420-5DD-HY
Autodocumentation et tests

Stéphane Denis

Aujourd'hui

- Révision des notions de modules et d'attributs
- ExDoc : outil de génération de documentation et tests
- ExUnit : Tests automatisés

ATTRIBUTS DE MODULES

- On les reconnait par leur nom débutant en @
- C'est des métadonnées qui disparaissent à la compilation
- Il y a 3 usages
  - Annoter un module pour fournir des informations à l'usager ou la VM
  - Définir des constantes (substitués par leur valeur au moment de la compilation)
  - Stocker des données temporairement pendant la compilation

- On s'intéressera ici aux deux attributs qui concernent la documentation
  - `@moduledoc` Pour la documentation spécifique au module
  - `@doc` Pour la documentation de la fonction (ou macro) qui suit immédiatement la ligne/bloc

- Gardez en tête que les attributs déjà définis sont généralement des ***fonctions***

ExDoc

- **ExDoc** est l'outil qui permet de générer la documentation d'un projet à partir des informations contenues dans les attributs @moduledoc et @doc
- La documentation principale et secondaire (fichiers de doc) est fait avec la notation MarkDown
- La documentation peut être générée en format HTML ou EPUB
  - https://hexdocs.pm/ex_doc
  - https://github.com/elixir-lang/ex_doc
- Ouvrir mix.exs pour y ajouter la dépendance dans la fonction ***def deps :***
  `{:ex_doc, "~> 0.27", only: :dev, runtime: false}`
  → mix deps.get
- On peut ensuite utiliser la commande **mix docs** en PowerShell pour partir la génération. La documentation est générée dans le dossier **doc** du projet.
  voir aussi **mix help docs** pour l'ensemble des options

Plus d'options

- extras: pour ajouter des fichiers
- output: pour changer l'endroit où seront générés les fichiers. Pour vos projets, ça doit aller dans le dossier /docs de votre repo GitHub. C'est le dossier qu'on va publier sur le serveur GitHub.
- language: permet de choisir la langue par défaut de la documentation.
  Déclaration seulement, pas de gabarit multilingue en ce moment.

```elixir
def project do
  [
    app: :demo_exdoc,
    version: "0.1.0",
    elixir: "~> 1.14",
    start_permanent: Mix.env() == :prod,
    deps: deps(),
    # nouveau code à ajouter
    name: "DemoExdoc",
    description: "Demo Exdoc",
    docs: [
      extras: ["README.md"],
      language: "fr",
      output: "../doc"
    ]
  ]
end
```

TESTS UNITAIRES

1. Approche pour les **cas triviaux**
   - Les tests valident aussi le **code inclus dans la documentation (DocTest)**
2. Pour les tests plus classiques : ExUnit
   https://hexdocs.pm/ex_unit/ExUnit.html
   - Dossier **test**
   - Modules avec suffixe ...Test
     - Préconditions
     - Action
     - Assertion
   - Ici, on a les résultats d'un test qui échoue
     La **partie de gauche de l'équation** donne 13, mais on a 10 comme résultat attendu dans la **partie de droite de l'équation**

```
Comparison (using ==) failed in:
code:  assert some_fun() == 10
left:  13
right: 10
```

   - Toute la variété des assertions est disponible. Pour l'égalité simple, on peut se contenter des exemples (intégrés à la documentation)

Préconditions de test (Setup)

- le contexte permet de recevoir les informations qui ont été initialisés au Setup

```elixir
defmodule KVTest do
  use ExUnit.Case

  setup do
    {:ok, pid} = KV.start_link()
    {:ok, pid: pid}
  end

  test "stores key-value pairs", context do
    assert KV.put(context[:pid], :hello, :world) == :ok
    assert KV.get(context[:pid], :hello) == :world
  end
end
```

- Il est aussi possible de consolider les préconditions dans un module pour mieux les réutiliser sur plusieurs cas de tests : voir [ExUnit.CaseTemplate — ExUnit v1.14.2 (hexdocs.pm)]