Introduction à Go – TD5: JaaS, le retour

Franklin ”Snaipe” Mathieu

2023

JaaS (voir TD4) est devenu un succès mondial et vous commencez à recevoir beaucoup de requêtes venant de scripts et utilisateurs (souvent barbus) en ligne de commande.

Malheureusement, vous avez des ressources limitées, et si cette tendance continue, vous n'aurez plus moyen de continuer ce service. Pour éviter cela, nous allons implémenter quelques mesures pour drastiquement réduire le problème:

- Ajouter des timeouts
- Ajouter des mesures de rate-limiting
- Limiter le nombre de processus jq simultanés
- Implémenter un cache

1 C'est parti

Nous travaillerons dans le dossier de jaas que vous avez créé lors du TD4.

1.1 Rate-limiting

Le rate-limiting est une technique qui consiste à limiter le nombre de requêtes autorisées par seconde par ”sujet” (par exemple une adresse IP). Typiquement, un service REST va appliquer du rate-limiting sur son API et retourner un statut `429 Too Many Requests` dans sa réponse.

Echo propose déjà cette fonctionalité sous forme de middleware; pour en faire usage:

```
package main

import (
        ...

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

func main() {
        ...
        e := echo.New()
        ...

        e.Use(middleware.RateLimiter(middleware.NewRateLimiterMemoryStore(20)))

        ...
        e.Start(...)
}
```

La documentation du middleware.RateLimiter est disponible sur `https://echo.labstack.com/docs/middleware/rate-limiter`.

Echo propose divers middlewares qui pourraient vous être utiles. La liste complête est disponible sur `https://echo.labstack.com/docs/category/middleware`.

1.2 Timeouts

Certains utilisateurs envoient des filtres jq qui ne terminent pas, et consomment des ressources ad vitam eternam. Par exemple, le filtre `repeat(null)` ne se termine jamais et produit continuellement des `null` dans la sortie:

```
$ jq -n 'repeat(null)' > /dev/null
<bloque pour toujours>
^C
$
```

Il est nécessaire d'appliquer un timeout sur l'exécution de Service.Post() pour combattre ce genre d'abus.

Vous avez déjà vu comment utiliser un context.Context, mais pour rappel:

```
import (
        "context"
        "time"
)

...

// Créer un contexte qui expire dans 1 minute
ctx, stop := context.WithTimeout(context.Background(), 1 * time.Minute)
defer stop()

// Bloque jusqu'à l'expiration du contexte
<-ctx.Done()
```

Une fonction `os/exec.CommandContext` existe pour contrôler la durée de vie d'une commande.

Modifiez le paquet jq et votre implémentation pour que vous puissiez passer un contexte lors de l'invocation de jq, puis Service.Post pour mettre un timeout de votre choix.

Points bonus si:

1. Le changement est backward-compatible dans le paquet jq
2. La durée du timeout est configurable par l'opérateur du service jaas

NOTE: Évitez d'utiliser le middleware Timeout d'Echo. La raison est expliquée ici: `https://github.com/labstack/echo/blob/98a523756d875bc13475bcb6237f09e771cbe321/middleware/timeout.go#L11-L56`.

1.3 Limiter le nombre de processus jq simultanés

La machine/VM/conteneur faisant tourner le service jaas ne peut faire tourner qu'une quantité limitée de processus jq à la fois avant que les performances ne se dégradent (voire même finisse par mettre l'environnement à genoux).

La manière la plus simple de régler le problème est d'introduire une sémaphore ou pool pour limiter le nombre de filtres actifs à la fois.

Pour rappel, un moyen simple de faire ça est d'utiliser un channel:

```
const maxRunners = 1000

pool := make(chan struct{}, maxRunners)

for i := 0; i < 1_000_000; i++ {
        pool <- struct{}{} // Aquérir un slot dans la pool
        go func() {
                defer func() {
                        <-pool // Libérer le slot acquis
                }()

                Work()
        }()
}
```

Ici, nous lançons la fonction Work un million de fois de manière concurrente, mais nous restreignons ces appels sur 1000 goroutines.

Modifiez votre Service et Service.Post pour pouvoir faire de même. Points bonus si le nombre limite de goroutines est configurable par l'opérateur du service JaaS (par exemple en ligne de commande).

1.4 Mise en cache

En analysant le traffic vers votre service, vous vous apercevez qu'une fraction non-négligeable des requêtes possède exactement les mêmes entrées: même filtre, même body, même arguments et options.

Nous allons adresser le problème avec l'utilisation d'un cache. L'idée est que lorsque vous recevez une requête POST, vous pouvez utiliser les inputs comme clé de cache, et:

- Si aucun résultat ne correspond à la clé, on lance le filtre, on récupère le résultat, et on le stocke dans le cache sous cette clé.
- Si il y a un résultat associé à la clé, ça veut dire qu'on a déjà lancé le filtre précédemment, et on peut retourner ce résultat immédiatement sans lancer jq.

Par exemple, imaginons que vous avez une fonction pour calculer le hash d'une string:

```
import (
        "crypto/sha256"
        "crypto/md5"
        "hash"
)

func Hash(data string, alg string) string {
        var h hash.Hash
        switch alg {
        case "sha256":
                h = sha256.New()
        case "md5":
                h = md5.New()
        default:
                panic("unsupported algorithm "+alg)
        }
        io.WriteString(h, alg)
        return fmt.Sprintf("%x", string(h.Sum(nil)))
}
```

Cette fonction est déterministe: appeler `Hash("foo", "sha256")` retournera systématiquement la valeur `2c26b46b68ffc68ff99b453c1d30413413422d706483bfa0f98a5e886266e7ae`.

Calculer le hash d'une valeur peut parfois être demandant en ressources, mais étant donné que la fonction est déterministe, nous pouvons mettre en place un cache très simple:

```
type Input struct {
        Data string
        Alg  string
}

cache := map[Input]string{}

hashWithCache := func(in Input) string {
        res, ok := cache[in]
        if !ok {
                res = Hash(in.Data, in.Alg)
                cache[in] = res
        }
        return res
}

inputs := []Input{
        {Data: "foo", Alg: "sha256"},
        {Data: "bar", Alg: "md5"},
        {Data: "foo", Alg: "sha256"},
}
for _, in := range inputs {
        fmt.Println(hashWithCache(in))
}
```

En pratique, ce cache a quelques problèmes: il n'est pas concurrent (le type map n'est pas protégé contre les accès concurrents), et il n'expire jamais, du coup il continuera à grandir perpétuellement.

Néanmoins, cet exemple devrait vous donner une idée de la fonctionalité attendue.

Créez un nouveau type dans example.com/jaas nommé InMemoryCache:

```
type InMemoryCache[K comparable, V any] struct {
        // FIXME
}

func NewInMemoryCache[K comparable, V any]() *InMemoryCache[K, V] {
        // FIXME
        return nil
}

// Set assigns the specified value to the specified key in the cache, with
// an expiration of ttl.
func (cache *InMemoryCache[K, V]) Set(key K, value V, ttl time.Duration) {
        // FIXME
}

// Get retrieves the value in the cache for the specified key if it exists,
// as well as whether the value was found.
func (cache *InMemoryCache[K, V]) Get(key K) (value V, found bool) {
        // FIXME
        return
}

// Expire expires the value associated with the specified key, if any.
func (cache *InMemoryCache[K, V]) Expire(key K) {
        // FIXME
}
```

InMemoryCache est un type générique, avec en paramètre de type K pour le type de la clé, et V pour le type de la valeur. K est restreint à l'interface `comparable` pour s'assurer que deux valeurs de type K peuvent être comparées avec l'opérateur ==.

Écrivez des tests pour ce nouveau type:

```
package main

import (
        "testing"
)

func TestInMemoryCache(t *testing.T) {
        stringCache := NewInMemoryCache[string, string]()
        ...
}
```

Ajoutons aussi une interface pour faire abstraction de ces méthodes:

```
type Cache[K comparable, V any] interface {
        Set(key K, value V, ttl time.Duration)
        Get(key K) (value V, found bool)
        Expire(key K)
}
```

Cette abstraction vous permettra d'avoir une interface simple pour mettre en cache des valeurs dans votre service. Modifiez votre type Service pour qu'il contienne un champ Cache:

```
type RunParams struct {
        Filter  string `form:"filter"`
        Data    string `form:"data"`
        Options string `form:"options"`
        Args    string `form:"args"`
}

type Service {
        ...
        Cache Cache[RunParams, string]
        ...
}

func (svc *Service) Post(c echo.Context) error {
        var params RunParams
        c.Bind(&params)
        ...
}
```

Vous noterez que la variable params de Service.Post, qui avait un type anonyme, a eu son type bougé dans un type global RunParams, pour qu'il soit réutilisable dans le champ Cache.

Puis, dans le main:

```
func main() {
        ...
        svc := Service{
                Cache: NewInMemoryCache[RunParams, string](),
        }
        ...
        e.POST("/", svc.Post)
        ...
}
```

Il ne vous reste qu'à utiliser `svc.Cache` depuis la méthode Service.Post().