# Introduction à Go – TD3: système de fichier distant

Franklin ”Snaipe” Mathieu

2023

Le but de ce TD est d'implémenter de manière incrémentale un service HTTP exposant un système de fichier local.

L'interface de ce service serait:

```
$ # expose /home sous localhost:1234
$ httpfs :1234 /home

$ # Récupère le fichier ~/greeting via http
$ echo Bonjour > ~/greeting.txt
$ curl localhost:1234/$USER/greeting
Bonjour

$ # Téléverse le fichier ~/foo via http
$ curl -X PUT --data 'bar' localhost:1234/$USER/foo
$ cat ~/foo.txt
Bonjour

$ # Récupère une tarball de votre dossier home et en affiche le contenu
$ curl localhost:1234/$USER/ | tar -t
```

Autrement dit:

- Faire une requête GET de localhost:1234/bar après avoir lancé `httpfs :1234 /foo` renvoie le contenu de /foo/bar si bar est un fichier régulier
- Faire une requête PUT de localhost:1234/baz après avoir lancé `httpfs :1234 /foo` crée (ou remplace) le fichier /foo/baz avec le contenu du corps de la requête PUT.
- Faire une requête GET de localhost:1234/qux après avoir lancé `httpfs :1234 /foo` renvoie une tarball avec le contenu de /foo/qux si qux est un dossier

## 1 Premiers pas

Commençons par créer un dossier de projet:

```
$ go mod init example.com/httpfs
$ go get github.com/labstack/echo/v4
```

Créez un fichier main.go:

```go
package main

import (
	"fmt"
	"net/http"

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

func main() {

	var cli struct {
		BindAddress string `arg`
		RootDir     string `arg`
	}

	kong.Parse(&cli)

	e := echo.New()

	svc := Service{
		RootDir: cli.RootDir,
	}
	e.GET("/*", svc.GetPath)
	e.PUT("/*", svc.PutPath)

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

Puis un fichier service.go:

```go
package main

import (
	"log/slog"

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

type Service struct {
	RootDir string
}

func (svc *Service) GetPath(c echo.Context) error {
	slog.Info("GetPath", "path", c.Request().URL.Path)
	return fmt.Errorf("unimplemented")
}

func (svc *Service) PutPath(c echo.Context) error {
	slog.Info("PutPath", "path", c.Request().URL.Path)
	return fmt.Errorf("unimplemented")
}
```

`e.GET("/*", svc.GetPath)` assigne à tous les chemins du service la méthode svc.GetPath. Cette première implémentation ne fait rien de particulier mais donne la structure générale.

## 1.1 Lire un fichier

Le paquet os regroupe tous les fonctions et types utiles pour interagir de manière portable avec le système.

Pour ouvrir (en lecture) un fichier existant:

```go
package main

import (
        "fmt"
        "io"
        "os"
)

func ReadFile(path string) ([]byte, error) {
        f, err := os.Open(path) // ouvre le fichier et retourne un *os.File
        if err != nil {
                return "", err
        }
        defer f.Close() // f.Close() est executé lorsque la fonction retourne

        // Lis tout le contenu du fichier et le retourne dans data
        return io.ReadAll(f)
}
```

`os.Open` retourne un `*os.File`, qui implémente la plupart des méthodes pour manipuler ce fichier; par exemple `f.Stat()` peut être utilisé pour avoir plus de détails sur le fichier (par exemple, si un fichier est un fichier régulier, ou un répertoire).

## 1.2 Créer des fichiers

Pour créer (ou remplacer) des fichiers, on utilise os.Create:

```go
package main

import (
        "fmt"
        "io"
        "os"
)

func WriteFile(path string, data []byte) error {
        f, err := os.Create(path) // créée (ou tronque) le fichier
        if err != nil {
                return "", err
        }
        defer f.Close() // f.Close() est executé lorsque la fonction retourne

        // écris data dans f
        if _, err := f.Write(data); err != nil {
                return "", err
        }

        // pensez à vérifier les erreurs de Close lors de l'écriture!
        // le defer f.Close() se lance quand même, mais ne fait rien (f.Close() est idempotent)
        return f.Close()
}
```

## 1.3 Créer une tarballs

Go propose dans sa bibliothèque standard le paquet `archive/tar` pour lire et écrire des fichiers tar.

Par exemple, pour créer dynamiquement une tarball avec un fichier greeting contenant ”Bonjour”:

```go
package main

import (
	"archive/tar"
	"io"
)

func WriteGreetingTarball(dest io.Writer) error {
	w := tar.NewWriter(dest)
	defer w.Close()

	const greeting = "Bonjour"

	// Crée un fichier nommé greeting dans la tarball
	err := w.WriteHeader(&tar.Header{
		Name: "greeting",
		Size: len(greeting),
	})
	if err != nil {
		return err
	}

	// Écris le contenu du fichier greeting
	if _, err := io.WriteString(w, greeting); err != nil {
		return err
	}

	return w.Close()
}
```

`io.Writer` est une interface. Si on regarde la documentation avec `io.Writer`, vous verrez cette définition:

```go
type Writer interface {
	Write(p []byte) (n int, err error)
}
```

Tout type qui implémente cette méthode implémente automatiquement io.Writer; par conséquent, WriteGreetingTarball fonctionne avec tout type de Writer. Vous avez déjà intéragi avec des types implémentant io.Writer; *bytes.Buffer, *strings.Builder, *os.File. Par exemple, pour écrire la tarball dans un fichier:

```go
func WriteGreetingTarballToFile(path string) error {
	f, err := os.Create(path) // créée (ou tronque) le fichier
	if err != nil {
		return "", err
	}
	defer f.Close()

	if err := WriteGreetingTarball(f); err != nil {
		return "", err
	}

	return f.Close()
}
```

Un cas particulier est le type `interface{}`, vu qu'il s'agit de l'interface sans méthode, tous les types de Go l'implémente. Il est donc possible de l'utiliser pour accepter des valeurs de n'importe quel type, et pour cette raison, Go définit any comme un alias de `interface{}`:

```go
func fatalf(code int, val any) {
	fmt.Println(val)
	os.Exit(1)
}
```

## 2 Dans le vif du sujet

Nous avons normalement toutes les bases pour attaquer le sujet initial du TD.

Commencez par implémenter le téléchargement et téléversement de fichiers régulier; pour rappel l'utilisation attendue est:

```
$ # expose /home sous localhost:1234
$ httpfs :1234 /home

$ # Récupère le fichier ~/greeting via http
$ echo Bonjour > ~/greeting.txt
$ curl localhost:1234/$USER/greeting
Bonjour

$ # Téléverse le fichier ~/foo via http
$ curl -X PUT --data 'bar' localhost:1234/$USER/foo
$ cat ~/foo.txt
Bonjour
```

Cette partie devrait exercer les concepts vus avec les fonctions ReadFile et WriteFile écrites précédemment.

Une fois que votre solution fonctionne, passez à l'implémentation de la fonction tarball, qui consiste à retourner une tarball quand le chemin demandé est un dossier (note: regardez la fonction filepath.WalkDir).