Elixir

Introduction

Elixir a été créé par José Valim, un acteur majeur l'écosystème Ruby et core contributeur au framework Ruby on Rails. En 2012, Valim a pour souhait de créer un langage aussi accessible que Ruby avec de meilleures performances et surtout une résilience à toute épreuve. C'est pour cette raison que Valim a décidé de créer Elixir sur la machine virtuelle Erlang, étant réputée pour les qualités souhaitées précédemment citées. Elixir simplifie la syntaxe du langage Erlang tout en gardant ses avantages.

Justification du langage

Chris McCord, créateur du framework Phoenix (framework web en Elixir le plus utilisé) a déclaré en parlant d'Elixir : « What kind of business could I build if what before took a hundred servers could today take two servers? I think that's enabling a lot of innovation. » Cette phrase, bien que non sans intérêt, est intéressante car plusieurs entreprises ayant migré vers Elixir donnent le même retour : le nombre de machines nécessaires au fonctionnement de leur application a pu largement diminuer. Alors comment cela est-ce possible ? Et bien notamment grâce aux principaux avantages du langage :

Elixir est conçu pour la concurrence, la performance et la robustesse, en partie grâce au fait qu'il se repose sur la machine virtuelle d'Erlang (langage inventé pour gérer les infrastructures téléphoniques grandissantes dans les années 80). Son système de processus légers que nous verrons par la suite permet de réaliser un nombre incalculable d'opérations simultanées en limitant au plus la latence sans bloquer un thread principal par exemple.

Mais dans ce cas-là, pourquoi ne pas tout simplement utiliser Erlang pourrait-on se dire ? Elixir propose une syntaxe moderne, facile à lire et à écrire, inspirée par le Ruby et le Python. Ce qui en fait à la fois un langage accessible tout en restant performant grâce à la VM Erlang.

C'est donc majoritairement ces deux aspects qui font que, selon moi, Elixir est un langage intéressant et mérite que l'on se penche dessus.

Concepts

Les bases

Elixir est un langage dynamiquement typé tout en restant fortement typé (à l'inverse de JavaScript par exemple). Il n'est donc pas possible de réaliser une opération du genre :

```
iex> 1 + "2"
(ArithmeticError) bad argument in arithmetic expression: 1 + "2"
```

L'annotation des types, appelée « typespecs » est possible mais facultative. Si le type n'est pas spécifié, il est inféré par le compilateur. Il est également possible de créer ses propres types.

Elixir est un langage fonctionnel, tous ses types de données sont immuables. Si nous essayons de réaffecter une variable existante, nous réassignons le label de la variable à une nouvelle

valeur, pas la variable en elle-même qui sera supprimée par le Garbage Collector lorsqu'elle n'est plus utilisée.

Bien entendu, comme le langage est fonctionnel les boucles traditionnelles ne sont pas disponibles. Il faut utiliser des fonctions d'ordres supérieures (comme map ou reduce) ou la récursivité.

Pour finir, la syntaxe du langage est clairement haut niveau et moderne. Des opérateurs comme le « pipe operator » existent afin de rendre le code très lisible à la façon des lambdas en Java :

```
" John Smythe"
  |> String.trim()
  |> String.downcase()
  |> String.replace(" ", "-")
"john-smythe"
```

Process

L'intégralité du code Elixir est exécuté dans des processus. Il ne faut pas confondre les processus Elixir avec les processus du système d'exploitation. En effet, les processus Elixir sont bien plus légers, même plus légers que les threads utilisés dans de nombreux autres langages. Il n'est donc pas rare d'avoir des centaines voire des milliers de processus simultanément dans un programme Elixir.

Afin de pouvoir accéder au même état entre les différents processus (state), une abstraction aux simples processus est disponible : les Agents.

Voici un exemple de compteur partagé, bien entendu il est également possible de partager des données bien plus complexes comme des maps ou autre.

```
defmodule Counter do
  use Agent

  def start_link(initial_value) do
    Agent.start_link(fn -> initial_value end, name: __MODULE__)
  end

  def value do
    Agent.get(__MODULE__, fn x -> x end)
  end

  def increment do
    Agent.update(__MODULE__, fn x -> x + 1 end)
  end
end
```

Et voici son utilisation :

```
iex(1)> Counter.start_link(0)
{:ok, #PID<0.114.0>}
iex(2)> Counter.value()
0
iex(3)> Counter.increment()
:ok
iex(4)> Counter.increment()
:ok
iex(5)> Counter.value()
2
```

La ligne « use Agent » sert à injecter le code nécessaire à l'Agent pour bien fonctionner. Elle rajoute notamment des méthodes permettant de superviser l'Agent. Le nom du module (Counter) est utilisé via la variable __MODULE__. Les méthodes get et update de la classe Agent reçoivent comme paramètre la valeur actuelle du state, libre à nous d'ensuite l'utiliser.

Fail fast / Let it crash

La philosophie d'Elixir est assez différente de la majorité des autres langages de programmation. L'idée, grâce aux processus indépendants les uns des autres, est de laisser planter toute erreur inattendue. Il existe une syntaxe comparable aux try catch des autres langages (try rescue) mais celle-ci n'est pas souvent utilisée. Il est plus courant de laisser planter dans un processus qui a pour rôle de superviser et de redémarrer un nouveau processus en cas de problème.

De plus, Elixir ne lance pas d'erreurs dans des contextes où il est courant de le faire dans d'autres langages. Un exemple pourrait être la lecture et l'écriture dans un fichier. Elixir nous renvoie un tuple contenant des informations si la lecture ou l'écriture s'est bien passée :

```
iex> File.read("hello")
{:error, :enoent}
iex> File.write("hello", "world")
:ok
iex> File.read("hello")
{:ok, "world"}

# Ex avec pattern matching:
iex> case File.read("hello") do
...>   {:ok, body} -> IO.puts("Success: #{body}")
...>   {:error, reason} -> IO.puts("Error: #{reason}")
...> end
```

Autres particularités

Documentation

Elixir traite la documentation comme « first-class citizen » en mettant à disposition des annotations comme @doc ou @moduledoc pour directement documenter son code.

De plus, tout élément du langage de base dispose de sa propre documentation. Il est donc possible d'y accéder dans l'environnement interactif via la commande h suivie de ce qui nous intéresse :

```
iex(4)> h trunc
   def trunc(number)
  @spec trunc(number()) :: integer()
guard: true
Returns the integer part of number.
Allowed in guard tests. Inlined by the compiler.
## Examples
    iex> trunc(5.4)
    5
    iex> trunc(-5.99)
    -5
```

Atoms

Les atomes sont des constantes avec comme valeur leur propre nom. Ils sont l'équivalents des symboles dans d'autres langages (ex : JavaScript). La différence est que ceux-ci doivent commencer par le caractère « : » et sont grandement utilisés dans le langage. Nous avons pu voir précédemment que des fonctions renvoyaient les atomes :ok ou :error afin de pouvoir être facilement comparés. Ils évitent notamment l'utilisation de chaînes de caractères en dur ou de nombres indiquant des états peu compréhensibles.

Opérateur =

Pour l'anecdote, l'opérateur = n'est pas un simple opérateur d'affectation en Elixir. Il est d'ailleurs appelé « match operator » car il effectue un pattern matching derrière les décors :

```
iex> [a, b] = [1, 2]
[1, 2]
iex> a
1
iex> b
2
```

Lorsque l'on fait une simple opération `x = 3`, le pattern matching va assigner 3 à x car le pattern matching est respecté.

Compatibilité Erlang

Les modules Erlang sont directement disponibles dans le code Elixir via des atomes. Dans l'exemple qui suit, le module crypto est appelé pour créer un hash, qui est un module Erlang :

```
iex> Base.encode16(:crypto.hash(:sha256, "Elixir"))
"3315715A7A3AD57428298676C5AE465DADA38D951BDFAC9348A8A31E9C7401CB"
```

Définition du projet

L'idée est de mettre en pratique certains des avantages principaux du langage :

1. La concurrence
2. Son écosystème autour du web, notamment grâce au framework Phoenix, spécialisé dans le temps réel et les sockets.
3. Sa faible latence, ce qui est très bénéfique dans le cas d'un serveur web

Pour ce faire, je souhaite réaliser une application web interactive entre les utilisateurs. L'idée et de faire un canvas partagé entre les utilisateurs comme pour une application de prise de de dessin collaborative. Cela semble bien adaptée car chaque point de chaque ligne de chaque utilisateur doit être envoyé au serveur centralisé pour synchroniser en temps réel tous les utilisateurs. Ce qui demande des ressources assez conséquentes. Cette fonctionnalité se ferait via des connexions WebSockets, un aspect bien mis en avant par le framework web Phoenix.

Le but principal est de générer un trafic important afin de pouvoir vraiment tester les avantages énoncés du langage. Pour ce faire, je pense utiliser un logiciel comme artillery afin d'avoir une vraie montée en charge et pas uniquement quelques utilisateurs en testant à la main.

En savoir plus

- Apprendre les bases d'Elixir grâce au très bon livre d'introduction : https://elixir-lang.org/getting-started/introduction.html
- Documentation officielle : https://hexdocs.pm/elixir/