Bee

Bee = Ant
+ YAML - XML
+ Ruby - Java

La syntaxe YAML permet :

- Un build file plus lisible et moins verbeux.
- Des types scalaires variés (chaînes, entiers, flottants, dates, booléens).
- Des structures de données (listes et dictionnaires).

Exemple de build file

```
- build: hello

- properties:
    who: World

- target: hello
  script:
  - print: "Hello #{who}!"
```

Exemples de scalaires

entier : 1

flottant: 3.14

chaine : bee

date : 2010-07-06T16:30:03

booléen : true

symbole : :monsymbole

indéfini : null

Exemples de structures

Liste

- un
- deux
- trois

Compacte

[un, deux, trois]

Dictionnaire

1: un
2: deux
3: trois

Compact

{1: un, 2: deux, 3: trois}

Structure d'un fichier de build

- En-tête d'information : indique le nom du build, la cible par défaut, le(s) build(s) parent(s), etc...

- Propriétés : les constantes à modifier pour paramétrer le build.

- Les cibles (ou targets) : les fonctions que le build est capable de réaliser.

Les propriétés de build

- Les propriétés permettent de paramétrer le build.
- Elles peuvent être de tout type scalaire ou composite.
- Les chaînes sont évaluées comme une chaîne Ruby (et peuvent faire référence à une autre propriété du build avec la notation #{propriété}).

Exemples de propriétés

Propriétés

```
ent: 1
flot: 3.14
liste: [1, 2, 3]
dico: { foo: bar }
chaine: "s = #{ent + flot}"
chaine2: "#{liste.join(', ')}"
```

Valeurs

```
1
3.14
[1, 2, 3]
{ foo: bar }
"s = 4.14"
"1, 2, 3"
```

Les targets du build

- Ce sont les objectifs que doit atteindre le build.
- On peut les comparer à des fonctions.
- Elles sont constituées de tâches.
- La target par défaut est la première du build ou bien celle déclarée dans l'en-tête du build file, par l'entrée 'default'.
- Une target peut dépendre d'une autre qui sera alors exécutée auparavant.

Exemple de targets

```
- target: world
  depends: hello
  script:
  - print: "World"

- target: hello
  script:
  - print: "Hello"
```

Les tâches

- Les tâches sont les instructions pour effectuer le build.
- Les tâches peuvent être de trois types :
- Les tâches Bee sont indépendantes de la plateforme.
- Les tâches Shell sont des instructions exécutées par l'interpréteur de commandes de la plateforme.
- Les tâches Ruby sont des fragments de code Ruby exécutés dans un contexte où les propriétés du build sont des variables.

Les tâches Bee

```
- print: "Hello World!"
- mkdir: :build
```

- Les tâches Bee sont portables : elles ne dépendent pas du système.
- Elles peuvent faire référence à des propriétés du build avec la notation Ruby ou des symboles.
- On peut développer facilement (en Ruby) ses propres tâches.

Les tâches Shell

```
- "echo 'Hello World!'"
- "mkdir #{build}"
```

- Les tâches Shell ne sont pas portables et dépendent du système.
- Elles peuvent faire référence à des propriétés du build avec la notation Ruby.
- Ce n'est pas sale !

Les tâches Ruby

```
- rb: "puts 'Hello World!'"
- rb: "Dir.mkdir build"
```

- Les tâches Ruby sont portables (si elles ne contiennent pas de code spécifique à une plateforme).

- Les tâches Ruby sont exécutées dans un contexte (ou Binding) où les propriétés du build sont des variables locales.

Héritage

On a la possibilité de factoriser ses builds files en utilisant l'héritage. Un build file qui hérite d'un autre peut utiliser ou surcharger ses propriétés et ses targets (mais aussi invoquer la target parente avec l'instruction super). L'héritage multiple est possible, mais il ne doit pas y avoir collision entre targets ou propriétés des parents.

Il est possible de partager ses build files par le système de fichiers, par HTTP ou par des packages (des gems à

Exemple d'héritage

Supposons que l'on souhaite factoriser une target clean qui est utilisée dans nombre de nos build files.

```
- properties:
    clean_dirs:  [build]
    clean_files: ["**/*~", "**/.#*"]

- target: clean
  script:
  - rmdir: :clean_dirs
  - rm:    :clean_files
```

Un build file peut hériter de ce parent de la manière suivante, grâce à l'entrée extends :

```
- build: test
  extends: clean.yml

- properties:
    clean_dirs: [target]

- target: clean
  script:
  - super:
  - rm: "**/.DS_Store"
```

Les templates

Les templates sont des squelettes customisables permettant de démarrer rapidement un nouveau projet. C'est une manière de réutiliser la structure de projets existants.

Il est très simple de créer des templates pour ses propres projets. Il existe même un templates pour créer un nouveau projet de template !

Documentation

- Les tâches et les templates sont documentés par l'outil lui-même : taper "bee -k cp" par exemple pour obtenir de l'aide sur la tâche cp. Taper "bee -e sinatra" pour obtenir de l'aide sur ce template de projet.

- La documentation est toujours à jour car extraite de la documentation du code lui-même.

Packages supplémentaires

- Un package est un ensemble de tâches et de templates spécifiques à un domaine particulier. Il existe ainsi des packages pour Java ou Python par exemple.

- Développer de nouvelles tâches ou templates est très simple et rapide. Il existe même un template pour ça : taper "bee -t package".

Ce projet est aussi le vôtre !

Merci

Images courtesy of PD