AWS and Go

2017-08-07

AWS owns the repository aws/aws-sdk-go, the Go version for its own API. There are several 3rd party libraries for many of the services that it offers, but there are some advantages in using the former:

- it's maintained by AWS itself
- it offers the most complete set of services (all of them)
- it's created programmatically from AWS API, so it's less prone to human error
- it's always up to date with the latest API changes

Also:

- there's a dedicated Gitter chat
- and a tag for Stack Overflow

Credentials

The first step in order to use AWS with Go is creating an user for your use case using IAM, and create an access key for that user. This will produce an access key and a secret that you will need in order to access the services.

The package requires that both values are stored in two environmental variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`. You can choose to store this variable in your environment directly or you can read them from a configuration file and set them in the Go runtime directly, or you can use the NewStaticCredentials function in your configuration:

```go
package main

import (
    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/credentials"
    "github.com/aws/aws-sdk-go/aws/endpoints"
    "github.com/aws/aws-sdk-go/aws/session"
)

var config struct { AccessKeyID, SecretAccessKey string }

var s *session.Session

func init() {
    // Load Configuration
}

func main() {
    // Environment
    os.Setenv("AWS_ACCESS_KEY_ID", config.AccessKeyID)
    os.Setenv("AWS_SECRET_ACCESS_KEY", config.SecretAccessKey)
    s = session.New(&aws.Config{
        Region: aws.String(endpoints.EuWest1RegionID),
    })
    // OR
    s = session.New(&aws.Config{
        Region: aws.String(endpoints.EuWest1RegionID),
        Credentials: credentials.NewStaticCredentials(
            config.AccessKeyID,
            config.SecretAccessKey,
            "", // a token will be created when the session it's used.
        ),
    })
}
```

If you want to have the same credentials in your instances and in all the AWS clients it's a good idea to use the environment variables, instead if you planning on specific keys for each client/instance the second choice is more reasonable.

Sample usage: video storage

Amazon Simple Storage Service (S3) is an object storage designed for scale, performance and durability. In the following example we'll use it to create a simple web app that offers an API to store and retrieve video files.

Permissions

Once you have your credentials, you can use it with every single service available. Give your user the permissions to the service you want to use with IAM and you're ready to go. It's a good practice to create a Group with permissions, then assign the user to the group. For our use case we create a `S3` Group with the `AmazonS3FullAccess` permission and we add our user to that group.

Endpoints

We will use two endpoints:

- `upload` will receive the file via POST and save it to S3
- `download` will recover and return the file

We will be using gin instead of the standard `net/http` because it adds a lot of utilities for a negligible overhead.

Upload We assign a key to the request, if it does not exists and upload the file to S3.

```go
func uploadHandler(c *gin.Context) {
    var key = c.Param("key")
    if key == "" {
        key = ulid.MustNew(ulid.Timestamp(time.Now()), entropy).String()
    }
    b, err := ioutil.ReadAll(c.Request.Body)
    if err != nil {
        log.Printf("%s: upload error (%s)", key, err)
        c.JSON(http.StatusInternalServerError, gin.H{"error": "internal error"})
        return
    }
    reader := bytes.NewReader(b)
    _, err = client.PutObject(&s3.PutObjectInput{
        Bucket:        aws.String(Bucket),
        Key:           aws.String(key),
        Body:          reader,
        ContentLength: aws.Int64(reader.Size()),
    })
    if err != nil {
        log.Printf("%s: put error (%s)", key, err)
        c.JSON(http.StatusInternalServerError, gin.H{"error": "internal error"})
        return
    }
    c.JSON(http.StatusCreated, gin.H{"key": key})
}
```

Download We retrieve the file from S3 and we return it to the user. In order to generate an unique ID we use the ULID package.

```go
func downloadHandler(c *gin.Context) {
    var key = c.Param("key")
    obj, err := client.GetObject(&s3.GetObjectInput{
        Bucket: aws.String(Bucket),
        Key:    aws.String(key),
    })
    if err != nil {
        log.Printf("%s: get error (%s)", key, err)
        c.JSON(http.StatusInternalServerError, gin.H{"error": "internal error"})
        return
    }
    defer obj.Body.Close()

    _, err = io.Copy(c.Writer, obj.Body)
    if err != nil {
        log.Printf("%s: download error (%s)", key, err)
        c.JSON(http.StatusInternalServerError, gin.H{"error": "internal error"})
        return
    }
}
```

Check `aws_v1.go` to see the full example.

Extra: add some caching

Even if S3 is fast, the user shouldn't wait to download the file each time. We can easily build a mechanism that downloads the file in a temporary directory and deletes it after a while. The idea is to download the file in a directory and delete it with a `time.Timer`, every time a file is requested again before its deletion the timer is reset.

```
var cache struct {
    sync.Mutex
    timers map[string]*time.Timer
    root   string
    client *s3.S3
}

cache.timers[key] = e.AfterFunc(keep, func() {
    cache.Lock()
    delete(cache.timers, key)
    os.Remove(cachePath(key))
    cache.Unlock()
})
```

The full code is available at `aws_v2.go`.

Beside reduction in response time, this approach grants a huge advantage for our use case. We are using the `http.ServeFile` method, that calls `http.ServeContent` under the hood. As we can see it accepts an `io.ReadSeeker` (an `*os.File` in our case), because it parses the `Range` header, and returns just the selected range, instead of the whole file. This is perfect for a big video file because it makes possible to start playing the video before the whole content is downloaded, or allows you to skip from a part to another. The HTTP code for the response it's not `200 - OK`, but it's `206 - Partial Content` (more info on 206).