (4..7) ; data offset bits
```

Les contrôles existants figurent dans un registre IANA <https://www.iana.org/assignments/cddl/cddl.xml#cddl-control-operators>, et d'autres pourront y être ajoutés, en échange d'une spécification écrite (cf. RFC 8126).

La section 3 du RFC décrit la syntaxe formelle de CDDL. L'ABNF (RFC 5234) complet est en annexe B. CDDL lui-même ressemble à ABNF, d'ailleurs, avec quelques changements comme l'autorisation du point dans les noms. Une originalité plus fondamentale, documentée dans l'annexe A, est que la grammaire utilise les PEG et pas le formalisme traditionnel des grammaires génératives.

La section 4 de notre RFC couvre les différents usages de CDDL. Il peut être utilisé essentiellement pour les humains, une sorte de documentation formelle de ce que doit contenir un fichier CBOR. Il peut aussi servir pour écrire des logiciels qui vont permettre une édition du fichier CBOR guidée par le schéma (empêchant de mettre des mauvaises valeurs, par exemple, mais je ne connais pas de tels outils, à l'heure actuelle). CDDL peut aussi servir à la validation automatique de fichiers CBOR. (Des exemples sont donnés plus loin, avec l'outil cddl.) Enfin, CDDL pourrait être utilisé pour automatiser une partie de la génération d'outils d'analyse de fichiers CBOR, si ce format continue à se répandre.

Un exemple réaliste d'utilisation de CDDL est donné dans l'annexe H, qui met en œuvre les « reputons » du RFC 7071. Voici le schéma CDDL (en ligne sur `https://www.bortzmeyer.org/files/`

`reputon.cddl`). Un autre exemple en annexe H est de réécrire des règles de l'ancien projet JCR (cf. "*Internet draft*" `draft-newton-json-content-rules`) en CDDL.

Quels sont les RFC et futurs RFC qui se servent de CDDL ? CDDL est utilisé par le RFC 8007 (son annexe A), le RFC 8152 et le RFC 8428. Il est également utilisé dans des travaux en cours comme le format C-DNS (RFC 8618), sur lequel j'avais eu l'occasion de travailler lors d'un hackathon <`https://www.bortzmeyer.org/c-dns-tests.html`>. Autre travail en cours, le système GRASP (RFC 8990) et dans OSCORE (RFC 8613). En dehors du monde IETF, CDDL est utilisé dans Web Authentication <`https://www.w3.org/TR/webauthn/#biblio-cddl`>.

Un outil <`https://rubygems.org/gems/cddl`>, décrit dans l'annexe F du RFC, a été développé pour générer des fichiers CBOR d'exemple suivant une définition CDDL, et pour vérifier des fichiers CBOR existants. Comme beaucoup d'outils modernes, il faut l'installer en utilisant les logiciels spécifiques d'un langage de programmation, ici Ruby :

```
% gem install cddl
```

Voici un exemple, pour valider un fichier JSON (il peut évidemment aussi valider du CBOR, rappelez-vous que c'est presque le même modèle de données, et que CDDL peut être utilisé pour les deux) :

```
% cddl person.cddl validate person.json
%
```

Ici, c'est bon. Quand le fichier de données ne correspond pas au schéma (ici, le membre `foo` n'est pas prévu) :

```
% cat person.json
{"age": 1198, "foo": "bar", "name": "tic", "employer": "tac"}

% cddl person.cddl validate person.json
CDDL validation failure (nil for {"age"=>1198, "foo"=>"bar", "name"=>"tic", "employer"=>"tac"}):
["tac", [:prim, 3], nil]
```

C'est surtout quand le schéma lui-même a une erreur que les messages d'erreur de l'outil cddl sont particulièrement mauvais. Ici, pour un peu d'espace en trop :

```
% cddl person.cddl generate
*** Look for syntax problems around the %%% markers:
%%%person = {
       age: int,
       name: tstr,
       employer: tstr,%%%             }
*** Parse error at 0 upto 69 of 93 (1439).
```

Et pour générer des fichiers de données d'exemple ?

```
% cat person.cddl
person = {
       "age" => uint, ; Or 'age: uint'
       name: tstr,
       employer: tstr
       }

% cddl person.cddl generate
{"age": 3413, "name": "tic", "employer": "tac"}
```

Ce format est du JSON mais c'est en fait le profil « diagnostic » de CBOR, décrit dans la section 8 du RFC 8949. (`cddl person.cddl json-generate` fabriquerait du JSON classique.) On peut avoir du CBOR binaire après une conversion avec les outils d'accompagnement <https://github.com/cabo/cbor-diag> :

```
% json2cbor.rb person.json > person.cbor
```

CBOR étant un format binaire, on ne peut pas le regarder directement, donc on se sert d'un outil spécialisé (même dépôt que le précédent) :

```
% cbor2pretty.rb person.cbor
a3                      # map(3)
   63                   # text(3)
      616765            # "age"
   19 0d55              # unsigned(3413)
   64                   # text(4)
      6e616d65          # "name"
   63                   # text(3)
      746963            # "tic"
   68                   # text(8)
      656d706c6f796572  # "employer"
   63                   # text(3)
      746163            # "tac"
```

Et voilà, tout s'est bien passé, et le fichier CBOR est valide :

```
%  cddl person.cddl validate person.cbor
%
```