Décoder une énumération depuis une API en ReScript avec decco

rescript

mardi 16 février 2021

Composition d'un décodeur

Un décodeur decco doit être composé de **4 éléments** :

- une fonction `encoder` qui gère la sérialisation
- une fonction `decoder` qui gère le désérialisation
- une variable `codec` contenant ces fonctions (sous forme de tuple)
- un type `t` qui correspondra au type que l'on souhaite générer/décoder

Gérer la sérialisation

Quand je dois travailler avec une énumération de string, j'aime bien utiliser la directive `@deriving(jsConverter)` qui génère automatiquement les fonctions permettant de permuter entre une string et un type. Voici un exemple :

```
@deriving(jsConverter)
type brand = [
  | #sony
  | #microsoft
  | #toyota
  | #apple
];

Js.log(brandToJs(#microsoft)); /* log "microsoft" */
brandFromJs("microsoft")->Belt.Option.forEach(v => Js.log(v)); /* log
```

Petit détail très intéressant concernant `brandFromJs`, la fonction retourne un type `option<string>` car on peut très bien lui passer une valeur qui n'existe pas dans l'énumération et donc retourner `None` le cas échéant.

Gardons donc ce type brand et écrivons la fonction de sérialisation :

```
@deriving(jsConverter)
type brand = [
  | #sony
  | #microsoft
  | #toyota
  | #apple
];

let encoder: Decco.encoder<brand> = (brand: brand) => {
  brand->brandToJs->Decco.stringToJson;
};
```

Ici, j'ai déclaré les types explicitement afin de rendre l'exemple le plus compréhensible possible mais vous pouvez très bien laisser l'inférence faire son job !

Concernant cette fonction `encoder`, elle prend un paramètre du type que l'on souhaite encoder et on transforme celui-ci en `string` afin de pouvoir appeler la fonction `Decco.stringToJson` qui va faire la conversion en JSON.

Voilà pour la sérialisation ! Rien de plus n'est nécessaire, nous pouvons passer à la désérialisation !

Gérer la désérialisation

C'est presque la même chose que la sérialisation sauf qu'il faut gérer les cas d'erreurs en plus :

```
let decoder: Decco.decoder<brand> = json => {
  switch (json->Decco.stringFromJson) {
  | Belt.Result.Ok(v) => switch (v->brandFromJs) {
      | None => Decco.error(~path="", "Invalid enum " ++ v, json)
```

```
      | Some(v) => v->Ok
    }
  | Belt.Result.Error(_) as err => err
  };
};
```

Dans cet exemple, il faut juste noter que `Decco.stringFromJson` retourne un type `Belt.Result.t` et que pour retourner une erreur il faut utiliser la fonction `Decco.error`.

Et le reste

Il reste maintenant les 2 variables à créer qui se présenteront comme ceci :

```
let codec: Decco.codec(brand) = (encoder, decoder);

[@decco]
type t = [@decco.codec codec] brand;
```

Nous devons impérativement associer les bons types et nous y sommes, nous avons notre propre sérialiseur ! Voici donc notre code regroupé dans un module :

```
module BrandCodec = {
  @deriving(jsConverter)
  type brand = [
    | #sony
    | #microsoft
    | #toyota
    | #apple
  ]

  let encoder: Decco.encoder<brand> = (brand: brand) => {
    brand->brandToJs->Decco.stringToJson;
  }

  let decoder: Decco.decoder<brand> = json => {
    switch (json->Decco.stringFromJson) {
    | Belt.Result.Ok(v) => switch (v->brandFromJs) {
        | None => Decco.error(~path="", "Invalid enum " ++ v, json)
```

```
        | Some(v) => v->Ok
      }
    | Belt.Result.Error(_) as err => err
    }
  }

  let codec: Decco.codec<brand> = (encoder, decoder)

  @decco
  type t = @decco.codec(codec) brand
};
```

Nous pouvons maintenant utiliser ce module dans un record en utilisant le type `t` :

```
/*...*/

@decco
type console = {
  id: string,
  name: string,
  brand: BrandCodec.t
};
```