Introduction à Go – TD4: JaaS

Franklin ”Snaipe” Mathieu

2023

`jq` est un outil en ligne de commande permettant de manipuler des documents JSON via un programme fonctionnel en jqlang (appelé un ”filtre”).

Par exemple:

```
$ sudo apt install -y jq # Pensez à installer jq!

$ curl -sS https://jsonplaceholder.typicode.com/todos/1
{
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
}

$ # Récupérer le titre du TODO qui a l'ID 1
$ curl -sS https://jsonplaceholder.typicode.com/todos/1 | jq '.title'
"delectus aut autem"

$ # Récupérer les 2 premiers TODOs
$ curl -sS https://jsonplaceholder.typicode.com/todos | jq '.[0], .[1]'
{
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
}
{
  "userId": 1,
  "id": 2,
  "title": "quis ut nam facilis et officia qui",
  "completed": false
}

$ # Récupérer le dernier TODOs non-complétés de l'utilisateur 1
$ curl https://jsonplaceholder.typicode.com/todos \
>   | jq '[.[] | select(.userId == 1 and (.completed | not))] | sort_by(.id) | last'
{
  "userId": 1,
  "id": 18,
  "title": "dolorum est consequatur ea mollitia in culpa",
  "completed": false
}
```

`jq` est extrêmement pratique, mais il a un gros défaut: il n’est utilisable qu’en ligne de commande, et si la révolution web 2.0 et l’émergence du cloud nous a bien appris quelque chose, c’est que si un outil n’a pas encore été implémenté en tant que service, alors quelqu’un le fera inévitablement.

Félicitations, vous êtes cette personne.

Le but de ce TD est d'implémenter pas à pas un service qui exposera les fonctionnalités de `jq` via une interface REST. Essentiellement, "JaaS" (Jq-as-a-Service).

L'interface proposée de ce service serait la suivante:

```
$ # Lancer le serveur jaas
$ jaas :1234

$ # Équivalent à curl -sS https://jsonplaceholder.typicode.com/todos/1 | jq '.title'
$ cat <<EOF | curl -F filter='.title' -F data=@- localhost:1234
> {
>   "userId": 1,
>   "id": 1,
>   "title": "delectus aut autem",
>   "completed": false
> }
> EOF
"delectus aut autem"

$ # Pareil que la commande précédente, mais avec jq -r (--raw-output)
$ cat <<EOF | curl -F filter='.title' -F data=@- -F options=raw-output localhost:1234
> {
>   "userId": 1,
>   "id": 1,
>   "title": "delectus aut autem",
>   "completed": false
> }
> EOF
delectus aut autem
```

Autrement dit, faire une requête POST de localhost:1234 après avoir lancé `jaas :1234` permet d'invoquer jq via cet endpoint et de récupérer le résultat. Les paramètres de cet endpoint sont passés via du form data, en utilisant -F ou -d avec curl.

1 Premiers pas

Comme toujours, commençons par créer un dossier de projet:

```
$ go mod init example.com/jaas
$ go get github.com/labstack/echo/v4
$ go get github.com/alecthomas/kong
```

Créez un fichier main.go:

```
package main

import (
        "fmt"
        "net/http"

        "github.com/alecthomas/kong"
        "github.com/labstack/echo/v4"
)

const usage = `
     _              ____
    | | __ _  __ _/ ___|
 _  | |/ _\ |/ _\ \___ \
| |_| | (_| | (_| |___) |
 \___/ \__,_|\__,_|____/

jq as a service. Basic usage:

$ echo '{"a":"Hello","b":"World"}' | curl -F filter='.a+", "+.b+"!"' -F data=@- %s
"Hello, World!"
`

func main() {
        var cli struct {
                UsageAddress string `default:"localhost:1234"`

                BindAddress string `arg optional default:":1234"`
        }
        kong.Parse(&cli)

        e := echo.New()

        svc := Service{}
        e.GET("/", func(c echo.Context) error {
                return c.String(http.StatusOK, fmt.Sprintf(usage, cli.UsageAddress))
        })
        e.POST("/", svc.Post)

        if err := e.Start(cli.BindAddress); err != nil {
                fmt.Println(err)
        }
}
```

Puis un fichier service.go:

```go
package main

import (
        "fmt"

        "github.com/labstack/echo/v4"
)

type Service struct{}

func (svc *Service) Post(c echo.Context) error {
        var params struct {
                Filter  string `form:"filter"`
                Data    string `form:"data"`
                Options string `form:"options"`
                Args    string `form:"args"`
        }
        c.Bind(&params)
        return fmt.Errorf("unimplemented")
}
```

1.1 Exécuter un programme

Le paquet `os/exec` donne une interface simplifier pour exécuter et manipuler des programmes:

```go
import (
        "os"
        "os/exec"
)

...

cmd := exec.Command("ls", "-al")

// Lance la commande et attends qu'elle finisse
if err := cmd.Run(); err != nil {
        return err
}
```

La méthode `Run` est une version plus courte d'un appel de `Start` puis `Wait`:

```go
// Lance la commande
if err := cmd.Start(); err != nil {
        return err
}

// Bloque jusqu'à ce qu'elle finisse
if err := cmd.Wait(); err != nil {
        return err
}
```

Si vous exécutez ce snippet de code sur `https://play.golang.com/p/yghqK6jgH9C`, vous vous apercevrez bien vite que rien n'a été affiché. C'est tout à fait normal, puisque les sorties standard n'ont pas été connectées, ce qui est équivalent à avoir redirigé le tout vers /dev/null. Corrigeons ce problème:

```
cmd := exec.Command("ls", "-al")
cmd.Stdout = os.Stdout
cmd.Stderr = os.Stderr

if err := cmd.Run(); err != nil {
        return err
}
```

Cette fois-ci, en exécutant le nouveau snippet sur `https://play.golang.com/p/CfCdKM1h5k3`, vous pouvez maintenant voir le résultat de `ls -al`.

`cmd.Stdout` et `cmd.Stderr` attendent des `io.Writer`, ce qui veut dire que vous pouvez attacher ce que vous voulez dessus. Par exemple, pour récupérer la sortie d'erreur dans une string pour ensuite en faire une erreur:

```
var stderr strings.Builder

cmd := exec.Command("ls", "-al", "/doesnotexist")
cmd.Stdout = os.Stdout
cmd.Stderr = &stderr

if err := cmd.Run(); err != nil {
        if _, ok := err.(*exec.ExitError); ok {
                // Run retourne une erreur de type *exec.ExitError si
                // le programme a terminé avec un statut autre que 0.
                // Malheureusement l'erreur dans ce cas est simplement
                // "exit status 1" sans plus de contexte, donc on
                // remplace l'erreur avec la sortie standard collectée.
                return errors.New(stderr.String())
        }
        return err
}
```

`cmd.Stdin` attends similairement un `io.Reader` si vous devez passer des données sur l'entrée standard du programme.

Le type `*exec.Cmd` contient d'autres attributs contrôlant l'initialisation du programme, tel que `Env` pour assigner des variables d'environnement; pensez à en regarder la documentation.

2 Retroussons-nous les manches

Il est temps de commencer à implémenter votre service. Commençons par une abstraction de jq: nous allons faire un nouveau paquet en dessous de votre projet.

```
$ # Simplement créer le dossier du paquet
$ mkdir jq
```

Tout sous-dossier est automatiquement considéré comme paquet; pas besoin de lancer `go mod init`.

Vu que le paquet parent est le module principal, et est nommé `example.com/jaas`, le paquet jq sera importable en tant que `example.com/jaas/jq`.

Faisons un type pour représenter un filtre jq; créez un fichier `jq/filter.go`

```go
package jq

import (
        "fmt"
        "io"
)

// Filter is a jqlang program (called a "filter") that is used to process
// JSON data.
type Filter struct {
        filter string
}

func NewFilter(filter string) *Filter {
        return &Filter{filter: filter}
}

// Run executes the filter on the contents of in, and writes the result
// to out.
func (f *Filter) Run(in io.Reader, out io.Reader) error {
        // FIXME: implement this function
        return fmt.Errorf("unimplemented")
}

func (f *Filter) String() string {
        return f.filter
}
```

Vous n'êtes pas obligés de suivre exactement cette API; si vous voulez en changer la structure, n'hésitez pas à le faire. Plus particulièrement, il est possible que vous serez amené à vouloir modifier cette API plus tard lors de l'implémentation du service.

Écrivons aussi des tests pour valider le comportement du filtre – dans un fichier `jq/filter_test.go`:

```go
package jq

import (
	"bytes"
	"encoding/json"
	"fmt"
	"testing"
)

func TestFilter(t *testing.T) {
	cases := []struct {
		Filter string
		Input  any
		Output string
	}{
		{
			Filter: ".title",
			Input: map[string]any{
				"userId":    1,
				"id":        1,
				"title":     "delectus aut autem",
				"completed": false,
			},
			Output: "\"delectus aut autem\"\n",
		},
	}

	for i, tcase := range cases {
		t.Run(fmt.Sprintf("%d", i), func(t *testing.T) {
			var in, out bytes.Buffer
			if err := json.NewEncoder(&in).Encode(tcase.Input); err != nil {
				t.Fatal(err)
			}
			if err := NewFilter(tcase.Filter).Run(&in, &out); err != nil {
				t.Fatal(err)
			}
			if actual := out.String(); tcase.Output != actual {
				t.Fatalf("expected %s, got %s", tcase.Output, actual)
			}
		})
	}
}
```

Pensez à ajouter d'autre cas dans la slices `cases`!

La fonction TestFilter teste l'utilisation basique de l'API de filtre, mais ne fais rien au niveau des erreurs. Il serait judicieux, par exemple, de pouvoir distinguer les erreurs de syntaxe sur le filtre des erreurs dans les documents JSON traités.

`jq` fait déjà la distinction au niveau du code de sortie du programme:

```
$ jq -n 'bad'
jq: error: bad/0 is not defined at <top-level>, line 1:
bad
jq: 1 compile error

$ echo $?
3

$ echo bad | jq .
jq: parse error: Invalid numeric literal at line 2, column 0

$ echo $?
5
```

Il nous suffit donc de définir quelques types d'erreur:

```go
package jq

// SyntaxError represents errors when the jq filter is malformed.
type SyntaxError struct {
        Filter  string
        Message string
}

func (e *SyntaxError) Error() string {
        return fmt.Sprintf("syntax error on filter %q: %v", e.Filter, e.Message)
}

// ParseError represents errors when parsing JSON data.
type ParseError struct {
        Message string
}

func (e *ParseError) Error() string {
        return fmt.Sprintf("json parse error: %v", e.Message)
}
```

Essayez d'écrire un test pour valider que Filter.Run() renvoie l'erreur appropriée. Pour vérifier si une erreur est d'un type particulier, vous pouvez utiliser `errors.As`:

```go
var synerr *SyntaxError
if errors.As(err, &synerr) {
    // err est (ou contient) une *SyntaxError,
    // et vous pouvez y accéder via synerr.
}
```

Une fois fini et fonctionnel, passez à l'implémentation de Service.Post(), où vous utiliserez votre nouveau paquet jq et ce que vous avez vu jusque là.

Bonus: `jq` autorise la spécification d'arguments via `--arg` et `--argjson`. Modifiez votre service et le paquet jq pour que l'utilisateur puisse passer des arguments au filtre lors de son exécution. Si possible, essayez de rendre le changement backward-compatible (un bon indicateur est que les tests ne devraient pas être modifiés).