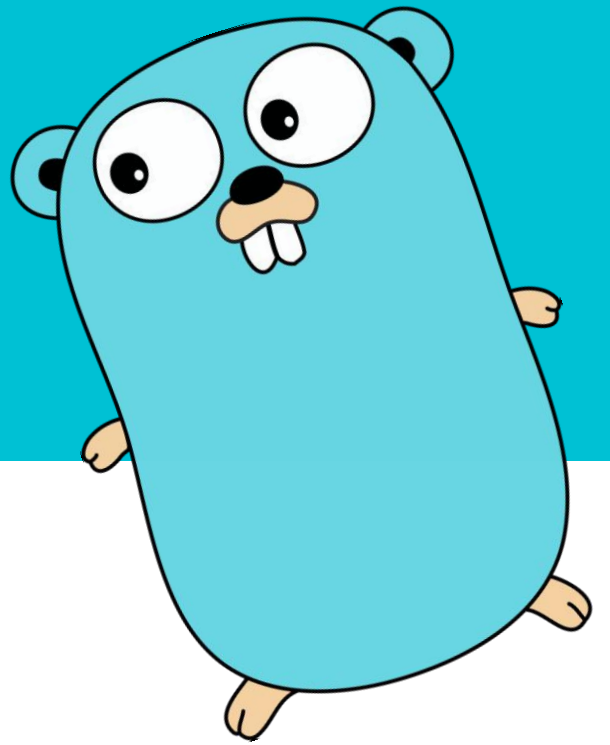
Go lang - TD

Web MVC



Laurent Guérin
Version 1.2 - Mars 2020

Le package « http »

Package pour développer des **clients http**
et des **serveurs http**

Doc : https://golang.org/pkg/net/http/

```
import (
   "net/http"
)
```

Exemples d'utilisation « côté client » :

```
resp, err := http.Get("http://example.com/")
...
resp, err := http.Post("http://example.com/upload", "image/jpeg", &buf)
...
resp, err := http.PostForm("http://example.com/form",
        url.Values{"key": {"Value"}, "id": {"123"}})
```

Premier serveur « http » → text

Step 1

- Création d'une fonction « **helloHandler** » qui renvoie le texte « Hello »
- Association de cette fonction à l'URL « **/hello** »
- Lancement du serveur http

"/hello"

```
func helloHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello !")
}
```

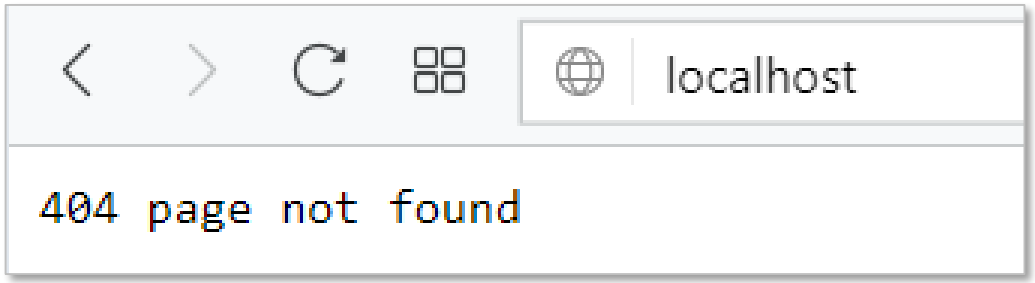
Pas d'indication sur le type « mime »

(string , func)

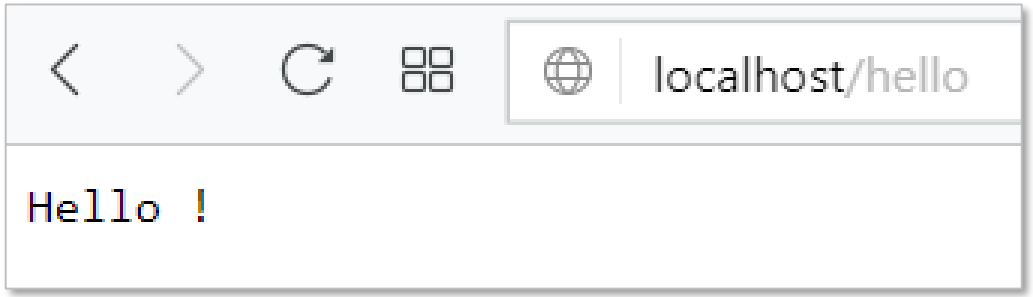
```
http.HandleFunc("/hello", helloHandler)

err := http.ListenAndServe(":80", nil)

log.Fatal(err)
```

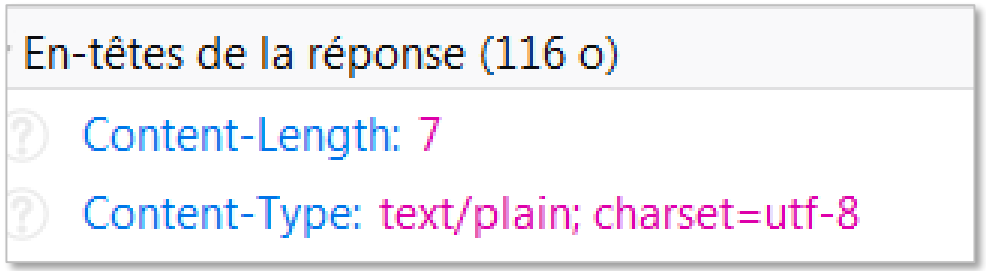
Premier serveur « http » → text

Tests et résultat :



"/hello"





Content-Type : text/plain

Serveur « http » → html

- Modification de « helloHandler » pour renvoyer du HTML

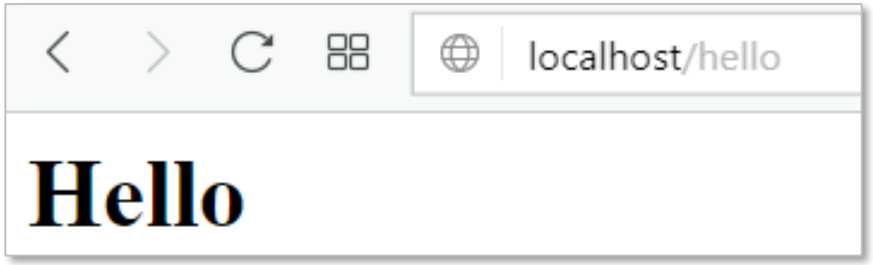
Step 2

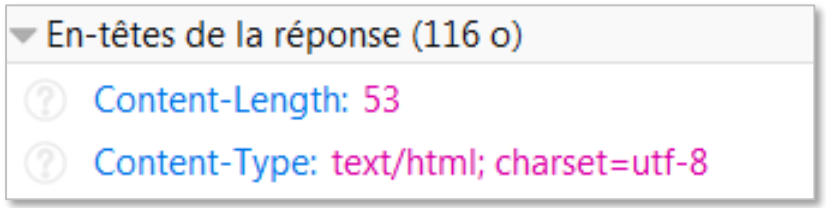
```
func helloHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "<html>")
    fmt.Fprintf(w, "<head>")
    fmt.Fprintf(w, "</head>")
    fmt.Fprintf(w, "<body>")
    fmt.Fprintf(w, "<h1>Hello</h1>")
    fmt.Fprintf(w, "</body>")
    fmt.Fprintf(w, "</html>")
}
```

Pas d'indication
sur le type « mime »

Serveur « http » → html

- Tests et résultat :





Content-Type : text/html

Type « Request »

- Chaque requête est décrite par une structure

https://golang.org/pkg/net/http/

```
type Request struct {
        // Method specifies the HTTP method (GET, POST, PUT, etc.).
        // For client requests an empty string means GET.
        Method string

        // URL specifies either the URI being requested (for server
        // requests) or the URL to access (for client requests).
        //
        // For server requests the URL is parsed from the URI
        // supplied on the Request-Line as stored in RequestURI.  For
        // most requests, fields other than Path and RawQuery will be
        // empty. (See RFC 2616, Section 5.1.2)
        //
        // For client requests, the URL's Host specifies the server to
        // connect to, while the Request's Host field optionally
        // specifies the Host header value to send in the HTTP
        // request.
        URL *url.URL

        // The protocol version for incoming server requests.
        //
        // For client requests these fields are ignored. The HTTP
        // client code always uses either HTTP/1.1 or HTTP/2.
        // See the docs on Transport for details.
        Proto      string // "HTTP/1.0"
        ProtoMajor int    // 1
        ProtoMinor int    // 0

        // Header contains the request header fields either received
```

Struct : fields + functions

```
type Request
    func NewRequest(method, url string, body io.Reader) (*Request, error)
    func ReadRequest(b *bufio.Reader) (*Request, error)
    func (r *Request) AddCookie(c *Cookie)
    func (r *Request) BasicAuth() (username, password string, ok bool)
    func (r *Request) Context() context.Context
    func (r *Request) Cookie(name string) (*Cookie, error)
    func (r *Request) Cookies() []*Cookie
    func (r *Request) FormFile(key string) (multipart.File, *multipart.FileHeader, error)
    func (r *Request) FormValue(key string) string
    func (r *Request) MultipartReader() (*multipart.Reader, error)
    func (r *Request) ParseForm() error
    func (r *Request) ParseMultipartForm(maxMemory int64) error
    func (r *Request) PostFormValue(key string) string
    func (r *Request) ProtoAtLeast(major, minor int) bool
    func (r *Request) Referer() string
    func (r *Request) SetBasicAuth(username, password string)
    func (r *Request) UserAgent() string
    func (r *Request) WithContext(ctx context.Context) *Request
    func (r *Request) Write(w io.Writer) error
    func (r *Request) WriteProxy(w io.Writer) error
```

Type « URL »

- A partir de la structure « Request » il possible de récupérer la structure « URL » décrivant l'URL de la requête

```
func urlHandler(w http.ResponseWriter, r *http.Request) {
    url := r.URL
}
```

Struct : fields + functions

```
type URL struct {
        Scheme     string
        Opaque     string    // encoded opaque data
        User       *Userinfo // username and password information
        Host       string    // host or host:port
        Path       string    // path (relative paths may omit leadi
        RawPath    string    // encoded path hint (see EscapedPath
        ForceQuery bool      // append a query ('?') even if RawQue
        RawQuery   string    // encoded query values, without '?'
        Fragment   string    // fragment for references, without '#
}
```

```
type URL
    func Parse(rawurl string) (*URL, error)
    func ParseRequestURI(rawurl string) (*URL, error)
    func (u *URL) EscapedPath() string
    func (u *URL) Hostname() string
    func (u *URL) IsAbs() bool
    func (u *URL) MarshalBinary() (text []byte, err error)
    func (u *URL) Parse(ref string) (*URL, error)
    func (u *URL) Port() string
    func (u *URL) Query() Values
    func (u *URL) RequestURI() string
    func (u *URL) ResolveReference(ref *URL) *URL
    func (u *URL) String() string
    func (u *URL) UnmarshalBinary(text []byte) error
```

Informations fournies par l'URL

Step 3

- Faire un « handler » qui décrit l'URL de la requête
  afficher les principales informations portées par la structure URL
  - Path
  - Host
  - URI
  - Query

- Une réponse de type« text/plain » est suffisante

Ressources statiques

- La fonction « **ServeFile** » de « http » permet de renvoyer le contenu de fichiers présents sur le filesystem, notamment des fichiers « .html »

```
http://localhost/foo.html
```

```
Path du fichier sur le serveur
Exemple : www/foo.html
```

```
func ServeFile(w ResponseWriter, r *Request, name string)
```

ServeFile replies to the request with the contents of the named file or directory.

If the provided file or directory name is a relative path, it is interpreted relative to the current directory and may ascend to parent directories. If the provided name is constructed from user input, it should be sanitized before calling ServeFile. As a precaution, ServeFile will reject requests where r.URL.Path contains a ".." path element.

As a special case, ServeFile redirects any request where r.URL.Path ends in "/index.html" to the same path, without the final "index.html". To avoid such redirects either modify the path or use ServeContent.

Serveur de ressources statiques

Step 4

- Faire un serveur http qui renvoie des ressources statiques :
  fichiers « .html », « .css », « .js », etc

```
http://localhost/foo.html
```

- Créer un **handler** associé à « **/** » qui utilise « **ServeFile** »
- Récupérer le « **file path** » à partir de « **r.URL** »
- Stocker les fichiers dans le répertoire « **www** » du projet (à utiliser pour construire le chemin complet du fichier)

Si “www” est un path relatif (pas de chemin absolu) la localisation est celle de l’exécutable

→

/x/y/z/**bin**/**application.exe**
/x/y/z/**bin**/**www/fichier.html**

Paramètres de la requête

- Pour les valeurs passés par l' **URL**

```
http://myhost/myaction?a=1&b=2
```

→ **http « GET » → handler**

utilisation de **r.URL.Query()**

- Pour les valeurs postées dans un formulaire (+ les valeurs de l'URL)

Your name : name

Your prefered color : red

→ **http « POST » → handler**

utilisation de **r.ParseForm()** puis **r.Form.Get()**

Exemple :

```
r.ParseForm()
name := r.Form.Get("name")
// returns "" (void string) if "xx" not found
```

URL / Query String → « Values »

- Exemple :

```
http://localhost/action?firstName=aaa&lastName=bbb
```

```
func myHandler(w http.ResponseWriter, r *http.Request) {

 queryValues := r.URL.Query() // Parse query & return Values

 firstName := queryValues.Get("firstName")
 lastName  := queryValues.Get("lastName")
}
```

```
type Values map[string][]string
```

NB : ne renvoie que les paramètres de l'URL (http GET)
pas ceux d'un formulaire soumis avec une méthode POST

URL / Query String → « Values »

- Le cas des valeurs multiples :

```
http://localhost/form1?a=1&a=2
```

- En cas de valeurs multiples seule la première valeur est récupérée

```
a := queryValues.Get("a")    // "1" uniquement
```

- Pour récupérer les valeurs multiples

```
allValues := queryValues["a"] // "1" et "2" (tableau)
```

URL / Query String – valeur d'un paramètre

Step 5

- Faire une fonction « **getParameter** »

```
func getParameter(r *http.Request, name string) string {
    // TODO
}
keyValue := getParameter(r, "a")
```

qui renvoie la valeur du paramètre dont le nom est « name »
ou une chaine vide si le paramètre n'existe pas

- Cas de test :

`http://localhost/param`	→	`""`
`http://localhost/param?a=1`	→	`"1"`
`http://localhost/param?a=1&a=2`	→	`"1"`

Paramètres postés + Query String

Step 6

- Faire 2 handlers

```
http.HandleFunc("/", welcomeHandler)

http.HandleFunc("/param", paramHandler)
```

- Dans le handler qui traite **«/param** » récupérer les paramètres postés avec les méthodes **r.ParseForm()** et **r.Form.Get("..")**

`http://localhost`

First name : Bart

Last name : Simpsons

Submit

`http://localhost/param`

'firstname' value is 'Bart'

'lastname' value is 'Simpsons'

Types « Response » et « ResponseWriter »


```
type Response struct {
        Status     string // e.g. "200 OK"
        StatusCode int    // e.g. 200
        Proto      string // e.g. "HTTP/1.0"
        ProtoMajor int    // e.g. 1
        ProtoMinor int    // e.g. 0

        // Header maps header keys to values
        // headers with the same key, they m
        // delimiters.  (Section 4.2 of RFC
        // be semantically equivalent to a c
        // Header values are duplicated by o
        // ContentLength, TransferEncoding,
        // authoritative.
        //
        // Keys in the map are canonicalized (see Car
        Header Header

        // Body represents the response body.
        //
        // The http Client and Transport guarantee th
```

https://golang.org/pkg/net/http/

```
type Response
    func Get(url string) (resp *Response, err error)
    func Head(url string) (resp *Response, err error)
    func Post(url string, contentType string, body io.Reader) (resp *Response, err error)
    func PostForm(url string, data url.Values) (resp *Response, err error)
    func ReadResponse(r *bufio.Reader, req *Request) (*Response, error)
    func (r *Response) Cookies() []*Cookie
    func (r *Response) Location() (*url.URL, error)
    func (r *Response) ProtoAtLeast(major, minor int) bool
    func (r *Response) Write(w io.Writer) error
```

```
type ResponseWriter interface {
        // Header returns the header map that will be sent by
        // WriteHeader. The Header map also is the mechanism with which
        //
        //
        //
```

Les « handlers » n'ont accès qu'à « **ResponseWriter** »

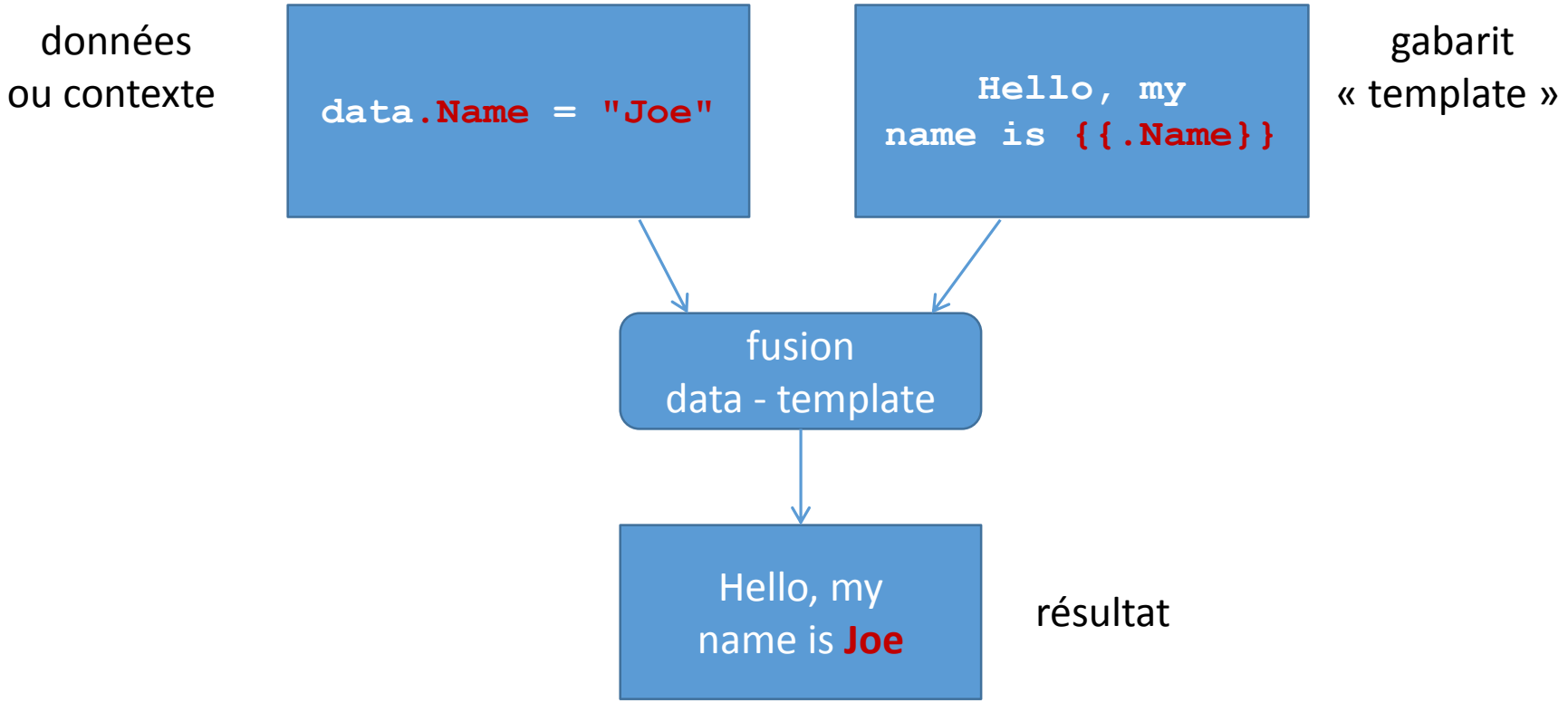
ResponseWriter interface (3 fonctions) :

```
Header() Header
Write([]byte) (int, error) // response data
WriteHeader(int) // http status code
```

Templates

Templates – Principe général

- Le principe de « templating » consiste à « fusionner » des données et un gabarit (généralement appelé « template ») pour produire un résultat final qui sera présenté à l'utilisateur



Principe identique à celui d'autres environnements :

- **JSP**
- **PHP**
- etc …

Les templates en Go - Syntaxe

- Documentation : https://golang.org/pkg/text/template/

```
{{.}}
Elément courant du « pipeline »
```

```
data := "Joe"
```

→

```
Hello {{.}}
```

```
{{.name}}
Valeur d'un champ d'une structure
ou d'une clé d'une map
```

```
type Student struct {
  FirstName string
  LastName string
} // exported fields
```

→

```
Hello {{.FirstName}}
{{.LastName}}
```

```
map[string]string{"a": "AA",  "b": "BB"}
```

→

```
Hello {{.a}} {{.b}}
```

Quelques cas d'erreurs :

- référence à un champ qui n'existe pas :
  executing "mytemplate" at <.Foo>: can't evaluate field Foo in type xxxx
- référence à un champ qui n'est pas exporté (champ privé) :
  executing "mytemplate" at <.secretCode>: secretCode is an unexported field of struct type xxxx

Les templates en Go - Syntaxe

- Notation pointée :

```
type Student struct {
  FirstName string
  LastName string
  Teacher Teacher
}
```

```
type Teacher struct {
  FirstName string
  LastName  string
}
```

```
{{.Teacher.FirstName}}

{{.Teacher.LastName}}
```

- Itérations avec « range »

```
type Person struct {
  Name     string
  Emails   []string
}
```

```
{{range .Emails}} email : {{.}}
{{end}}
Itération sur le tableau « Emails »
```

```
{{range $e := .Emails}} email {{$e}} {{end}}
```

Les templates en Go - Syntaxe

- Notion de « **pipeline** » :

« . » désigne l' **élément courant**,
NB : cet élément courant peut varier
par exemple dans le cas d'un **{{range}}** ou **{{with}}**

« $ » désigne la **racine de l'élément** qui a été passé au template

```
type Person struct {
  Name     string
  Emails   []string
}
```

```
{{range .Emails}} - email : {{.}} name = {{$.Name}}
{{end}}
« . » référence l'élément courant dans le tableau « Emails »
« $. » référence la racine --> « Person »
```

Les templates en Go - Syntaxe

- « if / else » :

Les templates sont « logic-less »
Ils ne disposent que de « if/else » pour gérer les conditions
Le « if » est suivi d'un nom de fonction : « not », « eq », « gt », etc

```
{{ if not .LoggedIn }}
<h1>Not logged in</h1>
{{ end }}
```

```
// si .Val > 1
{{ if or .Flag1 .Flag2 }}
  xxxx
{{ end }}
```

```
// si .Val == 1
{{ if eq .Val 1 }}
   xxxx
{{ else }}
   xxxx
{{ end }}
```

```
// si .Val > 1
{{ if gt .Val 1 }}
  xxxx
{{ end }}
```

Les templates en Go - Syntaxe

- Fonctions :

Comparaison :
eq ==
ne !=
lt <
le <=
gt >
ge >=

Booléens :
and
or
not

Print :
print
printf
println

Longueur de $x :
len $x

Itération avec la fonction « **index** $i » → 0 à N

```
{{range $i, $e := .Emails}} email {{index $i}} {{.}}
{{end}}
```

Les templates en Go

- Go propose 2 types de templates :
  - un pour produire du **texte**
  - un pour produire du **html**

- Le contenu d'un template peut provenir
  - D'une **chaine de caractères** en mémoire
  - D'un **fichier** stocké sur le filesystem

- L'utilisation d'un template se décompose de la façon suivante
  - Analyse (« parsing ») du contenu du template : « **Parse** » ou « **ParseFiles** »
  - Fusion : « **Execute** » ou « **ExecuteTemplate** »

Go – Templates de type « text »

- Principe de fonctionnement pour un template dont le **contenu** est défini **en mémoire** (dans une chaine de caractères)

```
import "text/template"
```

```
// define the data (model)
data := "Bart Simpson"

// create a new template with a name
tmpl := template.New("mytemplate")

// define the template content
content := "Hello {{.}} ! \n\n"

// parse the template content
tmpl, err1 := tmpl.Parse(content)

// merge the template 'tmpl' with the data
err2 := tmpl.Execute(os.Stdout, data)
```

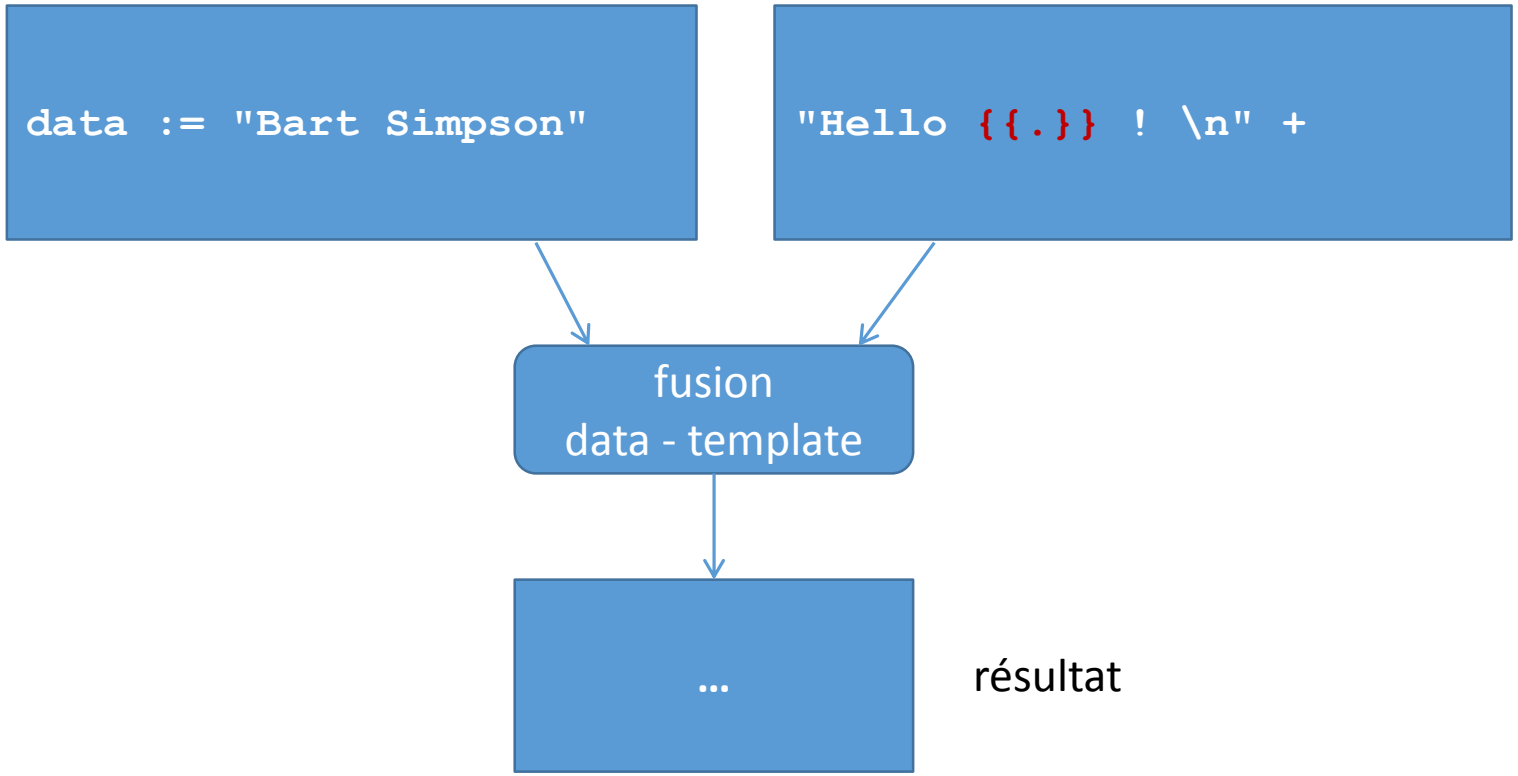
Le format `…` peut être plus pratique pour un template relativement long :

```
content := `Hello {{.Name}}
Your age is {{.Age}}
`
```

Go – Templates de type « text »

Step 10

- Exemple 1
  - Template : chaine de caractères
  - Data : chaine de caractères



Go – Templates de type « text »

Step 11

- Exemple 2
  - Template : chaine de caractères
  - Data : une structure « Student »

```
data := Student{"Bart",
"Simpson", 25, "abc" }
```

```
"Hello {{.FirstName}} {{.LastName}} ! \n" +
"I am {{.Age}} years old. \n"
```

```
type Student struct {
  FirstName   string
  LastName    string
  Age         int
  secretCode string
}
```

fusion
data - template

…

résultat

Go – Templates de type « text »

Step 12

- Pour des données contenues dans la map

```
{  "v1": "One",   "v2": "Two",   "v3": "Three",   "v4": "Four"  }
```

- Faire un template qui affiche les valeurs de « v1 » et « v2 »

- Faire un template qui affiche toutes les valeurs de la map avec **`{{range}}`**

Go – Templates de type « text »

- Principe de fonctionnement pour un template dont le **contenu** est défini **dans un fichier**

```
// define the data (model)
data := something

// parse a list of templates (1..N files)
tmpl, err := template.ParseFiles("path/file1.gotxt","path/file2.gotxt")
if err != nil … …

// merge one of the templates with the data
err = tmpl.ExecuteTemplate(os.Stdout, "file1.gotxt", data )
if err != nil … …
```

- Le nom de fichier peut avoir n'importe quelle extension, par exemple : « .tpl », « .html », « .thtml », « .gohtml »

Go – Templates de type « text »

Step 14

- Exemple
  - Templates : 2 fichiers « .gotxt »
  - Data : une structure « Student »

```
data := Student{"Bart",
"Simpson", 25, "abc" }
```

```
type Student struct {
  FirstName   string
  LastName    string
  Age         int
  secretCode string
}
```

fusion
data - template

…

résultat

test14.gotxt

```
Hello {{.FirstName}} !
```

test14bis.gotxt

```
Hello {{.FirstName}} {{.LastName}} !
I am {{.Age}} years old.
```

Fichiers templates
Si path relatif à placer au niveau de exécutables :
/x/y/z/bin/**application.exe**
/x/y/z/bin/**templates/xxx.gotxt**

Go – Templates de type « html »

- Et pour générer du **HTML** ?
- C'est exactement le même principe
- Il faut simplement importer « **html/template** » au lieu de « **text/template** »

```
import "html/template"
```

- L'interface est la même (mêmes fonctions) mais la sortie HTML est sécurisée afin d'éviter certaines attaques web

Go – Templates de type « html »

Step 15

- Exemple
  - Data : une structure « **Employee** »
  - Template : un fichier « **/template/employee.gohtml** » qui produit une page HTML qui affiche les informations de l'employé
    Pour « Manager » afficher « yes » pour « true » ou « no » pour « false »

```
type Employee struct {
  Id int
  FirstName string
  LastName string
  Rank int
  Manager bool
}
```

/template/employee.gohtml

```
<!DOCTYPE html>
<html>
<head>
.. .. .. ..
</head>
<body>
<h1>Hello employee {{.Id}} </h1>
.. .. .. ..
</body>
</html>
```

Fichiers templates
Si path relatif à placer au niveau de exécutables :
/x/y/z/bin/**application.exe**
/x/y/z/bin/**templates/xxx.gohtml**

Les templates en Go – Quelques liens

- Il est possible de créer des **fonctions spécifiques** utilisables ensuite dans les templates : http://goinbigdata.com/example-of-using-templates-in-golang/

Sessions

Go – Sessions

- Le package « http » de GO ne dispose pas de gestion des sessions
- Seuls les « cookies » sont supportés
- La gestion des informations propres à un utilisateur peuvent être gérées de différentes façon
  - Gérer toutes les informations avec des cookies
    → n'est acceptables que pour des cas limités avec peu d'information et peu de contraintes de sécurité
  - Développer son propre système de gestion de sessions
    → déconseillé
  - Utiliser des solutions prêtes à l'emploi
    → par exemple le web toolkit Gorilla (https://github.com/gorilla) qui apporte un système de gestion des sessions (https://github.com/gorilla/sessions)