Reverse proxy

Reverse proxy simple

GitHub repo

Le reverse proxy est quelque chose de majoritairement utilisé aujourd'hui.

Dans beaucoup de cas d'utilisation, on utilise des outils tels que Nginx, Apache, Caddy uniquement pour faire du reverse proxy.

Mais avec **Gin**, on peut coder ça soit même !

Contexte :

- Notre serveur **Gin** écoute en local sur le port 8080
- Mon serveur portainer tourne en local et écoute sur le port 9000
- Je veux qu'en me connectant sur mon serveur **Gin**, ce dernier me fasse un reverse proxy sur mon serveur portainer.

Dans mon fichier `main.go`, nous allons déclarer l'URL de mon reverse proxy ainsi qu'une méthode `proxy` qui sera la méthode utilisée par **Gin**

```
package main

import "github.com/gin-gonic/gin"

const reverseServerAddr = "http://127.0.0.1:9000"

func proxy(c *gin.Context) {

}

func main() {

}
```

Nous dire à notre routeur **Gin**, que TOUTES les requêtes, et ce, peu importe la méthode, doit utiliser notre fameuse méthode `func proxy(c *gin.Context)`.

```
func main() {
	router := gin.Default()

	router.Any("/*any", proxy)

	router.Run(":8080")
}
```

- `router.Any` signifie que quelle que soit la méthode (GET, POST, PUT, ...) utilisé, elle sera pris en charge.
- `/*any` est une expression indiquant à **Gin** que la route peut être n'importe quoi

---

Nous allons maintenant nous attaquer à la méthode `func proxy(c *gin.context)` qui va dans un premier temps parser notre URL de destination (la variable `reverseServerAddr`) avec la librairie `net/url` :

```
func proxy(c *gin.Context) {

	proxy, err := url.Parse(reverseServerAddr)
	if err != nil {
		fmt.Printf("Error parsing reverse proxy address: %s\n", err)
		c.IndentedJSON(http.StatusInternalServerError, gin.H{
			"message": "Error parsing reverse proxy address",
			"error":   err.Error(),
		})
		return
	}

}
```

On gère bien évidement le cas d'erreur où on n'arriverait pas à parser correctement cette URL et on gère le renvoie d'une erreur au client, on arrête également la méthode avec `return`.

La variable `proxy` sera du type `*url.URL`.

Il suffit ensuite d'extraire la requête de notre contexte `c *gin.Context` et modifier son chemin ainsi que son protocole par celui de notre `proxy`.

```
func proxy(c *gin.Context) {
	
	...

	req := c.Request
	req.URL.Scheme = proxy.Scheme
	req.URL.Host = proxy.Host
}
```

Notre requête est prête, nous allons maintenant l’exécuter et récupérer son retour

```
func proxy(c *gin.Context) {
	
	...

	transport := http.DefaultTransport
	resp, err := transport.RoundTrip(req)
	if err != nil {
		fmt.Printf("Error making request: %s\n", err)
		c.IndentedJSON(http.StatusInternalServerError, gin.H{
			"message": "Error making request",
			"error":   err.Error(),
		})
		return
	}

}
```

Là encore, si une erreur survient, on la dirige correctement et on met fin à l’exécution de la méthode.

- `http.DefaultTransport`
- `transport.RoundTrip(req)`

Ce sont des méthodes de la librairie `net/http` et permettent d’exécuter une seule requête web.

Maintenant que la requête a été effectuée et que son retour est récupéré, il faut maintenant la donner à notre réponse et notre reverse proxy sera complet.

```
func proxy(c *gin.Context) {
	
	...
```

```
	for headerKey, headerValues := range resp.Header {
		for _, headerValue := range headerValues {
			c.Header(headerKey, headerValue)
		}
	}
	defer resp.Body.Close()
	bufio.NewReader(resp.Body).WriteTo(c.Writer)
	return
}
```

Ce que nous faisons ici est :

- Nous récupèrerons l'en-tête de notre réponse et les passons à notre contexte (notre vraie réponse).
- Nous passons le body de notre réponse à celui de notre retour aussi.

C’est partit pour tester tout ça !

On lance notre application et on se rend sur notre adresse `localhost:8080`

Untitled

Untitled

Untitled

Load Balancer

Surprise ....

Revision #1
Created 9 May 2022 19:31:50 by Noé Larrieu-Lacoste
Updated 9 May 2022 19:32:39 by Noé Larrieu-Lacoste