AWS SDK for Go

Developer Guide

Table of Contents

What is the AWS SDK for Go?

The AWS SDK for Go provides APIs and utilities that developers can use to build Go applications that use AWS services, such as Amazon Elastic Compute Cloud (Amazon EC2) and Amazon Simple Storage Service (Amazon S3).

The SDK removes the complexity of coding directly against a web service interface. It hides a lot of the lower-level plumbing, such as authentication, request retries, and error handling.

The SDK also includes helpful utilities. For example, the Amazon S3 download and upload manager can automatically break up large objects into multiple parts and transfer them in parallel.

About the AWS SDK for Go Developer Guide

Use the developer guide to help you install, configure, and use the SDK. The guide provides configuration information, sample code, and an introduction to the SDK utilities.

- For information about everything you need before you start using the AWS SDK for Go, see Setting Up (p. 2).
- For code examples, see SDK for Go Code Examples (p. 20).
- For information about the SDK utilities, see SDK Utilities (p. 123).
- For information about the types and functionality provided by the library, see the AWS SDK for Go API Reference.
- To view a video introduction of the SDK and a sample application demonstration, see AWS SDK For Go: Gophers Get Going with AWS from AWS re:Invent 2015.

Setting Up

The AWS SDK for Go requires Go 1.5 or later. You can view your current version of Go by running the `go version` command. For information about installing or upgrading your version of Go, see https://golang.org/doc/install.

Install the AWS SDK for Go

To install the SDK and its dependencies, run the following Go command:

```
go get -u github.com/aws/aws-sdk-go/...
```

If you set the Go vendor experiment environment variable to `1`, you can use the following command to get the SDK. The SDK's runtime dependancies are vendored in the `vendor/` folder.

```
go get -u github.com/aws/aws-sdk-go
```

Get AWS Credentials

The AWS SDK for Go requires AWS access keys to sign requests you send to AWS. This is how AWS authenticates your requests. AWS access keys consist of an access key ID and a secret access key.

Depending on the scenario, you can create IAM users to generate long-term access keys or IAM roles to generate temporary access keys. For more information about how and when to generate access keys, and which type to use for your scenario, see Best Practices for Managing AWS Access Keys.

To set up your credentials for use with the AWS SDK for Go, see SDK Configuration (p. 4).

Packages

After you have installed the SDK, you import AWS packages into your Go applications to use the SDK, as shown in the following example:

```
import "github.com/aws/aws-sdk-go/service/s3"
```

SDK Configuration

In the AWS SDK for Go, you can configure settings for service clients, such as the log level and maximum number of retries. Most settings are optional; however, for each service client, you must specify a region and your credentials. The SDK uses these values to send requests to the correct AWS region and sign requests with the correct credentials. You can specify these values as part of a session or as environment variables.

Specifying the Region

When you specify the region, you specify where to send requests, such as `us-west-2` or `us-east-1`. The SDK does not select a default region. For a list of regions for each service, see Regions and Endpoints in the *Amazon Web Services General Reference*.

To specify the region, set the `AWS_REGION` environment variable or specify it in a session. If you do both, the SDK will always use the region you specified in the session.

The following examples show you how to configure the environment variable.

Linux, OS X, or Unix

```
$ export AWS_REGION=us-west-2
```

Windows

```
> set AWS_REGION=us-west-2
```

The following snippet specifies the region in a session:

```
sess, err := session.NewSession(&aws.Config{Region: aws.String("us-west-2")})
```

Specifying Credentials

The AWS SDK for Go requires credentials (an access key and secret access key) to sign requests to AWS. You can specify your credentials in several different locations, depending on your particular use case. For information about obtaining credentials, see Setting Up (p. 2).

When you initialize a new service client without providing any credential arguments, the SDK uses the default credential provider chain to find AWS credentials. The SDK uses the first provider in the chain that returns credentials without an error. The default provider chain looks for credentials in the following order:

1. Environment variables.
2. Shared credentials file.
3. If your application is running on an Amazon EC2 instance, IAM role for Amazon EC2.

The SDK detects and uses the built-in providers automatically, without requiring manual configurations. For example, if you use IAM roles for Amazon EC2 instances, your applications will automatically use the instance's credentials. You don't need to manually configure credentials in your application.

As a best practice, AWS recommends that you specify credentials in the following order:

1. Use IAM roles for Amazon EC2 (if your application is running on an Amazon EC2 instance).

   IAM roles provide applications on the instance temporary security credentials to make AWS calls. IAM roles provide an easy way to distribute and manage credentials on multiple Amazon EC2 instances.
2. Use a shared credentials file.

   This credentials file is the same one used by other SDKs and the AWS CLI If you're already using a shared credentials file, you can use it for this purpose, too.
3. Use environment variables.

   Setting environment variables is useful if you're doing development work on a machine other than an Amazon EC2 instance.
4. Hard-code credentials (not recommended).

   Hard-coding credentials in your application can make it difficult to manage and rotate those credentials. Use this method for small personal scripts or testing purposes only. Do not submit code with credentials to source control.

IAM Roles for Amazon EC2 Instances

If you are running your application on an Amazon EC2 instance, you can use the instance's IAM role to get temporary security credentials to make calls to AWS.

If you have configured your instance to use IAM roles, the SDK will use these credentials for your application automatically. You don't need to manually specify these credentials.

Shared Credentials File

A credential file is a plaintext file that contains your access keys. The file must be on the same machine on which you're running your application. The file must be named `credentials` and located in the `.aws/` folder in your home directory. The home directory can vary by operating system. In Windows, you can refer to your home directory by using the environment variable `%UserProfile%`. In Unix-like systems, you can use the environment variable `$HOME` or `~` (tilde).

If you already use this file for other SDKs and tools (like the AWS CLI), you don't need to change anything to use the files in this SDK. If you use different credentials for different tools or applications, you can use *profiles* to configure multiple access keys in the same configuration file.

Creating the Credentials File

If you do not have a shared credentials file (`.aws/credentials`), you can use any text editor to create one in your home directory. Add the following content to your credentials file, replacing `<YOUR_ACCESS_KEY_ID>` and `<YOUR_SECRET_ACCESS_KEY>` with your credentials:

```
[default]
aws_access_key_id = <YOUR_ACCESS_KEY_ID>
aws_secret_access_key = <YOUR_SECRET_ACCESS_KEY>
```

The `[default]` heading defines credentials for the default profile, which the SDK will use unless you configure it to use another profile.

You can also use temporary security credentials by adding the session tokens to your profile, as shown in the following example:

```
[temp]
aws_access_key_id = <YOUR_TEMP_ACCESS_KEY_ID>
aws_secret_access_key = <YOUR_TEMP_SECRET_ACCESS_KEY>
aws_session_token = <YOUR_SESSION_TOKEN>
```

Specifying Profiles

You can include multiple access keys in the same configuration file by associating each set of access keys with a profile. For example, in your credentials file, you can declare multiple profiles:

```
[default]
aws_access_key_id = <YOUR_DEFAULT_ACCESS_KEY_ID>
aws_secret_access_key = <YOUR_DEFAULT_SECRET_ACCESS_KEY>

[test-account]
aws_access_key_id = <YOUR_TEST_ACCESS_KEY_ID>
aws_secret_access_key = <YOUR_TEST_SECRET_ACCESS_KEY>

[prod-account]
; work profile
aws_access_key_id = <YOUR_PROD_ACCESS_KEY_ID>
aws_secret_access_key = <YOUR_PROD_SECRET_ACCESS_KEY>
```

By default, the SDK checks the `AWS_PROFILE` environment variable to determine which profile to use. If no `AWS_PROFILE` variable is set, the SDK uses the default profile.

If you have an application named `myapp` that uses the SDK, you can run it with the test credentials by setting the variable to `test-account myapp`, as shown in the following command:

```
$ AWS_PROFILE=test-account myapp
```

You can also use the SDK to select a profile by specifying `os.Setenv("AWS_PROFILE", test-account)` before constructing any service clients or by manually setting the credential provider, as shown in the following example:

```
sess, err := session.NewSession(&aws.Config{
    Region:      aws.String("us-west-2"),
    Credentials: credentials.NewSharedCredentials("", "test-account"),
})
```

In addition, checking if your credentials have been found is fairly easy.

```
_, err := sess.Config.Credentials.Get()
```

If `ChainProvider` is being used, set `CredentialsChainVerboseErrors` to `true` in the session config.

> **Note**
> If you specify credentials in environment variables, the SDK will always use those credentials, no matter which profile you specify.

Environment Variables

By default, the SDK detects AWS credentials set in your environment and uses them to sign requests to AWS. That way you don't need to manage credentials in your applications.

The SDK looks for credentials in the following environment variables:

- `AWS_ACCESS_KEY_ID`
- `AWS_SECRET_ACCESS_KEY`
- `AWS_SESSION_TOKEN` (optional)

The following examples show how you configure the environment variables.

Linux, OS X, or Unix

```
$ export AWS_ACCESS_KEY_ID=YOUR_AKID
$ export AWS_SECRET_ACCESS_KEY=YOUR_SECRET_KEY
$ export AWS_SESSION_TOKEN=TOKEN
```

Windows

```
> set AWS_ACCESS_KEY_ID=YOUR_AKID
> set AWS_SECRET_ACCESS_KEY=YOUR_SECRET_KEY
> set AWS_SESSION_TOKEN=TOKEN
```

Hard-Coded Credentials in an Application (not recommended)

> **Warning**
> Do not embed credentials inside an application. Use this method only for testing purposes.

You can hard-code credentials in your application by passing the access keys to a configuration instance, as shown in the following snippet:

```
sess, err := session.NewSession(&aws.Config{
    Region:      aws.String("us-west-2"),
    Credentials: credentials.NewStaticCredentials("AKID", "SECRET_KEY", "TOKEN"),
})
```

Other Credentials Providers

The SDK provides other methods for retrieving credentials in the `aws/credentials` package. For example, you can retrieve temporary security credentials from AWS Security Token Service or credentials from encrypted storage. For more information, see Credentials.

Configuring a Proxy

If you cannot directly connect to the Internet, you can use Go-supported environment variables (`HTTP_PROXY`) or create a custom HTTP client to configure your proxy. Use the Config.HTTPClient struct to specify a custom HTTP client. For more information about how to create an HTTP client to use a proxy, see the Transport struct in the Go `http` package.

Sessions

In the AWS SDK for Go, a session is an object that contains configuration information for service clients (p. 13), which you use to interact with AWS services. For example, sessions can include information about the region where requests will be sent, which credentials to use, or additional request handlers. Whenever you create a service client, you must specify a session. For more information about sessions, see the session package in the AWS SDK for Go API Reference.

Sessions can be shared across all service clients that share the same base configuration. The Session is built from the SDK's default configuration and request handlers.

Sessions should be cached when possible, because creating a new Session will load all configuration values from the environment, and config files each time the Session is created. Sharing the Session value across all of your service clients will ensure the configuration is loaded the fewest number of times possible.

Concurrency

Sessions are safe to use concurrently as long as the Session is not being modified. The SDK will not modify the Session once the Session has been created. Creating service clients concurrently from a shared Session is safe.

Sessions with Shared Config

Sessions can be created using the method above that will only load the additional config if the `AWS_SDK_LOAD_CONFIG` environment variable is set. Alternatively you can explicitly create a Session with shared config enabled. To do this you can use `NewSessionWithOptions` to configure how the Session will be created. Using the `NewSessionWithOptions` with `SharedConfigState` set to `SharedConfigEnabled` will create the session as if the `AWS_SDK_LOAD_CONFIG` environment variable was set.

Creating Sessions

When creating `Session` optional `aws.Config` values can be passed in that will override the default, or loaded config values the Session is being created with. This allows you to provide additional, or case based, configuration as needed.

By default `NewSession` will only load credentials from the shared credentials file (`~/.aws/credentials`). If the `AWS_SDK_LOAD_CONFIG` environment variable is set to a truthy value the Session will be created from the configuration values from the shared config (`~/.aws/config`) and shared credentials (`~/.aws/credentials`) files. See the section Sessions from Shared Config for more information.

Create a Session with the default config and request handlers. With credentials region, and profile loaded from the environment and shared config automatically. Requires the `AWS_PROFILE` to be set, or `default` is used.

```
sess, err := session.NewSession()
```

The SDK provides a default configuration, which is used by all sessions unless you override a field. For example, you can specify an AWS region when you create a session by using the `aws.Config` struct. For more information about the fields you can specify, see the aws.Config in the AWS SDK for Go API Reference.

```
sess, err := session.NewSession(&aws.Config{
    Region: aws.String("us-east-1")},
)
```

Create an Amazon S3 client instance from a session:

```
sess, err := session.NewSession()
if err != nil {
    // Handle Session creation error
}
svc := s3.New(sess)
```

Create Session With Option Overrides

In addition to `NewSession`, Sessions can be created using `NewSessionWithOptions`. This func allows you to control and override how the Session will be created through code instead of being driven by environment variables only.

Use NewSessionWithOptions when you want to provide the config profile, or override the shared config state (AWS_SDK_LOAD_CONFIG).

```
// Equivalent to session.New
sess, err := session.NewSessionWithOptions(session.Options{})

// Specify profile to load for the session's config
sess, err := session.NewSessionWithOptions(session.Options{
     Profile: "profile_name",
})

// Specify profile for config and region for requests
sess, err := session.NewSessionWithOptions(session.Options{
     Config: aws.Config{Region: aws.String("us-east-1")},
     Profile: "profile_name",
})

// Force enable Shared Config support
sess, err := session.NewSessionWithOptions(session.Options{
    SharedConfigState: SharedConfigEnable,
})
```

```
// Assume an IAM role with MFA promting for token code on stdin.
sess := session.Must(session.NewSessionWithOptions(session.Options{
    AssumeRoleTokenProvider: stscreds.StdinTokenProvider,
    SharedConfigState: SharedConfigEnable,
}))
```

Deprecated New

The New function has been deprecated because it does not provide good way to return errors that occur when loading the configuration files and values. Because of this, NewSession was created so errors can be retrieved when creating a session fails.

Shared Config Fields

By default the SDK will only load the shared credentials file's (~/.aws/credentials) credentials values, and all other config is provided by the environment variables, SDK defaults, and user provided aws.Config values.

If the AWS_SDK_LOAD_CONFIG environment variable is set, or SharedConfigLoadEnable option is used to create the Session the full shared config values will be loaded. This includes credentials, region, and support for assume role. In addition the Session will load its configuration from both the shared config file (~/.aws/config) and shared credentials file (~/.aws/credentials). Both files have the same format.

If both config files are present the configuration from both files will be read. The Session will be created from configuration values from the shared credentials file (~/.aws/credentials) over those in the shared credentials file (~/.aws/config).

See the session package's documentation for more information on shared config setup.

Environment Variables

When a Session is created several environment variables can be set to adjust how the SDK functions, and what configuration data it loads when creating Sessions. All environment values are optional, but some values like credentials require multiple of the values to set or the partial values will be ignored. All environment variable values are strings unless otherwise noted.

See the session package's documentation for more information on environment variable setup.

Adding Request Handlers

You can add handlers to a session for processing HTTP requests. All service clients that use the session inherit the handlers. For example, the following handler logs every request and its payload made by a service client:

```
// Create a session, and add additional handlers for all service
// clients created with the Session to inherit. Adds logging handler.
sess, err := session.NewSession()
sess.Handlers.Send.PushFront(func(r *request.Request) {
    // Log every request made and its payload
    logger.Println("Request: %s/%s, Payload: %s",
        r.ClientInfo.ServiceName, r.Operation, r.Params)
})
```

Copying a Session

You can use the Copy method to create copies of sessions. Copying sessions is useful when you want to create multiple sessions that have similar settings. Each time you copy a session, you can specify different values for any field. For example, the following snippet copies the `sess` session while overriding the `Region` field to `us-east-1`:

```
usEast1Sess := sess.Copy(&aws.Config{Region: aws.String("us-east-1")})
```

Using AWS Services

To make calls to an AWS service, you must first construct a service client instance with a session. A service client provides low-level access to every API action for that service. For example, you create an Amazon S3 service client to make calls to Amazon S3.

When you call service operations, you pass in input parameters as a struct. A successful call usually results in an output struct that you can use. For example, after you successfully call an Amazon S3 create bucket action, the action returns an output struct with the bucket's location.

For the list of service clients, including their methods and parameters, see the AWS SDK for Go API Reference.

Constructing a Service

To construct a service client instance, use the NewSession() function. The following example creates an Amazon S3 service client.

```
sess, err := session.NewSession()
if err != nil {
    fmt.Println("Error creating session ", err)
    return
}
svc := s3.New(sess)
```

After you have a service client instance, you can use it to call service operations. For more information about configurations, see SDK Configuration (p. 4).

When you create a service client, you can pass in custom configurations so that you don't need to create a session for each configuration. The SDK merges the two configurations, overriding session values with your custom configuration. For example, in the following snippet, the Amazon S3 client uses the `mySession` session but overrides the `Region` field with a custom value (`us-west-2`):

```
svc := s3.New(mySession, aws.NewConfig().WithRegion("us-west-2"))
```

Service Operation Calls

You can call a service operation directly or with its request form. When you call a service operation, the SDK synchronously validates the input, builds the request, signs it with your credentials, sends it to AWS, and then gets a response or an error. In most cases, you can call service operations directly.

Calling Operations

Calling the operation will sync as the request is built, signed, sent, and the response is received. If an error occurs during the operation, it will be returned. The output or resulting structure won't be valid.

For example, to call the Amazon S3 GET Object API, use the Amazon S3 service client instance and call its GetObject method:

```
result, err := s3Svc.GetObject(&s3.GetObjectInput{...})
// result is a *s3.GetObjectOutput struct pointer
// err is a error which can be cast to awserr.Error.
```

Passing Parameters to a Service Operation

When calling an operation on a service, you pass in input parameters as option values, similar to passing in a configuration. For example, to retrieve an object, you must specify a bucket and the object's key by passing in the following parameters to the `GetObject` method:

```
svc := s3.New(session.New())
svc.GetObject(&s3.GetObjectInput{
    Bucket: aws.String("bucketName"),
    Key:    aws.String("keyName"),
})
```

Each service operation has an associated input struct and, usually, an output struct. The structs follow the naming pattern *OperationName* `Input` and *OperationName* `Output`.

For more information about the parameters of each method, see the service client documentation in the AWS SDK for Go API Reference.

Calling Operations with the Request Form

Calling the request form of a service operation, which follows the naming pattern *OperationName* `Request`, provides a simple way to control when a request is built, signed, and sent. Calling the request form immediately returns a request object. The request object output is a struct pointer that is not valid until the request is sent and returned successfully.

Calling the request form can be useful when you want to construct a number of pre-signed requests, such as pre-signed Amazon S3 URLs. You can also use the request form to modify how the SDK sends a request.

The following example calls the request form of the `GetObject` method. The Send method signs the request before sending it.

```
req, result := s3Svc.GetObjectRequest(&s3.GetObjectInput{...})
// result is a *s3.GetObjectOutput struct pointer, not populated until req.Send() returns
// req is a *aws.Request struct pointer. Used to Send request.
if err := req.Send(); err != nil {
    // process error
```

```
    return
}
// Process result
```

Handling Operation Response Body

Some of the API operations' response output struct will contain a `Body` field which is an `io.ReadCloser`. If you are making request with these operations you should always make sure to call `Close` on the field.

```
resp, err := s3svc.GetObject(&s3.GetObjectInput{...})
if err != nil {
    // handle error
    return
}
// Make sure to always close the response Body when finished
defer resp.Body.Close()

decoder := json.NewDecoder(resp.Body)
if err := decoder.Decode(&myStruct); err != nil {
    // handle error
    return
}
```

Concurrently Using Service Clients

You can create goroutines that concurrently use the same service client to send multiple requests. You can use a service client with as many goroutines as you want. However, you cannot concurrently modify the service client's configuration and request handlers. If you do, the service client operations might encounter race conditions. Define service client settings before you concurrently use it.

In the following example, an Amazon S3 service client is used in multiple goroutines. The example concurrently outputs all objects in `bucket1`, `bucket2`, and `bucket3`, which are all in the same region. To make sure all objects from the same bucket are printed together, the example uses a channel.

```
sess, err := session.NewSession()
if err != nil {
    fmt.Println("Error creating session ", err)
}
var wg sync.WaitGroup
keysCh := make(chan string, 10)

svc := s3.New(sess)
buckets := []string{"bucket1", "bucket2", "bucket3"}
for _, bucket := range buckets {
    params := &s3.ListObjectsInput{
        Bucket:  aws.String(bucket),
        MaxKeys: aws.Int64(100),
    }
    wg.Add(1)
    go func(param *s3.ListObjectsInput) {
        defer wg.Done()

        err = svc.ListObjectsPages(params,
            func(page *s3.ListObjectsOutput, last bool) bool {
                // Add the objects to the channel for each page
                for _, object := range page.Contents {
                    keysCh <- fmt.Sprintf("%s:%s", *params.Bucket, *object.Key)
                }
```

```
                return true
            },
        )
        if err != nil {
            fmt.Println("Error listing", *params.Bucket, "objects:", err)
        }
    }(params)
}
go func() {
    wg.Wait()
    close(keysCh)
}()
for key := range keysCh {
    // Print out each object key as its discovered
    fmt.Println(key)
}
```

Using Pagination Methods

Typically, when you retrieve a list of items, you might need to check the output for a token or marker to confirm whether AWS returned all results from your request. If present, you use the token or marker to request the next set of results. Instead of managing these tokens or markers, you can use pagination methods provided by the SDK.

Pagination methods iterate over a list operation until the method retrieves the last page of results or until the callback function returns `false`. The names of these method use the following pattern: *OperationName* `Pages`. For example, the pagination method for the Amazon S3 list objects operation (`ListObjects`) is `ListObjectPages`.

The following example uses the `ListObjectPages` pagination method to list, at most, three pages of object keys from the `ListObject` operation. Each page consists of at least 10 keys, which is defined by the `MaxKeys` field.

```
svc, err := s3.NewSession(sess)
if err != nil {
    fmt.Println("Error creating session ", err)
}
inputparams := &s3.ListObjectsInput{
    Bucket:  aws.String("mybucket"),
    MaxKeys: aws.Int64(10),
}
pageNum := 0
svc.ListObjectsPages(inputparams, func(page *s3.ListObjectsOutput, lastPage bool) bool {
    pageNum++
    for _, value := range page.Contents {
        fmt.Println(*value.Key)
    }
    return pageNum < 3
})
```

Using Waiters

The SDK provides waiters that continuously check for completion of a job. For example, when you send a request to create an Amazon S3 bucket, you can use a waiter to check when the bucket has been successfully created. That way, subsequent operations on the bucket are done only after the bucket has been created.

The following example uses a waiter that waits until specific instances have stopped:

```
sess, err := session.NewSession(aws.NewConfig().WithRegion("us-west-2"))
if err != nil {
    fmt.Println("Error creating session ", err)
}
// Create an EC2 client.
ec2client := ec2.New(sess)
// Specify two instances to stop.
instanceIDsToStop := aws.StringSlice([]string{"i-12345678", "i-23456789"})
// Send request to stop instances.
_, err = ec2client.StopInstances(&ec2.StopInstancesInput{
  InstanceIds: instanceIDsToStop,
})
if err != nil {
  panic(err)
}
// Use a waiter function to wait until the instances are stopped.
describeInstancesInput := &ec2.DescribeInstancesInput{
  InstanceIds: instanceIDsToStop,
}
if err := ec2client.WaitUntilInstanceStopped(describeInstancesInput); err != nil {
  panic(err)
}
fmt.Println("Instances are stopped.")
```

Handling Errors

The AWS SDK for Go returns errors that satisfy the Go `error` interface type and the Error interface in the `aws/awserr` package. You can use the `Error()` method to get a formatted string of the SDK error message without any special handling.

```
if err != nil {
    if awsErr, ok := err.(awserr.Error); ok {
        // process SDK error
    }
}
```

Errors returned by the SDK are backed by a concrete type that will satisfy the `awserr.Error` interface. The interface has the following methods, which provide classification and information about the error.

- `Code` returns the classification code by which related errors are grouped.
- `Message` returns a description of the error.
- `OrigErr` returns the original error of type `error` that is wrapped by the `awserr.Error` interface, such as a standard library error or a service error.

Handling Specific Service Error Codes

The example below demonstrates how to handle error codes that you encounter while using the AWS SDK for Go. The example assumes you have already set up and configured the SDK (that is, all required packages are imported and your credentials and region are set). For more information, see Setting Up (p. 2) and SDK Configuration (p. 4).

This example highlights how you can use the *awserr.Error* type to perform logic based on specific error codes returned by service API operations.

In this example the *S3 GetObject* API operation is used to request the contents of an object in S3. The example handles the *NoSuchBucket* and *NoSuchKey* error codes, printing custom messages to stderr. If any other error is received, a generic message is printed.

```
svc := s3.New(sess)
resp, err := svc.GetObject(&s3.GetObjectInput{
    Bucket: aws.String(os.Args[1]),
    Key:    aws.String(os.Args[2]),
})
```

```
if err != nil {
    // Casting to the awserr.Error type will allow you to inspect the error
    // code returned by the service in code. The error code can be used
    // to switch on context specific functionality. In this case a context
    // specific error message is printed to the user based on the bucket
    // and key existing.
    //
    // For information on other S3 API error codes see:
    // http://docs.aws.amazon.com/AmazonS3/latest/API/ErrorResponses.html
    if aerr, ok := err.(awserr.Error); ok {
        switch aerr.Code() {
        case s3.ErrCodeNoSuchBucket:
            exitErrorf("bucket %s does not exist", os.Args[1])
        case s3.ErrCodeNoSuchKey:
            exitErrorf("object with key %s does not exist in bucket %s", os.Args[2],
 os.Args[1])
        }
    }
```

You can see the complete example code on GitHub.

Additional Error Information

In addition to the `awserr.Error` interface, you might be able to use other interfaces to get more information about an error.

Specific Error Interfaces

Other packages might provide their own error interfaces. For example, the service/s3/s3manager package provides a MultiUploadFailure interface to retrieve the upload ID, which is helpful when you must manually clean up a failed multi-part upload.

```
output, err := s3manager.Upload(svc, input, opts)
if err != nil {
    if multierr, ok := err.(MultiUploadFailure); ok {
        // Process error and its associated uploadID
        fmt.Println("Error:", multierr.Code(), multierr.Message(), multierr.UploadID())
    } else {
        // Process error generically
        fmt.Println("Error:", err.Error())
    }
}
```

For more information, see the s3Manager.MultiUploadFailure interface in the AWS SDK for Go API Reference.

SDK for Go Code Examples

The AWS SDK for Go examples can help you write your own Go applications that use Amazon Web Services. The examples assume you have already set up and configured the SDK (that is, you have imported all required packages and set your credentials and region). For more information, see Setting Up (p. 2) and SDK Configuration (p. 4).

Topics

AWS SDK for Go Request Examples

The AWS SDK for Go examples can help you write your own applications. The examples assume you have already set up and configured the SDK (that is, you have imported all required packages and set your credentials and region). For more information, see Setting Up (p. 2) and SDK Configuration (p. 4).

Using context.Context with SDK Requests

In Go 1.7, the `context.Context` type was added to `http.Request`. This type provides an easy way to implement deadlines and cancellations on requests.

To use this pattern with the SDK, you call `WithContext` on the `HTTPRequest` field of the SDK's `request.Request` type, and provide your `Context value`. The following example highlights this process with a timeout on an Amazon SQS`ReceiveMessage` API call.

```
ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
defer cancel()

// SQS ReceiveMessage
params := &sqs.ReceiveMessageInput{ ... }
req, resp := s.ReceiveMessageRequest(params)
```

```
   req.HTTPRequest = req.HTTPRequest.WithContext(ctx)
   err := req.Send()
```

Using API Field Setters with SDK Requests

In addition to setting API parameters by using struct fields, you can also use chainable setters on the API operation parameter fields. This enables you to use a chain of setters to set the fields of the API struct.

```
resp, err := svc.PutObject((&s3.PutObject{}).
    SetBucket("myBucket").
    SetKey("myKey").
    SetBody(strings.NewReader("object body")).
    SetWebsiteRedirectLocation("https://example.com/something"),
)
```

You can also use this pattern with nested fields in API operation requests.

```
resp, err := svc.UpdateService((&ecs.UpdateServiceInput{}).
    SetService("myService").
    SetDeploymentConfiguration((&ecs.DeploymentConfiguration{}).
        SetMinimumHealthyPercent(80),
    ),
)
```

Amazon CloudWatch with Go Examples

Amazon CloudWatch is a web service that monitors your AWS resources and the applications you run on AWS in real time. You can use CloudWatch to collect and track metrics, which are variables you can measure for your resources and applications. CloudWatch alarms send notifications or automatically make changes to the resources you are monitoring based on rules that you define.

The AWS SDK for Go examples show you how to integrate CloudWatch into your Go applications. The examples assume you have already set up and configured the SDK (that is, you have imported all required packages and set your credentials and region). For more information, see Setting Up (p. 2) and SDK Configuration (p. 4).

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

Topics

Describing CloudWatch Alarms with Go

This example shows you how to:

- Retrieve basic information that describes your CloudWatch alarms

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

An alarm watches a single metric over a time period you specify. The alarm performs one or more actions based on the value of the metric relative to a given threshold over a number of time periods.

In this example, Go code is used to describe alarms in CloudWatch. The code uses the AWS SDK for Go to describe alarms by using this method of the `AWS.CloudWatch` client class:

- DescribeAlarms

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with CloudWatch alarms. To learn more, see Creating Amazon CloudWatch Alarms in the *CloudWatch User Guide*.

Describing Alarms

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new Amazon EC2 service client.

```
sess, err := session.NewSession()
if err != nil {
    fmt.Println("failed to create session,", err)
    return
}
```

Set up the `DescribeAlarmsInput` structure, pass it in to the `DescribeAlarms` method, and print the results.

```
svc := cloudwatch.New(sess)

params := &cloudwatch.DescribeAlarmsInput{
    ActionPrefix:    aws.String("ActionPrefix"),
    AlarmNamePrefix: aws.String("AlarmNamePrefix"),
    AlarmNames: []*string{
        aws.String("AlarmName"), // Required
        // More values...
    },
    MaxRecords: aws.Int64(1),
    NextToken:  aws.String("NextToken"),
    StateValue: aws.String("ALARM"),
}
resp, err := svc.DescribeAlarms(params)

if err != nil {
    fmt.Println(err.Error())
    return
}
fmt.Println(resp)
```

Using Alarms and Alarm Actions in CloudWatch with Go

These Go examples show you how to change the state of your Amazon EC2 instances automatically based on a CloudWatch alarm, as follows:

- Creating and enabling actions on an alarm
- Disabling actions on an alarm

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

You can use alarm actions to create alarms that automatically stop, terminate, reboot, or recover your Amazon EC2 instances. You can use the stop or terminate actions when you no longer need an instance to be running. You can use the reboot and recover actions to automatically reboot the instance.

In this example, Go code is used to define an alarm action in CloudWatch that triggers the reboot of an Amazon EC2 instance. The code uses the AWS SDK for Go to manage instances by using these methods of PutMetricAlarm type:

- PutMetricAlarm
- EnableAlarmActions
- DisableAlarmActions

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with CloudWatch alarm actions. To learn more, see Create Alarms to Stop, Terminate, Reboot, or Recover an Instance in the *CloudWatch User Guide*.

Creating and Enabling Actions on an Alarm

Create a new Go file named `create_enable_alarms.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/cloudwatch"
)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new Amazon EC2 service client.

```
func main() {
    // Load session from shared config.
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create new cloudwatch client.
    svc := cloudwatch.New(sess)
```

Create a metric alarm that will reboot an instance if its CPU utilization is greater than 70 percent.

```
    _, err := svc.PutMetricAlarm(&cloudwatch.PutMetricAlarmInput{
        AlarmName:          &os.Args[3],
        ComparisonOperator: aws.String(cloudwatch.ComparisonOperatorGreaterThanThreshold),
        EvaluationPeriods:  aws.Int64(1),
        MetricName:         aws.String("CPUUtilization"),
        Namespace:          aws.String("AWS/EC2"),
        Period:             aws.Int64(60),
        Statistic:          aws.String(cloudwatch.StatisticAverage),
        Threshold:          aws.Float64(70.0),
        ActionsEnabled:     aws.Bool(true),
        AlarmDescription:   aws.String("Alarm when server CPU exceeds 70%"),
        Unit:               aws.String(cloudwatch.StandardUnitSeconds),
```

You can use one of the default workflow actions to reboot the instance if the alarm is triggered.

```
        AlarmActions: []*string{
            aws.String(fmt.Sprintf("arn:aws:swf:us-east-1:%s:action/actions/
AWS_EC2.InstanceId.Reboot/1.0", os.Args[1])),
        },
        Dimensions: []*cloudwatch.Dimension{
            &cloudwatch.Dimension{
                Name:  aws.String("InstanceId"),
                Value: &os.Args[2],
            },
        },
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }
```

Call `EnableAlarmActions` with the new alarm for the instance.

```
    result, err := svc.EnableAlarmActions(&cloudwatch.EnableAlarmActionsInput{
        AlarmNames: []*string{
            &os.Args[3],
        },
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Alarm action enabled", result)
}
```

Disabling Actions on an Alarm

Create a new Go file named `disable_alarm.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
```

```
    "os"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/cloudwatch"
)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new Amazon EC2 service client.

```
func main() {
    // Load session from shared config.
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create new cloudwatch client.
    svc := cloudwatch.New(sess)
```

Call the `DisableAlarmActions` method to disable the actions for this alarm.

```
    result, err := svc.DisableAlarmActions(&cloudwatch.DisableAlarmActionsInput{
        AlarmNames: []*string{
            &os.Args[1],
        },
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success", result)
}
```

Getting Metrics from CloudWatch with Go

These Go examples show you how to retrieve a list of published CloudWatch metrics and publish data points to CloudWatch metrics with the AWS SDK for Go, as follows:

- Listing metrics
- Submitting custom metrics

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

Metrics are data about the performance of your systems. You can enable detailed monitoring of some resources, such as your Amazon EC2 instances, or your own application metrics.

In this example, Go code is used to get metrics from CloudWatch and to send events to CloudWatch Events. The code uses the AWS SDK for Go to get metrics from CloudWatch by using these methods of the CloudWatch type:

- ListMetrics
- PutMetricData

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with CloudWatch metrics. To learn more, see Using Amazon CloudWatch Metrics in the *CloudWatch User Guide*.

Listing Metrics

Create a new Go file named `listing_metrics.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/cloudwatch"
)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new Amazon EC2 service client.

```
func main() {
    // Load session from shared config.
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create new cloudwatch client.
    svc := cloudwatch.New(sess)
```

Call `ListMetrics`, supplying the metric name, namespace, and dimensions. Print the metrics returned in the result.

```
    result, err := svc.ListMetrics(&cloudwatch.ListMetricsInput{
        MetricName: aws.String("IncomingLogEvents"),
        Namespace:  aws.String("AWS/Logs"),
        Dimensions: []*cloudwatch.DimensionFilter{
            &cloudwatch.DimensionFilter{
                Name: aws.String("LogGroupName"),
            },
        },
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Metrics", result.Metrics)
}
```

Submitting Custom Metrics

Create a new Go file named `custom_metrics.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/cloudwatch"
)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new Amazon EC2 service client.

```
func main() {
    // Load session from shared config.
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create new cloudwatch client.
    svc := cloudwatch.New(sess)
```

Call `PutMetricData`, suppylying the metric name, unit, value, and dimensions. Print any errors, or a success message.

```
    result, err := svc.PutMetricData(&cloudwatch.PutMetricDataInput{
        MetricData: []*cloudwatch.MetricDatum{
            &cloudwatch.MetricDatum{
                MetricName: aws.String("PAGES_VISITED"),
                Unit:       aws.String(cloudwatch.StandardUnitNone),
                Value:      aws.Float64(1.0),
                Dimensions: []*cloudwatch.Dimension{
                    &cloudwatch.Dimension{
                        Name:  aws.String("UNIQUE_PAGES"),
                        Value: aws.String("URLS"),
                    },
                },
            },
        },
        Namespace: aws.String("SITE/TRAFFIC"),
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success", result)
}
```

Sending Events to Amazon CloudWatch Events with Go

These Go examples show you how to use the AWS SDK for Go to:

- Create and update a rule used to trigger an event

- Define one or more targets to respond to an event
- Send events that are matched to targets for handling

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

CloudWatch Events delivers a near real-time stream of system events that describe changes in AWS resources to any of various targets. Using simple rules, you can match events and route them to one or more target functions or streams.

In these examples, Go code is used to send events to CloudWatch Events. The code uses the AWS SDK for Go to manage instances by using these methods of the CloudWatchEvents type:

- PutRule
- PutTargets
- PutEvents

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with CloudWatch Events. To learn more, see Adding Events with PutEvents in the *CloudWatch Events User Guide*.

Tasks Before You Start

To set up and run this example, you must first complete these tasks:

1. Create a Lambda function using the hello-world blueprint to serve as the target for events. To learn how, see Step 1: Create an AWS Lambda function in the CloudWatch Events User Guide.
2. Create an IAM role whose policy grants permission to CloudWatch Events and that includes `events.amazonaws.com` as a trusted entity. For more information about creating an IAM role, see Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide*.

   Use the following role policy when creating the IAM role.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "CloudWatchEventsFullAccess",
         "Effect": "Allow",
         "Action": "events:*",
         "Resource": "*"
      },
      {
         "Sid": "IAMPassRoleForCloudWatchEvents",
         "Effect": "Allow",
         "Action": "iam:PassRole",
         "Resource": "arn:aws:iam::*:role/AWS_Events_Invoke_Targets"
      }
   ]
}
```

Use the following trust relationship when creating the IAM role.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Principal": {
            "Service": "events.amazonaws.com"
         },
         "Action": "sts:AssumeRole"
      }
   ]
}
```

Creating a Scheduled Rule

Create a new Go file named `events_schedule_rule.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/cloudwatchevents"
)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new Amazon EC2 service client.

```
func main() {
    // Load session from shared config.
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create the cloudwatch events client
    svc := cloudwatchevents.New(sess)
```

Call `PutRule`, supplying a name, ARN of the IAM role you created, and an expression defining the schedule. Print any errors, or a success message.

```
    result, err := svc.PutEvents(&cloudwatchevents.PutEventsInput{
        Entries: []*cloudwatchevents.PutEventsRequestEntry{
            &cloudwatchevents.PutEventsRequestEntry{
                Detail:     aws.String("{ \"key1\": \"value1\", \"key2\": \"value2\" }"),
                DetailType: aws.String("appRequestSubmitted"),
                Resources: []*string{
                    aws.String("RESOURCE_ARN"),
                },
                Source: aws.String("com.company.myapp"),
            },
        },
    })
```

Adding a Lambda Function Target

Create a new Go file named `events_put_targets.go`.

Call the `PutRule` method to create the rule. The method returns the ARN of the new or updated rule.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/cloudwatchevents"
)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new Amazon EC2 service client.

```
func main() {
    // Load session from shared config.
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create the cloudwatch events client
    svc := cloudwatchevents.New(sess)
```

Call `PutTargets`, supplying a name for the rule. For the target, specify the ARN of the Lambda function you created, and the ID of the rule. Print any errors, or a success message.

```
    result, err := svc.PutTargets(&cloudwatchevents.PutTargetsInput{
        Rule: aws.String("DEMO_EVENT"),
        Targets: []*cloudwatchevents.Target{
            &cloudwatchevents.Target{
                Arn: aws.String("LAMBDA_FUNCTION_ARN"),
                Id:  aws.String("myCloudWatchEventsTarget"),
            },
        },
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success", result)
}
```

Sending Events

Create a new Go file named `events_put_events.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main
```

```
import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/cloudwatchevents"
)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new Amazon EC2 service client.

```
func main() {
    // Load session from shared config.
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create the cloudwatch events client
    svc := cloudwatchevents.New(sess)
```

Call `PutEvents`, supplying key-name value pairs in the Details field, and specifying the ARN of the Lambda function you created. See PutEventsRequestEntry for a description of the fields. Print out any errors, or a success message.

```
    result, err := svc.PutEvents(&cloudwatchevents.PutEventsInput{
        Entries: []*cloudwatchevents.PutEventsRequestEntry{
            &cloudwatchevents.PutEventsRequestEntry{
                Detail:     aws.String("{ \"key1\": \"value1\", \"key2\": \"value2\" }"),
                DetailType: aws.String("appRequestSubmitted"),
                Resources: []*string{
                    aws.String("RESOURCE_ARN"),
                },
                Source: aws.String("com.company.myapp"),
            },
        },
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success", result)
}
```

Amazon DynamoDB with Go Examples

The AWS SDK for Go examples can integrate Amazon DynamoDB into your Go applications. The examples assume you have already set up and configured the SDK (that is, you have imported all required packages and set your credentials and region). For more information, see Setting Up (p. 2) and SDK Configuration (p. 4).

Listing Tables

The following example uses the DynamoDBListTables operation to list all tables for the region you specified (`us-west-2`).

```
svc := dynamodb.New(session.New(&aws.Config{Region: aws.String("us-west-2")}))
result, err := svc.ListTables(&dynamodb.ListTablesInput{})
if err != nil {
    log.Println(err)
    return
}

log.Println("Tables:")
for _, table := range result.TableNames {
    log.Println(*table)
}
```

Amazon EC2 with Go Examples

The AWS SDK for Go examples can integrate Amazon EC2 into your Go applications. The examples assume you have already set up and configured the SDK (that is, you have imported all required packages and set your credentials and region). For more information, see Setting Up (p. 2) and SDK Configuration (p. 4).

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

Topics

Creating Amazon EC2 Instances with Tags or without Block Devices with Go

This Go example shows you how to:

- Create an Amazon EC2 instance with tags or set up an instance without a block device

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenarios

In these examples, you use a series of Go routines to create Amazon EC2 instances with tags or set up an instance without a block device.

The routines use the AWS SDK for Go to perform these tasks by using these methods of the EC2 type:

- BlockDeviceMapping
- RunInstances

- CreateTags

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.

Scenario: Creating an Instance with Tags

The Amazon EC2 service has an operation for creating instances (RunInstances) and another for attaching tags to instances (CreateTags). To create an instance with tags, call both of these operations in succession. The following example creates an instance and then adds a `Name` tag to it. The Amazon EC2 console displays the value of the `Name` tag in its list of instances.

```
    svc := ec2.New(session.New(&aws.Config{Region: aws.String("us-west-2")}))
    // Specify the details of the instance that you want to create.
    runResult, err := svc.RunInstances(&ec2.RunInstancesInput{
        // An Amazon Linux AMI ID for t2.micro instances in the us-west-2 region
        ImageId:      aws.String("ami-e7527ed7"),
        InstanceType: aws.String("t2.micro"),
        MinCount:     aws.Int64(1),
        MaxCount:     aws.Int64(1),
    })

    if err != nil {
        log.Println("Could not create instance", err)
        return
    }

    log.Println("Created instance", *runResult.Instances[0].InstanceId)

    // Add tags to the created instance
    _ , errtag := svc.CreateTags(&ec2.CreateTagsInput{
        Resources: []*string{runResult.Instances[0].InstanceId},
        Tags: []*ec2.Tag{
            {
                Key:   aws.String("Name"),
                Value: aws.String("MyFirstInstance"),
            },
        },
    })
    if errtag != nil {
        log.Println("Could not create tags for instance",
runResult.Instances[0].InstanceId, errtag)
        return
    }

    log.Println("Successfully tagged instance")
```

You can add up to 10 tags to an instance in a single `CreateTags` operation.

Scenario: Creating an Image without a Block Device

Sometimes when you create an Amazon EC2 image, you might want to explicitly exclude certain block devices. To do this, you can use the `NoDevice` parameter in BlockDeviceMapping. When this parameter is set to an empty string `""`, the named device isn't mapped.

The `NoDevice` parameter is compatible only with `DeviceName`, not with any other field in `BlockDeviceMapping`. The request will fail if other parameters are present.

```
func main() {
    svc := ec2.New(session.New())
    opts := &ec2.CreateImageInput{
        Description: aws.String("image description"),
        InstanceId:  aws.String("i-abcdef12"),
        Name:        aws.String("image name"),
        BlockDeviceMappings: []*ec2.BlockDeviceMapping{
            &ec2.BlockDeviceMapping{
                DeviceName: aws.String("/dev/sda1"),
                NoDevice:    aws.String(""),
            },
            &ec2.BlockDeviceMapping{
                DeviceName: aws.String("/dev/sdb"),
                NoDevice:    aws.String(""),
            },
            &ec2.BlockDeviceMapping{
                DeviceName: aws.String("/dev/sdc"),
                NoDevice:    aws.String(""),
            },
        },
    }
    resp, err := svc.CreateImage(opts)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println("success", resp)
}
```

Managing Amazon EC2 Instances with Go

These Go examples show you how to:

- Describe Amazon EC2 instances
- Manage Amazon EC2 instance monitoring
- Start and stop Amazon EC2 instances
- Reboot Amazon EC2 instances

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

In these examples, you use a series of Go routines to perform several basic instance management operations.

The routines use the AWS SDK for Go to perform the operations by using these methods of the Amazon EC2 client class:

- DescribeInstances
- MonitorInstances
- UnmonitorInstances
- StartInstances
- StopInstances
- RebootInstances

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with the lifecycle of Amazon EC2 instances. To learn more, see Instance Lifecycle in the *Amazon EC2 User Guide for Linux Instances*.

Describing Your Instances

Create a new Go file named `describing_instances.go`.

The Amazon EC2 service has an operation for describing instances, DescribeInstances.

Import the required AWS SDK for Go packages.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/ec2"
)
```

Use the following code to create a session and Amazon EC2 client.

```
func main() {
    // Load session from shared config
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create new EC2 client
    ec2Svc := ec2.New(sess)
```

Call `DescribeInstances` to get detailed information for each instance.

```
    result, err := ec2Svc.DescribeInstances(nil)
    if err != nil {
        fmt.Println("Error", err)
    } else {
        fmt.Println("Success", result)
    }
}
```

Managing Instance Monitoring

Create a new Go file named `monitoring_instances.go`.

Import the required AWS SDK for Go packages.

```
package main

import (
    "fmt"
    "os"
```

```
    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/ec2"
)
```

To access Amazon EC2, create an EC2 client.

```
func main() {
    // Load session from shared config
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create new EC2 client
    svc := ec2.New(sess)
```

Based on the value of a command-line argument (ON or OFF), call either the `MonitorInstances` method of the Amazon EC2 service object to begin detailed monitoring of the specified instances, or the `UnmonitorInstances` method. Before you try to change the monitoring of these instances, use the `DryRun` parameter to test whether you have permission to change instance monitoring.

```
    if os.Args[1] == "ON" {
        input := &ec2.MonitorInstancesInput{
            InstanceIds: []*string{
                aws.String(os.Args[2]),
            },
            DryRun: aws.Bool(true),
        }
        result, err := svc.MonitorInstances(input)
        awsErr, ok := err.(awserr.Error)

        if ok && awsErr.Code() == "DryRunOperation" {
            input.DryRun = aws.Bool(false)
            result, err = svc.MonitorInstances(input)
            if err != nil {
                fmt.Println("Error", err)
            } else {
                fmt.Println("Success", result.InstanceMonitorings)
            }
        } else {
            fmt.Println("Error", err)
        }
    } else if os.Args[1] == "OFF" { // Turn monitoring off
        input := &ec2.UnmonitorInstancesInput{
            InstanceIds: []*string{
                aws.String(os.Args[2]),
            },
            DryRun: aws.Bool(true),
        }
        result, err := svc.UnmonitorInstances(input)
        awsErr, ok := err.(awserr.Error)
        if ok && awsErr.Code() == "DryRunOperation" {
            input.DryRun = aws.Bool(false)
            result, err = svc.UnmonitorInstances(input)
            if err != nil {
                fmt.Println("Error", err)
            } else {
                fmt.Println("Success", result.InstanceMonitorings)
            }
        } else {
            fmt.Println("Error", err)
        }
```

```
    }

}
```

Starting and Stoppping Instances

Create a new Go file named `start_stop_instances.go`.

Import the required AWS SDK for Go packages.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/ec2"
)
```

To access Amazon EC2, create an EC2 client. The user will pass in a state value of START or STOP and the instance ID.

```
func main() {
    // Load session from shared config
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create new EC2 client
    svc := ec2.New(sess)
```

Based on the value of a command-line argument (START or STOP), call either the `StartInstances` method of the Amazon EC2 service object to start the specified instances, or the `StopInstances` method to stop them. Before you try to start or stop the selected instances, use the `DryRun` parameter to test whether you have permission to start or stop them.

```
    if os.Args[1] == "START" {
        input := &ec2.StartInstancesInput{
            InstanceIds: []*string{
                aws.String(os.Args[2]),
            },
            DryRun: aws.Bool(true),
        }
        result, err := svc.StartInstances(input)
        awsErr, ok := err.(awserr.Error)

        if ok && awsErr.Code() == "DryRunOperation" {
            // Let's now set dry run to be false. This will allow us to start the instances
            input.DryRun = aws.Bool(false)
            result, err = svc.StartInstances(input)
            if err != nil {
                fmt.Println("Error", err)
            } else {
                fmt.Println("Success", result.StartingInstances)
            }
        } else { // This could be due to a lack of permissions
            fmt.Println("Error", err)
        }
```

```
    } else if os.Args[1] == "STOP" { // Turn instances off
        input := &ec2.StopInstancesInput{
            InstanceIds: []*string{
                aws.String(os.Args[2]),
            },
            DryRun: aws.Bool(true),
        }
        result, err := svc.StopInstances(input)
        awsErr, ok := err.(awserr.Error)
        if ok && awsErr.Code() == "DryRunOperation" {
            input.DryRun = aws.Bool(false)
            result, err = svc.StopInstances(input)
            if err != nil {
                fmt.Println("Error", err)
            } else {
                fmt.Println("Success", result.StoppingInstances)
            }
        } else {
            fmt.Println("Error", err)
        }
    }
}
```

Rebooting Instances

Create a new Go file named `reboot_instances.go`.

Import the required AWS SDK for Go packages.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/ec2"
)
```

To access Amazon EC2, create an EC2 client. The user will pass in a state value of START or STOP and the instance ID.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create new EC2 client
    svc := ec2.New(sess)
```

Based on the value of a command-line argument (START or STOP), call either the `StartInstances` method of the Amazon EC2 service object to start the specified instances, or the `StopInstances` method to stop them. Before you try to reboot the selected instances, use the `DryRun` parameter to test whether the instance exists and that you have permission to reboot it.

```
    input := &ec2.RebootInstancesInput{
        InstanceIds: []*string{
            aws.String(os.Args[1]),
        },
        DryRun: aws.Bool(true),
```

```
    }
    result, err := svc.RebootInstances(input)
    awsErr, ok := err.(awserr.Error)
```

If the error code is `DryRunOperation`, it means that you do have the permissions you need to reboot the instance.

```
    if ok && awsErr.Code() == "DryRunOperation" {
        input.DryRun = aws.Bool(false)
        result, err = svc.RebootInstances(input)
        if err != nil {
            fmt.Println("Error", err)
        } else {
            fmt.Println("Success", result)
        }
    } else { // This could be due to a lack of permissions
        fmt.Println("Error", err)
    }
}
```

Working with Amazon EC2 Key Pairs with Go

These Go examples show you how to:

- Describe an Amazon EC2 key pair
- Create an Amazon EC2 key pair
- Delete an Amazon EC2 key pair

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

Amazon EC2 uses public–key cryptography to encrypt and decrypt login information. Public–key cryptography uses a public key to encrypt data, then the recipient uses the private key to decrypt the data. The public and private keys are known as a key pair.

The routines use the AWS SDK for Go to perform these tasks by using these methods of the EC2 type:

- CreateKeyPair
- DeleteKeyPair
- DescribeKeyPairs

Prerequisites

- You have set up (p. 2) and configured (p. 4) the SDK.
- You are familiar with Amazon EC2 key pairs. To learn more, see Amazon EC2 Key Pairs in the *Amazon EC2 User Guide for Linux Instances*.

Describing Your Key Pairs

Create a new Go file named `ec2_describe_keypairs.go`.

Import the required AWS SDK for Go packages.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/ec2"
)
```

Use the following code to create a session and Amazon EC2 client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create an EC2 service client.
    svc := ec2.New(sess)
```

Call `DescribeKeyPairs` to get a list of key pairs and print them out.

```
    result, err := svc.DescribeKeyPairs(nil)
    if err != nil {
        exitErrorf("Unable to get key pairs, %v", err)
    }

    fmt.Println("Key Pairs:")
    for _, pair := range result.KeyPairs {
        fmt.Printf("%s: %s\n", *pair.KeyName, *pair.KeyFingerprint)
    }
}
```

The routine uses the following utility function.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Creating a Key Pair

Create a new Go file named `ec2_create_keypair.go`.

Import the required AWS SDK for Go packages.

```
package main

import (
    "fmt"
    "os"
    "path/filepath"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/ec2"
```

```
)
```

Get the key pair name passed in to the code and, to access Amazon EC2, create an EC2 client.

```
func main() {
    if len(os.Args) != 2 {
        exitErrorf("pair name required\nUsage: %s key_pair_name",
            filepath.Base(os.Args[0]))
    }
    pairName := os.Args[1]
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create an EC2 service client.
    svc := ec2.New(sess)
```

Create a new key pair with the provided name.

```
    result, err := svc.CreateKeyPair(&ec2.CreateKeyPairInput{
        KeyName: aws.String(pairName),
    })
    if err != nil {
        if aerr, ok := err.(awserr.Error); ok && aerr.Code() ==
 "InvalidKeyPair.Duplicate" {
            exitErrorf("Keypair %q already exists.", pairName)
        }
        exitErrorf("Unable to create key pair: %s, %v.", pairName, err)
    }

    fmt.Printf("Created key pair %q %s\n%s\n",
        *result.KeyName, *result.KeyFingerprint,
        *result.KeyMaterial)
}
```

The routine uses the following utility function.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Deleting a Key Pair

Create a new Go file named `ec2_delete_keypair.go`.

Import the required AWS SDK for Go packages.

```
package main

import (
    "fmt"
    "os"
    "path/filepath"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/ec2"
```

```
)
```

Get the key pair name passed in to the code and, to access Amazon EC2, create an EC2 client.

```
func main() {
    if len(os.Args) != 2 {
        exitErrorf("pair name required\nUsage: %s key_pair_name",
            filepath.Base(os.Args[0]))
    }
    pairName := os.Args[1]

    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create an EC2 service client.
    svc := ec2.New(sess)
```

Delete the key pair with the provided name.

```
    _, err := svc.DeleteKeyPair(&ec2.DeleteKeyPairInput{
        KeyName: aws.String(pairName),
    })
    if err != nil {
        if aerr, ok := err.(awserr.Error); ok && aerr.Code() ==
 "InvalidKeyPair.Duplicate" {
            exitErrorf("Key pair %q does not exist.", pairName)
        }
        exitErrorf("Unable to delete key pair: %s, %v.", pairName, err)
    }

    fmt.Printf("Successfully deleted %q key pair\n", pairName)
}
```

The routine uses the following utility function.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Using Regions and Availability Zones with Amazon EC2 with Go

These Go examples show you how to:

- Retrieve details about AWS Regions and Availability Zones.

An Amazon EC2 security group acts as a virtual firewall that controls the traffic for one or more instances. You add rules to each security group to allow traffic to or from its associated instances. You can modify the rules for a security group at any time; the new rules are automatically applied to all instances that are associated with the security group.

The code in this example uses the AWS SDK for Go to perform these tasks by using these methods of the Amazon EC2 client class:

- DescribeSecurityGroups

- AuthorizeSecurityGroupIngress
- CreateSecurityGroup
- DescribeVpcs
- DeleteSecurityGroup

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

Amazon EC2 is hosted in multiple locations worldwide. These locations are composed of AWS Regions and Availability Zones. Each region is a separate geographic area with multiple, isolated locations known as Availability Zones. Amazon EC2 provides the ability to place instances and data in these multiple locations.

In this example, you use Go code to retrieve details about regions and Availability Zones. The code uses the AWS SDK for Go tomanage instances by using the following methods of the Amazon EC2 client class:

- DescribeAvailabilityZones
- DescribeRegions

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with AWS Regions and Availability Zones. To learn more, see Regions and Availability Zones in the *Amazon EC2 User Guide for Linux Instances* or Regions and Availability Zones in the *Amazon EC2 User Guide for Windows Instances*.

Listing the Groups

This example describes the security groups by IDs that are passed in to the routine. It takes a space separated list of group IDs as input.

To get started, create a new Go file named `regions_and_availability.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/ec2"
)
```

In the `main` function, create a session from the shared config file and a new EC2 client.

```
func main() {
    // Load session from shared config
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))
```

```
    // Create new EC2 client
    svc := ec2.New(sess)
```

Print out the list of regions that work with Amazon EC2 that are returned by calling `DescribeRegions`.

```
    resultRegions, err := svc.DescribeRegions(nil)
    if err != nil {
        fmt.Println("Error", err)
        return
    }
```

Add a call that retrieves Availability Zones only for the region of the EC2 service object.

```
    resultAvalZones, err := svc.DescribeAvailabilityZones(nil)
    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success", resultAvalZones.AvailabilityZones)
}
```

Working with Security Groups in Amazon EC2 with Go

These Go examples show you how to:

- Retrieve information about your security groups
- Create a security group to access an Amazon EC2 instance
- Delete an existing security group

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

An Amazon EC2 security group acts as a virtual firewall that controls the traffic for one or more instances. You add rules to each security group to allow traffic to or from its associated instances. You can modify the rules for a security group at any time; the new rules are automatically applied to all instances that are associated with the security group.

The code in this example uses the AWS SDK for Go to perform these tasks by using these methods of the Amazon EC2 client class:

- DescribeSecurityGroups
- AuthorizeSecurityGroupIngress
- CreateSecurityGroup
- DescribeVpcs
- DeleteSecurityGroup

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.

- You are familiar with Amazon EC2 security groups. To learn more, see Amazon EC2 Amazon Security Groups for Linux Instances in the *Amazon EC2 User Guide for Linux Instances* or Amazon EC2 Amazon Security Groups for Windows Instances in the *Amazon EC2 User Guide for Windows Instances*.

Describing Your Security Groups

This example describes the security groups by IDs that are passed into the routine. It takes a space separated list of group IDs as input.

To get started, create a new Go file named `ec2_describe_security_groups.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/cloudwatch"
)
```

In the `main` function, get the security group ID that is passed in.

```
func main() {
    if len(os.Args) < 2 {
        exitErrorf("Security Group ID required\nUsage: %s group_id ...",
            filepath.Base(os.Args[0]))
    }
    groupIds := os.Args[1:]
```

Initialize a session and create an EC2 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create an EC2 service client.
    svc := ec2.New(sess)
```

Obtain and print out the security group descriptions. You will explicity check for errors caused by an invalid group ID.

```
    result, err := svc.DescribeSecurityGroups(&ec2.DescribeSecurityGroupsInput{
        GroupIds: aws.StringSlice(groupIds),
    })
    if err != nil {
        if aerr, ok := err.(awserr.Error); ok {
            switch aerr.Code() {
            case "InvalidGroupId.Malformed":
                fallthrough
            case "InvalidGroup.NotFound":
                exitErrorf("%s.", aerr.Message())
            }
        }
        exitErrorf("Unable to get descriptions for security groups, %v", err)
```

```
    }

    fmt.Println("Security Group:")
    for _, group := range result.SecurityGroups {
        fmt.Println(group)
    }
}
```

The following utility function is used by this example.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Creating a Security Group

You can create new Amazon EC2 security groups. To do this, you use the CreateSecurityGroup method.

This example creates a new security group with the given name and description for access to open ports 80 and 22. If a VPC ID is not provided, it associates the security group with the first VPC in the account.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "flag"
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/ec2"
)
```

Get the parameters (name, description, and optional ID of the VPC) that are passed in to the routine.

```
func main() {
    var name, desc, vpcID string
    flag.StringVar(&name, "n", "", "Group Name")
    flag.StringVar(&desc, "d", "", "Group Description")
    flag.StringVar(&vpcID, "vpc", "", "(Optional) VPC ID to associate security group with")
    flag.Parse()

    if len(name) == 0 || len(desc) == 0 {
        flag.PrintDefaults()
        exitErrorf("Group name and description require")
    }
```

Create a session.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create an EC2 service client.
    svc := ec2.New(sess)
```

If the VPC ID was not provided, you have to retrieve the first one in the account.

```
if len(vpcID) == 0 {
    // Get a list of VPCs so we can associate the group with the first VPC.
    result, err := svc.DescribeVpcs(nil)
    if err != nil {
        exitErrorf("Unable to describe VPCs, %v", err)
    }
    if len(result.Vpcs) == 0 {
        exitErrorf("No VPCs found to associate security group with.")
    }
    vpcID = aws.StringValue(result.Vpcs[0].VpcId)
}
```

Then create the security group with the VPC ID, name, and description.

```
createRes, err := svc.CreateSecurityGroup(&ec2.CreateSecurityGroupInput{
    GroupName:   aws.String(name),
    Description: aws.String(desc),
    VpcId:       aws.String(vpcID),
})
if err != nil {
    if aerr, ok := err.(awserr.Error); ok {
        switch aerr.Code() {
        case "InvalidVpcID.NotFound":
            exitErrorf("Unable to find VPC with ID %q.", vpcID)
        case "InvalidGroup.Duplicate":
            exitErrorf("Security group %q already exists.", name)
        }
    }
    exitErrorf("Unable to create security group %q, %v", name, err)
}
fmt.Printf("Created security group %s with VPC %s.\n",
    aws.StringValue(createRes.GroupId), vpcID)
```

Add permissions to the security group.

```
_, err = svc.AuthorizeSecurityGroupIngress(&ec2.AuthorizeSecurityGroupIngressInput{
    GroupName: aws.String(name),
    IpPermissions: []*ec2.IpPermission{
        (&ec2.IpPermission{}).
            SetIpProtocol("tcp").
            SetFromPort(80).
            SetToPort(80).
            SetIpRanges([]*ec2.IpRange{
                {CidrIp: aws.String("0.0.0.0/0")},
            }),
        (&ec2.IpPermission{}).
            SetIpProtocol("tcp").
            SetFromPort(22).
            SetToPort(22).
            SetIpRanges([]*ec2.IpRange{
                (&ec2.IpRange{}).
                    SetCidrIp("0.0.0.0/0"),
            }),
    },
})
if err != nil {
    exitErrorf("Unable to set security group %q ingress, %v", name, err)
}

fmt.Println("Successfully set security group ingress")
```

```
}
```

The following utility function is used by this example.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Deleting a Security Group

You can delete an Amazon EC2 security group in code. To do this, you use the DeleteSecurityGroup method.

This example deletes a security group with the given group ID.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"
    "path/filepath"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/ec2"
)
```

Get the group ID that is passed in to the routine.

```
func main() {
    if len(os.Args) != 2 {
        exitErrorf("Security Group ID required\nUsage: %s group_id",
            filepath.Base(os.Args[0]))
    }
    groupID := os.Args[1]
```

Create a session.

```
    svc := ec2.New(sess)
```

Then delete the security group with the group ID that is passed in.

```
    _, err := svc.DeleteSecurityGroup(&ec2.DeleteSecurityGroupInput{
        GroupId: aws.String(groupID),
    })
    if err != nil {
        if aerr, ok := err.(awserr.Error); ok {
            switch aerr.Code() {
            case "InvalidGroupId.Malformed":
                fallthrough
            case "InvalidGroup.NotFound":
                exitErrorf("%s.", aerr.Message())
            }
        }
```

```
        exitErrorf("Unable to get descriptions for security groups, %v.", err)
    }

    fmt.Printf("Successfully delete security group %q.\n", groupID)
}
```

This example uses the following utility function.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Using Elastic IP Addresses in Amazon EC2 with Go

These Go examples show you how to:

- Describe Amazon EC2 instance IP addresses
- Allocate addresses to Amazon EC2 instances
- Release Amazon EC2 instance IP addresses

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

An Elastic IP address is a static IP address designed for dynamic cloud computing that is associated with your AWS account. It is a public IP address, reachable from the Internet. If your instance doesn't have a public IP address, you can associate an Elastic IP address with the instance to enable communication with the Internet.

In this example, you use a series of Go routines to perform several Amazon EC2 operations involving Elastic IP addresses. The routines use the AWS SDK for Go to manage Elastic IP addresses by using these methods of the Amazon EC2 client class:

- DescribeAddresses
- AllocateAddress
- AssociateAddress
- ReleaseAddress

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with Elastic IP addresses in Amazon EC2. To learn more, see Elastic IP Addresses in the *Amazon EC2 User Guide for Linux Instances* or Elastic IP Addresses in the *Amazon EC2 User Guide for Windows Instances*.

Describing Instance IP Addresses

Create a new Go file named `ec2_describe_addresses.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/ec2"
)
```

Getting the Address Descriptions

This routine prints out the Elastic IP Addresses for the account's VPC. Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new EC2 service client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create an EC2 service client.
    svc := ec2.New(sess)
```

Make the API request to EC2 filtering for the addresses in the account's VPC.

```
    result, err := svc.DescribeAddresses(&ec2.DescribeAddressesInput{
        Filters: []*ec2.Filter{
            {
                Name:   aws.String("domain"),
                Values: aws.StringSlice([]string{"vpc"}),
            },
        },
    })
    if err != nil {
        exitErrorf("Unable to elastic IP address, %v", err)
    }

    // Printout the IP addresses if there are any.
    if len(result.Addresses) == 0 {
        fmt.Printf("No elastic IPs for %s region\n", *svc.Config.Region)
    } else {
        fmt.Println("Elastic IPs")
        for _, addr := range result.Addresses {
            fmt.Println("*", fmtAddress(addr))
        }
    }
}
```

The `fmtAddress` and `exitErrorf` functions are utility functions used in the example.

```
func fmtAddress(addr *ec2.Address) string {
    out := fmt.Sprintf("IP: %s,  allocation id: %s",
        aws.StringValue(addr.PublicIp), aws.StringValue(addr.AllocationId))
    if addr.InstanceId != nil {
        out += fmt.Sprintf(", instance-id: %s", *addr.InstanceId)
    }
    return out
}
```

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Allocating Addresses to Instances

Create a new Go file named `ec2_allocate_address.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"
    "path/filepath"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/ec2"
)
```

This routine attempts to allocate a VPC Elastic IP address for the current region. The IP address requires and will be associated with the instance ID that is passed in.

```
func main() {
    if len(os.Args) != 2 {
        exitErrorf("instance ID required\nUsage: %s instance_id",
            filepath.Base(os.Args[0]))
    }
    instanceID := os.Args[1]
```

You will need to initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new Amazon S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create an EC2 service client.
    svc := ec2.New(sess)
```

Call AllocateAddress, passing in "vpc" as the Domain value.

```
    allocRes, err := svc.AllocateAddress(&ec2.AllocateAddressInput{
        Domain: aws.String("vpc"),
    })
    if err != nil {
        exitErrorf("Unable to allocate IP address, %v", err)
    }
```

Call AssociateAddress to associate the new Elastic IP address with an existing Amazon EC2 instance, and print out the results.

```
    assocRes, err := svc.AssociateAddress(&ec2.AssociateAddressInput{
        AllocationId: allocRes.AllocationId,
        InstanceId:   aws.String(instanceID),
```

```
    })
    if err != nil {
        exitErrorf("Unable to associate IP address with %s, %v",
            instanceID, err)
    }

    fmt.Printf("Successfully allocated %s with instance %s.\n\tallocation id: %s,
 association id: %s\n",
        *allocRes.PublicIp, instanceID, *allocRes.AllocationId, *assocRes.AssociationId)
}
```

This example also uses the `exitErrorf` utility function.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Releasing Instance IP Addresses

This routine releases an Elastic IP address allocation ID. If the address is associated with an Amazon EC2 instance, the association is removed.

Create a new Go file named `ec2_release_address.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"
    "path/filepath"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/ec2"
)
```

The routine requires that the user pass in the allocation ID of the Elastic IP address.

```
func main() {
    if len(os.Args) != 2 {
        exitErrorf("allocation ID required\nUsage: %s allocation_id",
            filepath.Base(os.Args[0]))
    }
    allocationID := os.Args[1]
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new EC2 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create an EC2 service client.
    svc := ec2.New(sess)
```

Attempt to release the Elastic IP address by using the allocation ID.

```
    _, err := svc.ReleaseAddress(&ec2.ReleaseAddressInput{
        AllocationId: aws.String(allocationID),
    })
    if err != nil {
        if aerr, ok := err.(awserr.Error); ok && aerr.Code() ==
 "InvalidAllocationID.NotFound" {
            exitErrorf("Allocation ID %s does not exist", allocaitonID)
        }
        exitErrorf("Unable to release IP address for allocation %s, %v",
            allocationID, err)
    }

    fmt.Printf("Successfully released allocation ID %s\n", allocationID)
}
```

This example uses the `fmtAddress` and `exitErrorf` utility functions.

```
func fmtAddress(addr *ec2.Address) string {
    out := fmt.Sprintf("IP: %s,  allocation id: %s",
        aws.StringValue(addr.PublicIp), aws.StringValue(addr.AllocationId))
    if addr.InstanceId != nil {
        out += fmt.Sprintf(", instance-id: %s", *addr.InstanceId)
    }
    return out
}

func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Amazon Glacier with Go Examples

The AWS SDK for Go examples can integrate Amazon Glacier into your applications. The examples assume you have already set up and configured the SDK (that is, you've imported all required packages and set your credentials and region). For more information, see Setting Up (p. 2) and SDK Configuration (p. 4).

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

Amazon Glacier is a secure cloud storage service for data archiving and long-term backup. The service is optimized for infrequently accessed data where a retrieval time of several hours is suitable. These examples show you how to create a vault and upload an archive with Go. The methods used include:

- CreateVault
- UploadArchive

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.

- You are familiar with the Amazon Glacier data model. To learn more, see Amazon Glacier Data Model in the *Amazon Glacier Developer Guide*.

Creating a Vault

The following example uses the Amazon GlacierCreateVault operation to create a vault named `YOUR_VAULT_NAME`.

```
svc := glacier.New(session.New(&aws.Config{Region: aws.String("us-west-2")}))
_, err := svc.CreateVault(&glacier.CreateVaultInput{
    VaultName: aws.String("YOUR_VAULT_NAME"),
})
if err != nil {
    log.Println(err)
    return
}
log.Println("Created vault!")
```

Uploading an Archive

The following example assumes you have a vault named `YOUR_VAULT_NAME`. It uses the Amazon GlacierUploadArchive operation to upload a single reader object as an entire archive. The AWS SDK for Go automatically computes the tree hash checksum for the data to be uploaded.

```
vaultName := "YOUR_VAULT_NAME"

svc := glacier.New(session.New(&aws.Config{Region: aws.String("us-west-2")}))
result, err := svc.UploadArchive(&glacier.UploadArchiveInput{
    AccountId: aws.String("-"),
    VaultName: &vaultName,
    Body:      bytes.NewReader(make([]byte, 2*1024*1024)), // 2 MB buffer
})
if err != nil {
    log.Println("Error uploading archive.", err)
    return
}

log.Println("Uploaded to archive", *result.ArchiveId)
```

IAM with Go Examples

AWS Identity and Access Management (IAM) is a web service that enables AWS customers to manage users and user permissions in AWS. The service is targeted at organizations with multiple users or systems in the cloud that use AWS products. With IAM, you can centrally manage users, security credentials such as access keys, and permissions that control which AWS resources users can access.

The examples assume you have already set up and configured the SDK (that is, you've imported all required packages and set your credentials and region). For more information, see Setting Up (p. 2) and SDK Configuration (p. 4).

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

Topics

Managing IAM Users with Go

This Go example shows you how to create, update, view, and delete IAM users. You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

In this example, you use a series of Go routines to manage users in IAM. The routines use the AWS SDK for Go IAM client methods that follow:

- CreateUser
- ListUsers
- UpdateUser
- GetUser
- DeleteUser

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with IAM users. To learn more, see IAM Users in the *IAM User Guide*.

Creating a New IAM User

This code creates a new IAM user.

Create a new Go file named `iam_createuser.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

The code takes the new user name as an argument, and then calls `GetUser` with the user name.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))
```

```
    // Create a IAM service client.
    svc := iam.New(sess)

    _, err := svc.GetUser(&iam.GetUserInput{
        UserName: &os.Args[1],
    })
```

If you receive a `NoSuchEntity` error, call `CreateUser` because the user doesn't exist.

```
    if awserr, ok := err.(awserr.Error); ok && awserr.Code() ==
 iam.ErrCodeNoSuchEntityException {
        result, err := svc.CreateUser(&iam.CreateUserInput{
            UserName: &os.Args[1],
        })

        if err != nil {
            fmt.Println("CreateUser Error", err)
            return
        }

        fmt.Println("Success", result)
    } else {
        fmt.Println("GetUser Error", err)
    }
}
```

Listing IAM Users in Your Account

You can get a list of the users in your account and print the list to the console.

Create a new Go file named `iam_listusers.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a new IAM client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

Call `ListUsers` and print the results.

```
    result, err := svc.ListUsers(&iam.ListUsersInput{
        MaxItems: aws.Int64(10),
    })
```

```
    if err != nil {
        fmt.Println("Error", err)
        return
    }

    for i, user := range result.Users {
        if user == nil {
            continue
        }
        fmt.Printf("%d user %s created %v\n", i, *user.UserName, user.CreateDate)
    }
}
```

Updating a User's Name

In this example, you change the name of an IAM user to a new value.

Create a new Go file named `iam_updateuser.go`. You must import the relevant Go and|sdk-go| packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a new IAM client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

Call `UpdateUser`, passing in the original user name and the new name, and print the results.

```
    result, err := svc.UpdateUser(&iam.UpdateUserInput{
        UserName:    &os.Args[1],
        NewUserName: &os.Args[2],
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success", result)
}
```

Deleting an IAM User

In this example, you delete an IAM user.

Create a new Go file named `iam_updateuser.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a new IAM client.

```
    // Initialize a session that the SDK will use to load configuration,
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)

    result, err := svc.UpdateUser(&iam.UpdateUserInput{
        UserName:    &os.Args[1],
        NewUserName: &os.Args[2],
    })
```

Call `UpdateUser`, passing in the user name, and print the results. If the user doesn't exist, log an error.

```
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success", result)
}
```

Managing IAM Access Keys with Go

This Go example shows you how to create, modify, view, or rotate IAM access keys. You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

Users need their own access keys to make programmatic calls to the AWS SDK for Go. To fill this need, you can create, modify, view, or rotate access keys (access key IDs and secret access keys) for IAM users. By default, when you create an access key its status is Active, which means the user can use the access key for API calls.

In this example, you use a series of Go routines to manage access keys in IAM. The routines use the AWS SDK for Go IAM client methods that follow:

- CreateAccessKey
- ListAccessKeys
- GetAccessKeyLastUsed
- UpdateAccessKey

- DeleteAccessKey

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with IAM access keys. To learn more, see Managing Access Keys for IAM Users in the *IAM User Guide*.

Creating a New IAM Access Key

This code creates a new IAM access key for the IAM user named IAM_USER_NAME.

Create a new Go file named `iam_createaccesskey.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up the session.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

Call `CreateAccessKey` and print the results.

```
    result, err := svc.CreateAccessKey(&iam.CreateAccessKeyInput{
        UserName: aws.String("IAM_USER_NAME"),
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success", *result.AccessKey)
}
```

Listing a User's Access Keys

In this example, you get a list of the access keys for a user and print the list to the console.

Create a new Go file named `iam_listaccesskeys.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a new IAM client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

Call `ListAccessKeys` and print the results.

```
    result, err := svc.ListAccessKeys(&iam.ListAccessKeysInput{
        MaxItems: aws.Int64(5),
        UserName: aws.String("IAM_USER_NAME"),
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success", result)
}
```

Getting the Last Use for an Access Key

In this example, you find out when an access key was last used.

Create a new Go file named `iam_accesskeylastused.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a new IAM client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))
```

```
    // Create a IAM service client.
    svc := iam.New(sess)
```

Call `GetAccessKeyLastUsed`, passing in the access key ID, and print the results.

```
    result, err := svc.GetAccessKeyLastUsed(&iam.GetAccessKeyLastUsedInput{
        AccessKeyId: aws.String("ACCESS_KEY_ID"),
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success", *result.AccessKeyLastUsed)
}
```

Updating Access Key Status

In this example, you delete an IAM user.

Create a new Go file with the name `iam_updateaccesskey.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a new IAM client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

Call `UpdateAccessKey`, passing in the access key ID, status (making it active in this case), and user name.

```
    _, err := svc.UpdateAccessKey(&iam.UpdateAccessKeyInput{
        AccessKeyId: aws.String("ACCESS_KEY_ID"),
        Status:      aws.String(iam.StatusTypeActive),
        UserName:    aws.String("USER_NAME"),
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Access Key updated")
```

```
}
```

Deleting an Access Key

In this example, you delete an access key.

Create a new Go file named `iam_deleteaccesskey.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a new IAM client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

Call `DeleteAccessKey`, passing in the access key ID and user name.

```
    result, err := svc.DeleteAccessKey(&iam.DeleteAccessKeyInput{
        AccessKeyId: aws.String("ACCESS_KEY_ID"),
        UserName:    aws.String("USER_NAME"),
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success", result)
}
```

Managing IAM Account Aliases with Go

This Go example shows you how to create, list, and delete IAM account aliases. You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

You can use a series of Go routines to manage aliases in IAM. The routines use the AWS SDK for Go IAM client methods that follow:

- CreateAccountAlias
- ListAccountAliases

- DeleteAccountAlias

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with IAM account aliases. To learn more, see Your AWS Account ID and Its Alias in the *IAM User Guide*.

Creating a New IAM Account Alias

This code creates a new IAM user.

Create a new Go file named `iam_createaccountalias.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a session and an IAM client.

```
    // Initialize a session that the SDK will use to load configuration,
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)

    _, err := svc.CreateAccountAlias(&iam.CreateAccountAliasInput{
        AccountAlias: &os.Args[1],
    })
```

The code takes the new alias as an argument, and then calls `CreateAccountAlias` with the alias name.

```
    _, err := svc.CreateAccountAlias(&iam.CreateAccountAliasInput{
        AccountAlias: &os.Args[1],
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Printf("Account alias %s has been created\n", os.Args[1])
}
```

Listing IAM Account Aliases

This code lists the aliases for your IAM account.

Create a new Go file named `iam_listaccountaliases.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a session and an IAM client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

The code calls `ListAccountAliases`, specifying to return a maximum of 10 items.

```
    result, err := svc.ListAccountAliases(&iam.ListAccountAliasesInput{
        MaxItems: aws.Int64(10),
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    for i, alias := range result.AccountAliases {
        if alias == nil {
            continue
        }
        fmt.Printf("Alias %d: %s\n", i, *alias)
    }
}
```

Deleting an IAM Account Alias

This code deletes a specified IAM account alias.

Create a new Go file with the name `iam_deleteaccountalias.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a session and an IAM client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

The code calls `ListAccountAliases`, specifying to return a maximum of 10 items.

```
    _, err := svc.DeleteAccountAlias(&iam.DeleteAccountAliasInput{
        AccountAlias: &os.Args[1],
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Printf("Alias %s has been deleted\n", os.Args[1])
}
```

Working with IAM Policies with Go

This Go example shows you how to create, get, attach, and detach IAM policies. You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

You grant permissions to a user by creating a policy, which is a document that lists the actions that a user can perform and the resources those actions can affect. Any actions or resources that are not explicitly allowed are denied by default. Policies can be created and attached to users, groups of users, roles assumed by users, and resources.

In this example, you use a series of Go routines to manage policies in IAM. The routines use the AWS SDK for Go IAM client methods that follow:

- CreatePolicy
- GetPolicy
- ListAttachedRolePolicies
- AttachRolePolicy
- DetachRolePolicy

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with IAM policies. To learn more, see Overview of Access Management: Permissions and Policies in the *IAM User Guide*.

Creating an IAM Policy

This code creates a new IAM Policy. Create a new Go file named `iam_createpolicy.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "encoding/json"
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Define two structs. The first is the definition of the policies to upload to IAM.

```
type PolicyDocument struct {
    Version   string
    Statement []StatementEntry
}
```

The second dictates what this policy will allow or disallow.

```
type StatementEntry struct {
    Effect   string
    Action   []string
    Resource string
}
```

Set up the session and IAM client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

Build the policy document using the structures defined earlier.

```
    policy := PolicyDocument{
        Version: "2012-10-17",
        Statement: []StatementEntry{
            StatementEntry{
                Effect: "Allow",
                Action: []string{
                    "logs:CreateLogGroup", // Allow for creating log groups
                },
                Resource: "RESOURCE ARN FOR logs:*",
            },
            StatementEntry{
                Effect: "Allow",
                // Allows for DeleteItem, GetItem, PutItem, Scan, and UpdateItem
                Action: []string{
                    "dynamodb:DeleteItem",
                    "dynamodb:GetItem",
                    "dynamodb:PutItem",
                    "dynamodb:Scan",
                    "dynamodb:UpdateItem",
                },
                Resource: "RESOURCE ARN FOR dynamodb:*",
            },
```

```
        },
    }
```

Marshal the policy to JSON and pass to `CreatePolicy`.

```
    b, err := json.Marshal(&policy)
    if err != nil {
        fmt.Println("Error marshaling policy", err)
        return
    }

    result, err := svc.CreatePolicy(&iam.CreatePolicyInput{
        PolicyDocument: aws.String(string(b)),
        PolicyName:     aws.String("myDynamodbPolicy"),
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("New policy", result)
}
```

Getting an IAM Policy

In this example, you retrieve an existing policy from IAM. Create a new Go file named `iam_getpolicy.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a new IAM client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

Call `GetPolicy`, passing in the ARN for the policy (which is hard coded in this example), and print the results.

```
    svc := iam.New(sess)

    arn := "arn:aws:iam::aws:policy/AWSLambdaExecute"
    result, err := svc.GetPolicy(&iam.GetPolicyInput{
        PolicyArn: &arn,
    })
```

```
    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Printf("%s - %s\n", arn, *result.Policy.Description)
}
```

Attaching a Managed Role Policy

In this example, you attach an IAM managed role policy. Create a new Go file named `iam_attachuserpolicy.go`. You'll call the `ListAttachedRolePolicies` method of the IAM service object, which returns an array of managed policies.

Then, you'll check the array members to see if the policy you want to attach to the role is already attached. If the policy isn't attached, you'll call the `AttachRolePolicy` method to attach it.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a new IAM client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

Declare variables to hold the name and ARN of the policy.

```
    var pageErr error
    policyName := "AmazonDynamoDBFullAccess"
    policyArn := "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess"
```

Paginate through all the role policies. If your role exists on any role policy, you set the `pageErr` and return `false`, stopping the pagination.

```
    err := svc.ListAttachedRolePoliciesPages(
        &iam.ListAttachedRolePoliciesInput{
            RoleName: &os.Args[1],
        },
        func(page *iam.ListAttachedRolePoliciesOutput, lastPage bool) bool {
            if page != nil && len(page.AttachedPolicies) > 0 {
                for _, policy := range page.AttachedPolicies {
                    if *policy.PolicyName == policyName {
                        pageErr = fmt.Errorf("%s is already attached to this role",
 policyName)
```

```
                        return false
                    }
                }
                // We should keep paginating because we did not find our role
                return true
            }
            return false
        },
    )
```

If your role policy is not attached already, call `AttachRolePolicy`.

```
    if pageErr != nil {
        fmt.Println("Error", pageErr)
        return
    }

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    _, err = svc.AttachRolePolicy(&iam.AttachRolePolicyInput{
        PolicyArn: &policyArn,
        RoleName:  &os.Args[1],
    })

    if err != nil {
        fmt.Println("Unable to attach role policy to role")
        return
    }
    fmt.Println("Role attached successfully")
}
```

Detaching a Managed Role Policy

In this example, you detach a role policy. Once again, you call the `ListAttachedRolePolicies` method of the IAM service object, which returns an array of managed policies.

Then, check the array members to see if the policy you want to detach from the role is attached. If the policy is attached, call the `DetachRolePolicy` method to detach it.

Create a new Go file named `iam_detachuserpolicy.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a new IAM client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
```

```
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

Declare variables to hold the name and ARN of the policy.

```
    foundPolicy := false
    policyName := "AmazonDynamoDBFullAccess"
    policyArn := "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess"
```

Paginate through all the role policies. If the role exists on any role policy, you stop iterating and detach the role.

```
    err := svc.ListAttachedRolePoliciesPages(
        &iam.ListAttachedRolePoliciesInput{
            RoleName: &os.Args[1],
        },
        func(page *iam.ListAttachedRolePoliciesOutput, lastPage bool) bool {
            if page != nil && len(page.AttachedPolicies) > 0 {
                for _, policy := range page.AttachedPolicies {
                    if *policy.PolicyName == policyName {
                        foundPolicy = true
                        return false
                    }
                }
                return true
            }
            return false
        },
    )

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    if !foundPolicy {
        fmt.Println("Policy was not attached to role")
        return
    }

    _, err = svc.DetachRolePolicy(&iam.DetachRolePolicyInput{
        PolicyArn: &policyArn,
        RoleName:  &os.Args[1],
    })

    if err != nil {
        fmt.Println("Unable to detach role policy to role")
        return
    }
    fmt.Println("Role detached successfully")
}
```

Working with IAM Server Certificates with Go

This Go example shows you how to carry out basic tasks for managing server certificate HTTPS connections with the AWS SDK for Go.

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

To enable HTTPS connections to your website or application on AWS, you need an SSL/TLS server certificate. To use a certificate that you obtained from an external provider with your website or application on AWS, you must upload the certificate to IAM or import it into AWS Certificate Manager.

In this example, you use a series of Go routines to manage policies in IAM. The routines use the AWS SDK for GoIAM client methods that follow:

- ListServerCertificates
- GetServerCertificate
- UpdateServerCertificate
- DeleteServerCertificate

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with server certificates. To learn more, see Working with Server Certificates in the *IAM User Guide*.

Listing Your Server Certificates

This code lists your certificates.

Create a new Go file named `iam_listservercerts.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

Call `ListServerCertificates` and print the details.

```
    result, err := svc.ListServerCertificates(nil)
    if err != nil {
        fmt.Println("Error", err)
        return
    }

    for i, metadata := range result.ServerCertificateMetadataList {
        if metadata == nil {
            continue
        }

        fmt.Printf("Metadata %d: %v\n", i, metadata)
    }
}
```

Getting a Server Certificate

In this example, you retrieve an existing server certificate.

Create a new Go file named `iam_getservercert.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a new IAM client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

Call `GetServerCertificate`, passing the name of the certificate, and print the results.

```
    result, err := svc.GetServerCertificate(&iam.GetServerCertificateInput{
        ServerCertificateName: aws.String("CERTIFICATE_NAME"),
    })
    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("ServerCertificate:", result)
}
```

Updating a Server Certificate

In this example, you update an existing server certificate.

Create a new Go file named `iam_updateservercert.go`. You call the `UpdateServerCertificate` method of the IAM service object to change the name of the certificate.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a new IAM client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
```

```
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

Update the certificate name.

```
    _, err := svc.UpdateServerCertificate(&iam.UpdateServerCertificateInput{
        ServerCertificateName:    aws.String("CERTIFICATE_NAME"),
        NewServerCertificateName: aws.String("NEW_CERTIFICATE_NAME"),
    })
    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Server certificate updated")
}
```

Deleting a Server Certificate

In this example, you delete an existing server certificate.

Create a new Go file named `iam_deleteservercert.go`. You call the `DeleteServerCertificate` method of the IAM service object to change the name of the certificate.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/iam"
)
```

Set up a new IAM client.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a IAM service client.
    svc := iam.New(sess)
```

Call the method to delete the certificate, specifying the name of certificate.

```
    _, err := svc.DeleteServerCertificate(&iam.DeleteServerCertificateInput{
        ServerCertificateName: aws.String("CERTIFICATE_NAME"),
    })
    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Server certificate deleted")
```

```
}
```

Amazon S3 with Go Examples

The AWS SDK for Go examples can integrate Amazon S3 into your applications. The examples assume you have already set up and configured the SDK (that is, you've imported all required packages and set your credentials and region). For more information, see Setting Up (p. 2) and SDK Configuration (p. 4).

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

Topics

Basic Amazon S3 Bucket Operations in Go

These Go examples show you how to perform the following operations on Amazon S3 buckets and bucket items:

- List the buckets in your account
- Create a bucket
- List the items in a bucket
- Upload a file to a bucket
- Download a bucket item
- Copy a bucket item to another bucket
- Delete a bucket item
- Delete all the items in a bucket
- Restore a bucket item
- Delete a bucket

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

In these examples, a series of Go routines are used to perform operations on your Amazon S3 buckets. The routines use the AWS SDK for Go to perform Amazon S3 bucket operations using the following methods of the Amazon S3 client class, unless otherwise noted:

- ListBuckets
- CreateBucket
- ListObjects
- Upload (from the **s3manager.NewUploader** class)
- Download (from the **s3manager.NewDownloader** class)

- CopyObject
- DeleteObject
- DeleteObjects
- RestoreObject
- DeleteBucket

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with buckets. To learn more, see Working with Amazon S3 Buckets in the *Amazon S3 Developer Guide*.

Listing Buckets

The ListBuckets function lists the buckets in your account.

The following example lists the buckets in your account. There are no command-line arguments.

Create the file *s3_list_buckets.go*. Add the following statements to import the Go and AWS SDK for Go packages used in the example.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

Create a function we use to display errors and exit.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Initialize the session that the SDK uses to load configuration, credential, and region information from the shared config file *~/.aws/config*, and create a new Amazon S3 service client.

```
func main() {
    // Initialize a session that the SDK will use to load configuration,
    // credentials, and region from the shared config file. (~/.aws/config).
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Call ListBuckets. Passing `nil` means no filters are applied to the returned list. If an error occurs, call `exitErrorf`. If no error occurs, loop through the buckets, printing the name and creation date of each bucket.

```
    result, err := svc.ListBuckets(nil)

    if err != nil {
        exitErrorf("Unable to list buckets, %v", err)
    }

    fmt.Println("Buckets:")

    for _, b := range result.Buckets {
        fmt.Printf("* %s created on %s\n",
            aws.StringValue(b.Name), aws.TimeValue(b.CreationDate))
    }
```

See the complete example on GitHub.

Creating a Bucket

The CreateBucket function creates a bucket in your account.

The following example creates a bucket with the name specified as a command-line argument. You must specify a globally unique name for the bucket.

Create the file *s3_create_bucket.go*. Import the following Go and AWS SDK for Go packages.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

Create a function we use to display errors and exit.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

The program requires one argument, the name of the bucket to create.

```
func main() {
    if len(os.Args) != 2 {
        exitErrorf("Bucket name missing!\nUsage: %s bucket_name", os.Args[0])
    }

    bucket := os.Args[1]
```

Initialize the session that the SDK uses to load configuration, credentials, and region information from the shared config file *~/.aws/config*, and create a new S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
```

```
    svc := s3.New(sess)
```

Call CreateBucket, passing in the bucket name defined previously. If an error occurs, call `exitErrorf`. If there are no errors, wait for a notification that the bucket was created.

```
    _, err := svc.CreateBucket(&s3.CreateBucketInput{
        Bucket: aws.String(bucket),
    })

    if err != nil {
        exitErrorf("Unable to create bucket %q, %v", bucket, err)
    }

    // Wait until bucket is created before finishing
    fmt.Printf("Waiting for bucket %q to be created...\n", bucket)

    err = svc.WaitUntilBucketExists(&s3.HeadBucketInput{
        Bucket: aws.String(bucket),
    })
```

If the `WaitUntilBucketExists` call returns an error, call `exitErrorf`. If there are no errors, notify the user of success.

```
    if err != nil {
        exitErrorf("Error occurred while waiting for bucket to be created, %v", bucket)
    }

    fmt.Printf("Bucket %q successfully created\n", bucket)
```

See the complete example on GitHub.

Listing Bucket Items

The ListObjects function lists the items in a bucket.

The following example lists the items in the bucket with the name specified as a command-line argument.

Create the file *s3_list_objects.go*. Add the following statements to import the Go and AWS SDK for Go packages used in the example.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

Create a function we use to display errors and exit.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

The program requires one command-line argument, the name of the bucket.

```
func main() {
    if len(os.Args) != 2 {
        exitErrorf("Bucket name required\nUsage: %s bucket_name",
            os.Args[0])
    }

    bucket := os.Args[1]
```

Initialize the session that the SDK uses to load configuration, credential, and region information from the shared config file *~/.aws/config*, and create a new Amazon S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Call ListObjects, passing in the name of the bucket. If an error occurs, call `exitErrorf`. If no error occurs, loop through the items, printing the name, last modified date, size, and storage class of each item.

```
    resp, err := svc.ListObjects(&s3.ListObjectsInput{Bucket: aws.String(bucket)})

    if err != nil {
        exitErrorf("Unable to list items in bucket %q, %v", bucket, err)
    }

    for _, item := range resp.Contents {
        fmt.Println("Name:         ", *item.Key)
        fmt.Println("Last modified:", *item.LastModified)
        fmt.Println("Size:         ", *item.Size)
        fmt.Println("Storage class:", *item.StorageClass)
        fmt.Println("")
    }
```

See the complete example on GitHub.

Uploading a File to a Bucket

The Upload function uploads an object to a bucket.

The following example uploads a file to a bucket with the names specified as command-line arguments.

Create the file *s3_upload_object.go*. Add the following statements to import the Go and AWS SDK for Go packages used in the example.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3/s3manager"
)
```

Create a function we use to display errors and exit.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Initialize the session that the SDK uses to load configuration, credential, and region information from the shared config file *~/.aws/config*, and create a `NewUploader` object.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))
    uploader := s3manager.NewUploader(sess)
```

Get the bucket and filename from the command-line arguments, open the file, and defer the file closing until we are done with it. If an error occurs, call `exitErrorF`.

```
func main() {
    if len(os.Args) != 3 {
        exitErrorf("bucket and file name required\nUsage: %s bucket_name filename",
            os.Args[0])
    }

    bucket := os.Args[1]
    filename := os.Args[2]
    file, err := os.Open(filename)

    if err != nil {
        exitErrorf("Unable to open file %q, %v", err)
    }

    defer file.Close()
```

Upload the file to the bucket. If an error occurs, call `exitErrorF`. Otherwise notify the user that the upload succeeded.

```
    _, err = uploader.Upload(&s3manager.UploadInput{
        Bucket: aws.String(bucket),
        Key: aws.String(filename),
        Body: file,
    })

    if err != nil {
        // Print the error and exit.
        exitErrorf("Unable to upload %q to %q, %v", filename, bucket, err)
    }

    fmt.Printf("Successfully uploaded %q to %q\n", filename, bucket)
```

See the complete example on GitHub.

Downloading a File from a Bucket

The Download function downloads an object from a bucket.

The following example downloads an item from a bucket with the names specified as command-line arguments.

Create the file *s3_download_object.go*. Add the following statements to import the Go and AWS SDK for Go packages used in the example.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
    "github.com/aws/aws-sdk-go/service/s3/s3manager"
)
```

Create a function we use to display errors and exit.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Initialize the session that the SDK uses to load configuration, credential, and region information from the shared config file *~/.aws/config*, and create a `NewDownloader` object.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))
    downloader := s3manager.NewDownloader(sess)
```

Get the bucket and filename from the command-line arguments. If there aren't two arguments, call `exitErrorf`.

```
func main() {
    if len(os.Args) != 3 {
        exitErrorf("Bucket and item names required\nUsage: %s bucket_name item_name",
            os.Args[0])
    }

    bucket := os.Args[1]
    item := os.Args[2]
```

Create the file and defer file closing until we are done downloading. If an error occurs, call `exitErrorf`.

```
    file, err := os.Create(item)

    if err != nil {
        exitErrorf("Unable to open file %q, %v", err)
    }

    defer file.Close()
```

Download the item from the bucket. If an error occurs, call `exitErrorf`. Otherwise notify the user that the download succeeded.

```
    numBytes, err := downloader.Download(file,
        &s3.GetObjectInput{
            Bucket: aws.String(bucket),
            Key:    aws.String(item),
        })
```

```
    if err != nil {
        exitErrorf("Unable to download item %q, %v", item, err)
    }

    fmt.Println("Downloaded", file.Name(), numBytes, "bytes")
```

See the complete example on GitHub.

Copying an Item from one Bucket to Another

The CopyObject function copies an object from one bucket to another.

The following example copies an item from one bucket to another with the names specified as command-line arguments.

Create the file *s3_copy_object.go*. Add the following statements to import the Go and AWS SDK for Go packages used in the example.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

Create a function we use to display errors and exit.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Get the names of the bucket containing the item, the item to copy, and the name of the bucket to which the item is copied. If there aren't four command-line arguments, call `exitErrorf`.

```
func main() {
    if len(os.Args) != 4 {
        exitErrorf("Bucket, item, and other bucket names required\nUsage: go run
 s3_copy_object bucket item other-bucket")
    }

    bucket := os.Args[1]
    item := os.Args[2]
    other := os.Args[3]
```

Initialize the session that the SDK uses to load configuration, credential, and region information from the shared config file *~/.aws/config*, and create a new Amazon S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Call CopyObject, with the names of the bucket containing the item, the item to copy, and the name of the bucket to which the item is copied. If an error occurs, call `exitErrorf`. If no error occurs, wait for the item to be copied.

```
    source := bucket + "/" + item
    _, err := svc.CopyObject(&s3.CopyObjectInput{Bucket: aws.String(other), CopySource:
 aws.String(source), Key: aws.String(item)})

    if err != nil {
        exitErrorf("Unable to copy item from bucket %q to bucket %q, %v", bucket, other,
 err)
    }

    // Wait to see if the item got copied
    err = svc.WaitUntilObjectExists(&s3.HeadObjectInput{Bucket: aws.String(other), Key:
 aws.String(item)})
```

If the `WaitUntilObjectExists` call returns an error, If an error occurs, call `exitErrorf`, otherwise notify the user that the copy succeeded.

```
    if err != nil {
        exitErrorf("Error occurred while waiting for item %q to be copied to bucket %q,
 %v", bucket, item, other, err)
    }

    fmt.Printf("Item %q successfully copied from bucket %q to bucket %q\n", item, bucket,
 other)
```

See the complete example on GitHub.

Deleting an Item in a Bucket

The DeleteObject function deletes an object from a bucket.

The following example deletes an item from a bucket with the names specified as command-line arguments.

Create the file *s3_delete_object.go*. Add the following statements to import the Go and AWS SDK for Go packages used in the example.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

Create a function we use to display errors and exit.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Get the name of the bucket and object to delete.

```
func main() {
    if len(os.Args) != 3 {
        exitErrorf("Bucket and object name required\nUsage: %s bucket_name object_name",
            os.Args[0])
    }

    bucket := os.Args[1]
    obj := os.Args[2]
```

Initialize the session that the SDK uses to load configuration, credential, and region information from the shared config file *~/.aws/config*, and create a new Amazon S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Call DeleteObject, passing in the names of the bucket and object to delete. If an error occurs, call `exitErrorf`. If no error occurs, wait until the object is deleted.

```
    _, err := svc.DeleteObject(&s3.DeleteObjectInput{Bucket: aws.String(bucket), Key:
 aws.String(obj)})

    if err != nil {
        exitErrorf("Unable to delete object %q from bucket %q, %v", obj, bucket, err)
    }

    err = svc.WaitUntilObjectNotExists(&s3.HeadObjectInput{
        Bucket: aws.String(bucket),
        Key:    aws.String(obj),
    })
```

If `WaitUntilObjectNotExists` returns an error, call `exitErrorf`. Otherwise, inform the user that the object was successfully deleted.

```
    if err != nil {
        exitErrorf("Error occurred while waiting for object %q to be deleted, %v", obj)
    }
```

See the complete example on GitHub.

Deleting all the Items in a Bucket

The DeleteObjects function deletes objects from a bucket.

The following example deletes all of the items from a bucket with the bucket name specified as a command-line argument.

Create the file *s3_delete_objects.go*. Add the following statements to import the Go and AWS SDK for Go packages used in the example.

```
package main

import (
    "fmt"
    "os"
```

```
    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

Create a function we use to display errors and exit.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Get the name of the bucket.

```
func main() {
    if len(os.Args) != 2 {
        exitErrorf("Bucket name required\nUsage: %s BUCKET",
            os.Args[0])
    }

    bucket := os.Args[1]
```

Initialize the session that the SDK uses to load configuration, credential, and region information from the shared config file *~/.aws/config*, and create a new Amazon S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Create the list of objects to delete from the list of items in the bucket. If an error occurs, call `exitErrorf`.

```
    resp, err := svc.ListObjects(&s3.ListObjectsInput{Bucket: aws.String(bucket)})

    if err != nil {
        exitErrorf("Unable to list items in bucket %q, %v", bucket, err)
    }

    num_objs := len(resp.Contents)

    // Create Delete object with slots for the objects to delete
    var items s3.Delete
    var objs = make([]*s3.ObjectIdentifier, num_objs)

    for i, o := range resp.Contents {
        // Add objects from command line to array
        objs[i] = &s3.ObjectIdentifier{Key: aws.String(*o.Key)}
    }

    // Add list of objects to delete to Delete object
    items.SetObjects(objs)
```

Call DeleteObjects, passing in the name of the bucket and the list of objects to delete. If an error occurs, call `exitErrorf`. If no error occurs, inform the user of the number of objects deleted.

```
    _, err = svc.DeleteObjects(&s3.DeleteObjectsInput{Bucket: &bucket, Delete: &items})
```

```
    if err != nil {
        exitErrorf("Unable to delete objects from bucket %q, %v", bucket, err)
    }

    fmt.Println("Deleted", num_objs, "object(s) from bucket", bucket)
```

See the complete example on GitHub.

Restoring a Bucket Item

The RestoreObject function restores an item in a bucket.

The following example restores the items in a bucket with the names specified as command-line arguments.

Create the file *s3_restore_object.go*. Add the following statements to import the Go and AWS SDK for Go packages used in the example.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

Create a function we use to display errors and exit.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

The program requires two arguments, the names of the bucket and object to restore.

```
func main() {
    if len(os.Args) != 3 {
        exitErrorf("Bucket name and object name required\nUsage: %s bucket_name
 object_name",
            os.Args[0])
    }

    bucket := os.Args[1]
    obj := os.Args[2]
```

Initialize the session that the SDK uses to load configuration, credential, and region information from the shared config file *~/.aws/config*, and create a new Amazon S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Call RestoreObject, passing in the bucket and object names and the number of days to temporarily restore. If an error occurs, call `exitErrorf`. Otherwise, inform the user that they bucket should be restored in the next 4 hours or so.

```
    _, err := svc.RestoreObject(&s3.RestoreObjectInput{Bucket: aws.String(bucket), Key:
 aws.String(obj), RestoreRequest: &s3.RestoreRequest{Days: aws.Int64(30)}})

    if err != nil {
        exitErrorf("Could not restore %s in bucket %s, %v", obj, bucket, err)
    }

    fmt.Printf("%q should be restored to %q in about 4 hours\n", obj, bucket)
```

See the complete example on GitHub.

Deleting a Bucket

The DeleteBucket function deletes a bucket.

The following example deletes the bucket with the name specified as a command-line argument.

Create the file *s3_delete_bucket.go*. Import the following Go and AWS SDK for Go packages.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

Create a function we use to display errors and exit.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

The program requires one argument, the name of the bucket to delete. If the argument is not supplied, call `exitErrorf`.

```
    if len(os.Args) != 2 {
        exitErrorf("bucket name required\nUsage: %s bucket_name", os.Args[0])
    }

    bucket := os.Args[1]
```

Initialize the session that the SDK uses to load configuration, credentials, and region information from the shared config file *~/.aws/config*, and create a new S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Call DeleteBucket, passing in the bucket name. If an error occurs, call `exitErrorf`. If there are no errors, wait for a notification that the bucket was deleted.

```
_, err := svc.DeleteBucket(&s3.DeleteBucketInput{
    Bucket: aws.String(bucket),
})

if err != nil {
    exitErrorf("Unable to delete bucket %q, %v", bucket, err)
}

// Wait until bucket is deleted before finishing
fmt.Printf("Waiting for bucket %q to be deleted...\n", bucket)

err = svc.WaitUntilBucketNotExists(&s3.HeadBucketInput{
    Bucket: aws.String(bucket),
})
```

If `WaitUntilBucketNotExists` returns an error, call `exitErrorf`. Otherwise, inform the user that the bucket was successfully deleted.

```
if err != nil {
    exitErrorf("Error occurred while waiting for bucket to be deleted, %v", bucket)
}

fmt.Printf("Bucket %q successfully deleted\n", bucket)
```

See the complete example on GitHub.

Creating Pre-Signed URLs for Amazon S3 Buckets with Go

This Go example shows you how to obtain a pre-signed URL for an Amazon S3 bucket. You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

In this example, a series of Go routines are used to obtain a pre-signed URL for an Amazon S3 bucket using either GetObject or a PUT operation. A pre-signed URL allows you to grant temporary access to users who don't have permission to directly run AWS operations in your account. A pre-signed URL is signed with your credentials and can be used by any user.

- Presign

Prerequisites

- You have set up (p. 2) and configured (p. 4) the SDK.
- You are familiar with pre-signed URLs. To learn more, see Uploading Objects Using Pre-Signed URLs in the *Amazon S3 Developer Guide*.

Generate a Pre-Signed URL for a GetObject Operation

To generate a pre-signed URL, use the Presign method on the `request` object. You must set an expiration value because the AWS SDK for Go doesn't set one by default.

The following example generates a pre-signed URL that enables you to temporarily share a file without making it public. Anyone with access to the URL can view the file.

```
svc := s3.New(session.New(&aws.Config{Region: aws.String("us-west-2")}))
req, _ := svc.GetObjectRequest(&s3.GetObjectInput{
    Bucket: aws.String("myBucket"),
    Key:    aws.String("myKey"),
})
urlStr, err := req.Presign(15 * time.Minute)

if err != nil {
    log.Println("Failed to sign request", err)
}

log.Println("The URL is", urlStr)
```

If the SDK has not retrieved your credentials before calling `Presign`, it will get them to generate the pre-signed URL.

Generate a Pre-Signed URL for an Amazon S3 PUT Operation with a Specific Payload

You can generate a pre-signed URL for a PUT operation that checks whether users upload the correct content. When the SDK pre-signs a request, it computes the checksum of the request body and generates an MD5 checksum that is included in the pre-signed URL. Users must upload the same content that produces the same MD5 checksum generated by the SDK; otherwise, the operation fails. This is not the Content-MD5, but the signature. To enforce Content-MD5, simply add the header to the request.

The following example adds a `Body` field to generate a pre-signed PUT operation that requires a specific payload to be uploaded by users.

```
svc := s3.New(session.New(&aws.Config{Region: aws.String("us-west-2")}))
req, _ := svc.PutObjectRequest(&s3.PutObjectInput{
    Bucket: aws.String("myBucket"),
    Key:    aws.String("myKey"),
    Body:   strings.NewReader("EXPECTED CONTENTS"),
})
str, err := req.Presign(15 * time.Minute)

log.Println("The URL is:", str, " err:", err)
```

If you omit the `Body` field, users can write any contents to the given object.

This example shows the enforcing of Content-MD5.

```
   h := md5.New()
   content := strings.NewReader("")
   content.WriteTo(h)
   svc := s3.New(
       session.New(),
       &aws.Config{Region: aws.String("us-west-2")},
   )

   r, _ := svc.PutObjectRequest(&s3.PutObjectInput{
       Bucket: aws.String("testBucket"),
       Key:    aws.String("testKey"),
   })

   md5s := base64.StdEncoding.EncodeToString(h.Sum(nil))
```

```
    r.HTTPRequest.Header.Set("Content-MD5", md5s)
    url, err := r.Presign(15 * time.Minute)
    if err != nil {
        fmt.Println("error presigning request", err)
        return
    }

    req, err := http.NewRequest("PUT", url, strings.NewReader(""))
    req.Header.Set("Content-MD5", md5s)
    if err != nil {
        fmt.Println("error creating request", url)
        return
    }

    resp, err := http.DefaultClient.Do(req)
    fmt.Println(resp, err)
```

Using an Amazon S3 Bucket as a Static Web Host with Go

This Go example shows you how to configure an Amazon S3 bucket to act as a static web host. You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

In this example, you use a series of Go routines to configure any of your buckets to act as a static web host. The routines use the AWS SDK for Go to configure a selected Amazon S3 bucket using these methods of the Amazon S3 client class:

- GetBucketWebsite
- PutBucketWebsite
- DeleteBucketWebsite

For more information about using an Amazon S3 bucket as a static web host, see Hosting a Static Website on Amazon S3 in the *Amazon S3 Developer Guide*.

Prerequisites

- You have set up (p. 2), and configured (p. 4) the AWS SDK for Go.
- You are familiar with hosting static websites on Amazon S3. To learn more, see Hosting a Static Website on Amazon S3 in the *Amazon S3 Developer Guide*.

Retrieving a Bucket's Website Configuration

Create a new Go file named `s3_get_bucket_website.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
```

```
    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

This routine requires you to pass in an argument containing the name of the bucket for which you want to get website configuration.

```
func main() {
    if len(os.Args) != 2 {
        exitErrorf("bucket name required\nUsage: %s bucket_name", os.Args[0])
    }
    bucket := os.Args[1]
```

Initialize a session that the SDK will use to load configuration, credentials, and region info from the shared config file, ~/.aws/config, and create a new S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Call GetBucketWebsite to get the bucket configuration. You should check for the `NoSuchWebsiteConfiguration` error code, which tells you that the bucket doesn't have a website configured.

```
    result, err := svc.GetBucketWebsite(&s3.GetBucketWebsiteInput{
        Bucket: aws.String(bucket),
    })
    if err != nil {
        if awsErr, ok := err.(awserr.Error); ok && awsErr.Code() ==
 "NoSuchWebsiteConfiguration" {
            exitErrorf("Bucket %s does not have website configuration\n", bucket)
        }
        exitErrorf("Unable to get bucket website config, %v", err)
    }
```

Print the bucket's website configuration.

```
    fmt.Println("Bucket Website Configuration:")
    fmt.Println(result)
}

func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Setting a Bucket's Website Configuration

Create a Go file named `s3_set_bucket_website.go` and add the code below. The Amazon S3PutBucketWebsite operation sets the website configuration on a bucket, replacing any existing configuration.

Create a new Go file named `s3_create_bucket.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"
    "path/filepath"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

This routine requires you to pass in an argument containing the name of the bucket and the index suffix page.

```
func main() {
    if len(os.Args) != 3 {
        exitErrorf("bucket name and index suffix page required\nUsage: %s bucket_name
 index_page [error_page]",
            filepath.Base(os.Args[0]))
    }
    bucket := fromArgs(os.Args, 1)
    indexSuffix := fromArgs(os.Args, 2)
    errorPage := fromArgs(os.Args, 3)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Create the parameters to be passed in to PutBucketWebsite, including the bucket name and index document details. If the user passed in an error page when calling the routine, also add that to the parameters.

```
    params := s3.PutBucketWebsiteInput{
        Bucket: aws.String(bucket),
        WebsiteConfiguration: &s3.WebsiteConfiguration{
            IndexDocument: &s3.IndexDocument{
                Suffix: aws.String(indexSuffix),
            },
        },
    }

    if len(errorPage) > 0 {
        params.WebsiteConfiguration.ErrorDocument = &s3.ErrorDocument{
            Key: aws.String(errorPage),
        }
    }
```

Call `PutBucketWebsite`, passing in the parameters you just defined. If an error occurs, print the errordetails and exit the routine.

```
    _, err := svc.PutBucketWebsite(&params)
    if err != nil {
```

```
        exitErrorf("Unable to set bucket %q website configuration, %v",
            bucket, err)
    }

    fmt.Printf("Successfully set bucket %q website configuration\n", bucket)
}
```

Deleting Website Configuration on a Bucket

Create a Go file named `s3_delete_bucket_website.go`. The following code shows you how to delete the bucket's website configuration.

Create a new Go file named `s3_upload.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"
    "os"
    "path/filepath"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

This routine requires you to pass in the name of the bucket for which you want to delete the website configuration.

```
func main() {
    if len(os.Args) != 2 {
        exitErrorf("bucket name required\nUsage: %s bucket_name",
            filepath.Base(os.Args[0]))
    }
    bucket := os.Args[1]
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Call *DeleteBucketWebsite* and pass in the name of the bucket to complete the action.

```
    _, err := svc.DeleteBucketWebsite(&s3.DeleteBucketWebsiteInput{
        Bucket: aws.String(bucket),
    })
    if err != nil {
        exitErrorf("Unable to delete bucket %q website configuration, %v",
            bucket, err)
    }

    fmt.Printf("Successfully delete bucket %q website configuration\n", bucket)
}
```

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Working with Amazon S3 CORS Permissions with Go

This Go example shows you how to list Amazon S3 buckets and configure CORS and bucket logging. You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

In this example, a series of Go routines are used to list your Amazon S3 buckets and to configure CORS and bucket logging. The routines use the AWS SDK for Go to configure a selected Amazon S3 bucket using these methods of the Amazon S3 client class:

- GetBucketCors
- PutBucketCors

If you are unfamiliar with using CORS configuration with an Amazon S3 bucket, it is worth your time to read Cross-Origin Resource Sharing (CORS) in the *Amazon S3 Developer Guide* before proceeding.

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with using CORS configuration with an Amazon S3 bucket. To learn more, see Cross-Origin Resource Sharing (CORS) in the *Amazon S3 Developer Guide*.

Configuring CORS on the Bucket

Create a new Go file named `s3_set_cors.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "flag"
    "fmt"
    "os"
    "strings"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

This routine configures CORS rules for a bucket by setting the allowed HTTP methods.

It requires the bucket name, and can also take a space-separated list of HTTP methods. Using the Go Language's `flag` package, it parses the input and validates the bucket name.

```
func main() {
    var bucket string
```

```
    flag.StringVar(&bucket, "b", "", "Bucket to set CORS on, (required)")
    flag.Parse()
    if len(bucket) == 0 {
        exitErrorf("-b <bucket> Bucket name required")
    }
    methods := filterMethods(flag.Args())
```

Notice the helper method, `filterMethods`, which ensures the methods passed in are uppercase.

```
func filterMethods(methods []string) []string {
    filtered := make([]string, 0, len(methods))
    for _, m := range methods {
        v := strings.ToUpper(m)
        switch v {
        case "POST", "GET", "PUT", "PATCH", "DELETE":
            filtered = append(filtered, v)
        }
    }
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Create a new CORSRule to set up the CORS configuration.

```
    rule := s3.CORSRule{
        AllowedHeaders: aws.StringSlice([]string{"Authorization"}),
        AllowedOrigins: aws.StringSlice([]string{"*"}),
        MaxAgeSeconds:  aws.Int64(3000),

        // Add HTTP methods CORS request that were specified in the CLI.
        AllowedMethods: aws.StringSlice(methods),
    }
```

Add the `CORSRule` to the `PutBucketCorsInput` structure, call `PutBucketCors` with that structure, and print a success or error message.

```
    params := s3.PutBucketCorsInput{
        Bucket: aws.String(bucket),
        CORSConfiguration: &s3.CORSConfiguration{
            CORSRules: []*s3.CORSRule{&rule},
        },
    }

    _, err := svc.PutBucketCors(&params)
    if err != nil {
        // Print the error message
        exitErrorf("Unable to set Bucket %q's CORS, %v", bucket, err)
    }

    // Print the updated CORS config for the bucket
    fmt.Printf("Updated bucket %q CORS for %v\n", bucket, methods)
}

func exitErrorf(msg string, args ...interface{}) {
```

```
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Working with Amazon S3 Bucket Policies with Go

This Go example show you how to retrieve, display, and set Amazon S3 bucket polices. You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

In this example, a series Go routines are used to retrieve or set a bucket policy on an Amazon S3 bucket. The routines use the AWS SDK for Go to configure policy for a selected Amazon S3 bucket using these methods of the Amazon S3 client class:

- GetBucketPolicy
- PutBucketPolicy
- DeleteBucketPolicy

For more information about bucket policies for Amazon S3 buckets, see Using Bucket Policies and User Policies in the *Amazon S3 Developer Guide*.

Prerequisites

- You have set up (p. 2), and configured (p. 4) the AWS SDK for Go.
- You are familiar with Amazon S3 bucket polices. To learn more, see Using Bucket Policies and User Policies in the *Amazon S3 Developer Guide*.

Retrieving and Displaying a Bucket Policy

Create a new Go file named `s3_get_bucket_policy.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "bytes"
    "encoding/json"
    "fmt"
    "os"
    "path/filepath"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

This routine prints the policy for a bucket. If the bucket doesn't exist, or there was an error, an error message is printed instead. It requires the bucket name as input.

```
func main() {
    if len(os.Args) != 2 {
        exitErrorf("bucket name required\nUsage: %s bucket_name",
            filepath.Base(os.Args[0]))
```

```
    }
    bucket := os.Args[1]
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Call GetBucketPolicy to fetch the policy, then display any errors.

```
    result, err := svc.GetBucketPolicy(&s3.GetBucketPolicyInput{
        Bucket: aws.String(bucket),
    })
    if err != nil {
        // Special error handling for the when the bucket doesn't
        // exists so we can give a more direct error message from the CLI.
        if aerr, ok := err.(awserr.Error); ok {
            switch aerr.Code() {
            case s3.ErrCodeNoSuchBucket:
                exitErrorf("Bucket %q does not exist.", bucket)
            case "NoSuchBucketPolicy":
                exitErrorf("Bucket %q does not have a policy.", bucket)
            }
        }
        exitErrorf("Unable to get bucket %q policy, %v.", bucket, err)
    }
```

Use Go's JSON package to print the Policy JSON returned by the call.

```
    out := bytes.Buffer{}
    policyStr := aws.StringValue(result.Policy)
    if err := json.Indent(&out, []byte(policyStr), "", "  "); err != nil {
        exitErrorf("Failed to pretty print bucket policy, %v.", err)
    }

    fmt.Printf("%q's Bucket Policy:\n", bucket)
    fmt.Println(out.String())
}
```

The `exitError` function is used to deal with printing any errors.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Setting Bucket Policy

This routine sets the policy for a bucket. If the bucket doesn't exist, or there was an error, an error message will be printed instead. It requires the bucket name as input. It also requires the same Go and AWS SDK for Go packages as the previous example, except for the `bytes` Go package.

```
import (
```

```
    "encoding/json"
    "fmt"
    "os"
    "path/filepath"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

Add the main function and parse the arguments to get the bucket name.

```
func main() {
    if len(os.Args) != 2 {
        exitErrorf("bucket name required\nUsage: %s bucket_name",
            filepath.Base(os.Args[0]))
    }
    bucket := os.Args[1]
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config, and create a new S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Create a policy using the map interface, filling in the bucket as the resource.

```
    readOnlyAnonUserPolicy := map[string]interface{}{
        "Version": "2012-10-17",
        "Statement": []map[string]interface{}{
            {
                "Sid":       "AddPerm",
                "Effect":    "Allow",
                "Principal": "*",
                "Action": []string{
                    "s3:GetObject",
                },
                "Resource": []string{
                    fmt.Sprintf("arn:aws:s3:::%s/*", bucket),
                },
            },
        },
    }
```

Use Go's JSON package to marshal the policy into a JSON value so that it can be sent to S3.

```
    policy, err := json.Marshal(readOnlyAnonUserPolicy)
    if err != nil {
        exitErrorf("Failed to marshal policy, %v", err)
    }
```

Call the S3 client's PutBucketPolicy to PUT the policy for the bucket and print the results.

```
    _, err = svc.PutBucketPolicy(&s3.PutBucketPolicyInput{
        Bucket: aws.String(bucket),
```

```
        Policy: aws.String(string(policy)),
    })
    if err != nil {
        if aerr, ok := err.(awserr.Error); ok && aerr.Code() == s3.ErrCodeNoSuchBucket {
            // Special error handling for the when the bucket doesn't
            // exists so we can give a more direct error message from the CLI.
            exitErrorf("Bucket %q does not exist", bucket)
        }
        exitErrorf("Unable to set bucket %q policy, %v", bucket, err)
    }

    fmt.Printf("Successfully set bucket %q's policy\n", bucket)
}
```

The `exitError` function is used to deal with printing any errors.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Working with Amazon S3 Bucket ACLs in Go

The following examples use AWS SDK for Go functions to:

- Get the ACLs on a bucket
- Get the ACLs on a bucket item
- Add a new user to the ACLs on a bucket
- Add a new user to the ACLs on a bucket item

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

In these examples, a series of Go routines are used to manage ACLs on your Amazon S3 buckets. The routines use the AWS SDK for Go to perform Amazon S3 bucket operations using the following methods of the Amazon S3 client class:

- GetBucketAcl
- GetObjectAcl
- PutBucketAcl
- PutObjectAcl

Prerequisites

- You have set up (p. 2), and configured (p. 4) the AWS SDK for Go.
- You are familiar with Amazon S3 bucket ACLs. To learn more, see Managing Access with ACLs in the *Amazon S3 Developer Guide*.

Getting a Bucket ACL

The GetBucketAcl function gets the ACLs on a bucket.

The following example gets the ACLs on a bucket with the name specified as a command-line argument.

Create the file *s3_get_bucket_acl.go*. Add the following statements to import the Go and AWS SDK for Go packages used in the example.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

Create a function to display errors and exit.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

This example requires one input parameter, the name of the bucket. If the name is not supplied, we call the error function and exit.

```
func main() {
    if len(os.Args) != 2 {
        exitErrorf("Bucket name required\nUsage: go run", os.Args[0], "BUCKET")
    }

    bucket := os.Args[1]
```

Initialize the session that the SDK uses to load configuration, credential, and region information from the shared config file *~/.aws/config*, and create a new Amazon S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Call GetBucketAcl, passing in the name of the bucket. If an error occurs, call `exitErrorf`. If no error occurs, loop through the results and print out the name, type, and permssion for the grantees.

```
    result, err := svc.GetBucketAcl(&s3.GetBucketAclInput{Bucket: &bucket})

    if err != nil {
        exitErrorf(err.Error())
    }

    fmt.Println("Owner:", *result.Owner.DisplayName)

    fmt.Println("Grants")

    for _, g := range result.Grants {
        fmt.Println("  Grantee:   ", *g.Grantee.DisplayName)
        fmt.Println("  Type:      ", *g.Grantee.Type)
        fmt.Println("  Permission:", *g.Permission)
```

```
        fmt.Println("")
    }
```

See the complete example on GitHub.

Setting a Bucket ACL

The PutBucketAcl function sets the ACLs on a bucket.

The following example gives a user, by email address, access to a bucket with the bucket name and email address specified as command-line arguments. The user can also supply a permission argumement, but if it is not supplied, the user is given READ access to the bucket.

Create the file *s3_put_bucket_acl.go*. Add the following statements to import the Go and AWS SDK for Go packages used in the example.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

Create a function to display errors and exit.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Get the two required input parameters. If the optional permission parameter is supplied, make sure it is one of the allowed values. If not, print an error message and quit.

```
func main() {
    if len(os.Args) < 3 {
        exitErrorf("Bucket name and email address required; permission optional (READ if
 omitted)\nUsage: go run", os.Args[0], "BUCKET EMAIL [PERMISSION]")
    }

    bucket := os.Args[1]
    address := os.Args[2]

    permission := "READ"

    if len(os.Args) == 4 {
        permission = os.Args[3]

        if !(permission == "FULL_CONTROL" || permission == "WRITE" || permission ==
 "WRITE_ACP" || permission == "READ" || permission == "READ_ACP") {
            fmt.Println("Illegal permission value. It must be one of:")
            fmt.Println("FULL_CONTROL, WRITE, WRITE_ACP, READ, or READ_ACP")
            os.Exit(1)
        }
    }
```

Initialize the session that the SDK uses to load configuration, credential, and region information from the shared config file *~/.aws/config*, and create a new Amazon S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Get the existing ACLs and append the new user to the list. If we encounter an error while retrieving the list, print an error message and quit.

```
    result, err := svc.GetBucketAcl(&s3.GetBucketAclInput{Bucket: &bucket})

    if err != nil {
        exitErrorf(err.Error())
    }

    owner := *result.Owner.DisplayName
    owner_id := *result.Owner.ID

    // Existing grants
    grants := result.Grants

    // Create new grantee to add to grants
    var new_grantee s3.Grantee = s3.Grantee{EmailAddress: &address, Type: &user_type}
    var new_grant s3.Grant = s3.Grant{Grantee: &new_grantee, Permission: &permission}

    // Add them to the grants
    grants = append(grants, &new_grant)
```

Build the parameter list for the call based on the existing ACLs and the new user information.

```
    params := &s3.PutBucketAclInput{
        Bucket: &bucket,
        AccessControlPolicy: &s3.AccessControlPolicy{
            Grants: grants,
            Owner: &s3.Owner{
                DisplayName: &owner,
                ID:          &owner_id,
            },
        },
    }
```

Call PutBucketAcl, passing in the parameter list. If an error occurs, display a message and quit. Otherwise display a message indicating success.

```
    _, err = svc.PutBucketAcl(params)

    if err != nil {
        exitErrorf(err.Error())
    }

    fmt.Println("Congratulations. You gave user with email address", address, permission,
"permission to bucket", bucket)
```

See the complete example on GitHub.

Getting a Bucket Object ACL

The PutObjectAcl function sets the ACLs on a bucket item.

The following example gets the ACLs for a bucket item with the bucket and item name specified as command-line arguments.

Create the file *s3_get_bucket_object_acl.go*. Add the following statements to import the Go and AWS SDK for Go packages used in the example.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

Create a function to display errors and exit.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

This example requires two input parameters, the names of the bucket and object. If either name is not supplied, call the error function and exit.

```
func main() {
    if len(os.Args) != 3 {
        exitErrorf("Bucket and object names required\nUsage: go run", os.Args[0], "BUCKET
 OBJECT")
    }

    bucket := os.Args[1]
    key := os.Args[2]
```

Initialize the session that the SDK uses to load configuration, credential, and region information from the shared config file *~/.aws/config*, and create a new Amazon S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Call GetObjectAcl, passing in the names of the bucket and object. If an error occurs, call `exitErrorf`. If no error occurs, loop through the results and print out the name, type, and permssion for the grantees.

```
    result, err := svc.GetObjectAcl(&s3.GetObjectAclInput{Bucket: &bucket, Key: &key})

    if err != nil {
        exitErrorf(err.Error())
    }

    fmt.Println("Owner:", *result.Owner.DisplayName)

    fmt.Println("Grants")
```

```
    for _, g := range result.Grants {
        fmt.Println("  Grantee:    ", *g.Grantee.DisplayName)
        fmt.Println("  Type:       ", *g.Grantee.Type)
        fmt.Println("  Permission:", *g.Permission)
        fmt.Println("")
    }
```

See the complete example on GitHub.

Setting a Bucket Object ACL

The PutObjectAcl function sets the ACLs on a bucket item.

The following example gives a user, by email address, access to a bucket item with the bucket and item names and email address specified as command-line arguments. The user can also supply a permission argumement, but if it is not supplied, the user is given READ access to the bucket.

Create the file *s3_put_bucket_object_acl.go*. Add the following statements to import the Go and AWS SDK for Go packages used in the example.

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
)
```

Create a function to display errors and exit.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Get the three required input parameters. If the optional permission parameter is supplied, make sure it is one of the allowed values. If not, print an error message and quit.

```
func main() {
    if len(os.Args) < 4 {
        exitErrorf("Bucket name, object name, and email address required; permission
 optional (READ if omitted)\nUsage: go run", os.Args[0], "BUCKET OBJECT EMAIL
 [PERMISSION]")
    }

    bucket := os.Args[1]
    key := os.Args[2]
    address := os.Args[3]

    permission := "READ"

    if len(os.Args) == 5 {
        permission = os.Args[4]

        if !(permission == "FULL_CONTROL" || permission == "WRITE" || permission ==
 "WRITE_ACP" || permission == "READ" || permission == "READ_ACP") {
            fmt.Println("Illegal permission value. It must be one of:")
            fmt.Println("FULL_CONTROL, WRITE, WRITE_ACP, READ, or READ_ACP")
```

```
            os.Exit(1)
        }
    }
```

Initialize the session that the SDK uses to load configuration, credential, and region information from the shared config file *~/.aws/config*, and create a new Amazon S3 service client.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create S3 service client
    svc := s3.New(sess)
```

Build the parameter list for the call based on the existing ACLs and the new user information.

```
    user_type := "AmazonCustomerByEmail"
    result, err := svc.GetObjectAcl(&s3.GetObjectAclInput{Bucket: &bucket, Key: &key})

    if err != nil {
        exitErrorf(err.Error())
    }

    owner := *result.Owner.DisplayName
    owner_id := *result.Owner.ID

    // Existing grants
    grants := result.Grants

    // Create new grantee to add to grants
    var new_grantee s3.Grantee = s3.Grantee{EmailAddress: &address, Type: &user_type}
    var new_grant s3.Grant = s3.Grant{Grantee: &new_grantee, Permission: &permission}

    // Add them to the grants
    grants = append(grants, &new_grant)

    params := &s3.PutObjectAclInput{
        Bucket: &bucket,
        Key:    &key,
        AccessControlPolicy: &s3.AccessControlPolicy{
            Grants: grants,
            Owner: &s3.Owner{
                DisplayName: &owner,
                ID:          &owner_id,
            },
        },
    }
```

Call PutObjectAcl, passing in the parameter list. If an error occurs, display a message and quit. Otherwise display a message indicating success.

```
    _, err = svc.PutObjectAcl(params)

    if err != nil {
        exitErrorf(err.Error())
    }

    fmt.Println("Congratulations. You gave user with email address", address, permission,
"permission to bucket", bucket, "object", key)
```

See the complete example on GitHub.

Amazon SQS with Go Examples

The AWS SDK for Go examples can integrate Amazon SQS into your applications. The examples assume you have already set up and configured the SDK (that is, you've imported all required packages and set your credentials and region). For more information, see Setting Up (p. 2) and SDK Configuration (p. 4).

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

Topics

Using Amazon SQS Queues with Go

These Go examples show you how to:

- List Amazon SQS queues
- Create Amazon SQS queues
- Get Amazon SQS queue URLs
- Delete Amazon SQS queues

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

These examples work with Amazon SQS queues.

The code uses these methods of the Amazon SQS client class:

- CreateQueue
- ListQueues
- GetQueueUrl
- DeleteQueue

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with using Amazon SQS. To learn more, see How Queues Work in the *Amazon SQS Developer Guide*.

Listing Queues

Create a new Go file named `sqs_listqueues.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sqs"
)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a SQS service client.
    svc := sqs.New(sess)
```

Call `ListQueues` passing in `nil` to return all queues. Print any errors or a success message and loop through the queue URLs to print them.

```
    result, err := svc.ListQueues(nil)
    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success")
    // As these are pointers, printing them out directly would not be useful.
    for i, urls := range result.QueueUrls {
        // Avoid dereferencing a nil pointer.
        if urls == nil {
            continue
        }
        fmt.Printf("%d: %s\n", i, *urls)
    }
}
```

Creating Queues

Create a new Go file named `sqs_createqueues.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sqs"
)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config.

```
func main() {
```

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a SQS service client.
    svc := sqs.New(sess)
```

Call `CreateQueue` passing in the new queue name and queue attributes. Print any errors or a success message.

```
    result, err := svc.CreateQueue(&sqs.CreateQueueInput{
        QueueName: aws.String("SQS_QUEUE_NAME"),
        Attributes: map[string]*string{
            "DelaySeconds":           aws.String("60"),
            "MessageRetentionPeriod": aws.String("86400"),
        },
    })
    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success", *result.QueueUrl)
}
```

Getting a Queue URL

Create a new Go file named `sqs_getqueueurl.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sqs"
)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a SQS service client.
    svc := sqs.New(sess)
```

Call `GetQueueUrl` passing in the queue name. Print any errors or a success message.

```
    result, err := svc.GetQueueUrl(&sqs.GetQueueUrlInput{
        QueueName: aws.String("SQS_QUEUE_NAME"),
    })

    if err != nil {
        fmt.Println("Error", err)
        return
```

```
    }

    fmt.Println("Success", *result.QueueUrl)
}
```

Deleting a Queue

Create a new Go file named `sqs_deletequeue.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sqs"
)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a SQS service client.
    svc := sqs.New(sess)
```

Call `DeleteQueue` passing in the queue name. Print any errors or a success message.

```
    result, err := svc.DeleteQueue(&sqs.DeleteQueueInput{
        QueueUrl: aws.String("SQS_QUEUE_URL"),
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success", result)
}
```

Sending and Receiving Messages in Amazon SQS with Go

These Go examples show you how to:

- Send a message to an Amazon SQS queue
- Receive and delete a message from an Amazon SQS queue
- Send and receive messages from an Amazon SQS queue

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

These examples demonstrate sending, receiving, and deleting messages from an Amazon SQS queue.

The code uses these methods of the Amazon SQS client class:

- SendMessage
- ReceiveMessage
- DeleteMessage
- GetQueueUrl

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with the details of Amazon SQS messages. To learn more, see Sending a Message to an Amazon SQS Queue and Receiving and Deleting a Message from an Amazon SQS Queue in the *Amazon SQS Developer Guide*.

Sending a Message to a Queue

Create a new Go file named `sqs_sendmessage.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sqs"
)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    svc := sqs.New(sess)

    // URL to our queue
    qURL := "QueueURL"
```

Now you're ready to send your message. In the example, the message input passed to `SendMessage` represents information about a fiction best seller for a particular week and defines title, author, and weeks on the list values.

```
    result, err := svc.SendMessage(&sqs.SendMessageInput{
        DelaySeconds: aws.Int64(10),
        MessageAttributes: map[string]*sqs.MessageAttributeValue{
```

```
            "Title": &sqs.MessageAttributeValue{
                DataType:    aws.String("String"),
                StringValue: aws.String("The Whistler"),
            },
            "Author": &sqs.MessageAttributeValue{
                DataType:    aws.String("String"),
                StringValue: aws.String("John Grisham"),
            },
            "WeeksOn": &sqs.MessageAttributeValue{
                DataType:    aws.String("Number"),
                StringValue: aws.String("6"),
            },
        },
        MessageBody: aws.String("Information about current NY Times fiction bestseller for
 week of 12/11/2016."),
        QueueUrl:    &qURL,
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success", *result.MessageId)
}
```

Receiving and Deleting a Message from a Queue

Create a new Go file named `sqs_deletemessage.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sqs"
)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    svc := sqs.New(sess)
```

Now you're ready to receive a message from a queue specified by a queue URL. In the example, the `qURL` variable would hold the URL for the queue containing the message.

```
    qURL := "QueueURL"

    result, err := svc.ReceiveMessage(&sqs.ReceiveMessageInput{
        AttributeNames: []*string{
            aws.String(sqs.MessageSystemAttributeNameSentTimestamp),
```

```
        },
        MessageAttributeNames: []*string{
            aws.String(sqs.QueueAttributeNameAll),
        },
        QueueUrl:            &qURL,
        MaxNumberOfMessages: aws.Int64(1),
        VisibilityTimeout:   aws.Int64(0),
        WaitTimeSeconds:     aws.Int64(0),
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    if len(result.Messages) == 0 {
        fmt.Println("Received no messages")
        return
    }
```

After retrieving the message, delete it from the queue with `DeleteMessage`, passing the `ReceiptHandle` returned from the previous call.

```
    resultDelete, err := svc.DeleteMessage(&sqs.DeleteMessageInput{
        QueueUrl:      &qURL,
        ReceiptHandle: result.Messages[0].ReceiptHandle,
    })

    if err != nil {
        fmt.Println("Delete Error", err)
        return
    }

    fmt.Println("Message Deleted", resultDelete)
}
```

Sending and Receiving Messages

Create a new Go file named `sqs_longpolling_receive_message.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "flag"
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sqs"
)
```

Get the queue name and timeout passed from the command.

```
func main() {
    var name string
    var timeout int64
    flag.StringVar(&name, "n", "", "Queue name")
```

```
    flag.Int64Var(&timeout, "t", 20, "(Optional) Timeout in seconds for long polling")
    flag.Parse()

    if len(name) == 0 {
        flag.PrintDefaults()
        exitErrorf("Queue name required")
    }
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a SQS service client.
    svc := sqs.New(sess)
```

Get the Queue. You need to convert the queue name into a URL. You can use the `GetQueueUrl` API call to retrieve the URL. This is needed for receiving messages from the queue. Print any errors.

```
    resultURL, err := svc.GetQueueUrl(&sqs.GetQueueUrlInput{
        QueueName: aws.String(name),
    })
    if err != nil {
        if aerr, ok := err.(awserr.Error); ok && aerr.Code() ==
 sqs.ErrCodeQueueDoesNotExist {
            exitErrorf("Unable to find queue %q.", name)
        }
        exitErrorf("Unable to queue %q, %v.", name, err)
    }
```

Call `ReceiveMessage` to get the latest message from the queue.

```
    result, err := svc.ReceiveMessage(&sqs.ReceiveMessageInput{
        QueueUrl: resultURL.QueueUrl,
        AttributeNames: aws.StringSlice([]string{
            "SentTimestamp",
        }),
        MaxNumberOfMessages: aws.Int64(1),
        MessageAttributeNames: aws.StringSlice([]string{
            "All",
        }),
        WaitTimeSeconds: aws.Int64(timeout),
    })
    if err != nil {
        exitErrorf("Unable to receive message from queue %q, %v.", name, err)
    }

    fmt.Printf("Received %d messages.\n", len(result.Messages))
    if len(result.Messages) > 0 {
        fmt.Println(result.Messages)
    }
}
```

The example uses this utility function.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
```

```
}
```

Managing Visibility Timeout in Amazon SQS Queues with Go

This Go example shows you how to:

- Change visibility timeout with Amazon SQS queues

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

This example manages visibility timeout with Amazon SQS queues, and uses these methods of the Amazon SQS client class:

- CreateQueue
- ListQueues
- GetQueueUrl
- DeleteQueue

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with using Amazon SQS visibility timeout. To learn more, see Visibility Timeout in the *Amazon SQS Developer Guide*.

Changing the Visibility Timeout

Create a new Go file named `sqs_changingvisibility.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
   "fmt"

   "github.com/aws/aws-sdk-go/aws"
   "github.com/aws/aws-sdk-go/aws/session"
   "github.com/aws/aws-sdk-go/service/sqs"
)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config.

```
func main() {
   sess := session.Must(session.NewSessionWithOptions(session.Options{
      SharedConfigState: session.SharedConfigEnable,
   }))
```

```
   // Create a SQS service client.
   svc := sqs.New(sess)
```

Get a message from the queue. Call `ReceiveMessage` passing in the URL of the queue to return details of the next message in the queue. Print any errors, or a message if no message was received.

```
   qURL := "QueueURL"

   result, err := svc.ReceiveMessage(&sqs.ReceiveMessageInput{
      AttributeNames: []*string{
         aws.String(sqs.MessageSystemAttributeNameSentTimestamp),
      },
      MaxNumberOfMessages: aws.Int64(1),
      MessageAttributeNames: []*string{
         aws.String(sqs.QueueAttributeNameAll),
      },
      QueueUrl: &qURL,
   })

   if err != nil {
      fmt.Println("Error", err)
      return
   }

   // Check if we have any messages
   if len(result.Messages) == 0 {
      fmt.Println("Received no messages")
      return
   }
```

If a message was returned, use its receipt handle to set the timeout to 10 hours.

```
   duration := int64(36000)
   resultVisibility, err := svc.ChangeMessageVisibility(&sqs.ChangeMessageVisibilityInput{
      ReceiptHandle:     result.Messages[0].ReceiptHandle,
      QueueUrl:          &qURL,
      VisibilityTimeout: &duration,
   })

   if err != nil {
      fmt.Println("Visibility Error", err)
      return
   }

   fmt.Println("Time Changed", resultVisibility)
}
```

Enabling Long Polling in Amazon SQS Queues with Go

These Go examples show you how to:

- Enable long polling when you create an Amazon SQS queue
- Enable long polling on an existing Amazon SQS queue
- Enable long polling when a message is received

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

Long polling reduces the number of empty responses by allowing Amazon SQS to wait a specified time for a message to become available in the queue before sending a response. Also, long polling eliminates false empty responses by querying all of the servers instead of a sampling of servers. To enable long polling, you must specify a non-zero wait time for received messages. You can do this by setting the `ReceiveMessageWaitTimeSeconds` parameter of a queue or by setting the `WaitTimeSeconds` parameter on a message when it is received.

The code uses these methods of the Amazon SQS client class:

- SetQueueAttributes
- ReceiveMessage
- CreateQueue

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with Amazon SQS polling. To learn more, see Long Polling in the *Amazon SQS Developer Guide*.

Enabling Long Polling When Creating a Queue

This example creates a queue with long polling enabled. If the queue already exists, no error will be returned.

Create a new Go file named `sqs_longpolling_create_queue.go`. You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "flag"
    "fmt"
    "os"
    "strconv"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sqs"
)
```

Get the queue name passed in by the user.

```
func main() {
    var name string
    var timeout int
    flag.StringVar(&name, "n", "", "Queue name")
    flag.IntVar(&timeout, "t", 20, "(Optional) Timeout in seconds for long polling")
    flag.Parse()

    if len(name) == 0 {
        flag.PrintDefaults()
        exitErrorf("Queue name required")
    }
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a SQS service client.
    svc := sqs.New(sess)
```

Create the queue with long polling enabled. Print any errors or a success message.

```
    result, err := svc.CreateQueue(&sqs.CreateQueueInput{
        QueueName: aws.String(name),
        Attributes: aws.StringMap(map[string]string{
            "ReceiveMessageWaitTimeSeconds": strconv.Itoa(timeout),
        }),
    })
    if err != nil {
        exitErrorf("Unable to create queue %q, %v.", name, err)
    }

    fmt.Printf("Succesffuly created queue %q. URL: %s\n", name,
        aws.StringValue(result.QueueUrl))
}
```

The example uses this utility function.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Enabling Long Polling on an Existing Queue

Create a new Go file named `sqs_longpolling_existing_queue.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "flag"
    "fmt"
    "os"
    "strconv"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sqs"
)
```

This example takes two flags, the -n flag is the queue name, and the -t flag contains the timeout value.

```
func main() {
    var name string
    var timeout int
    flag.StringVar(&name, "n", "", "Queue name")
    flag.IntVar(&timeout, "t", 20, "(Optional) Timeout in seconds for long polling")
```

```
    flag.Parse()

    if len(name) == 0 {
        flag.PrintDefaults()
        exitErrorf("Queue name required")
    }
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a SQS service client.
    svc := sqs.New(sess)
```

You need to convert the queue name into a URL. Make the `GetQueueUrl` API call to retrieve the URL. This is needed for setting attributes on the queue.

```
    resultURL, err := svc.GetQueueUrl(&sqs.GetQueueUrlInput{
        QueueName: aws.String(name),
    })
    if err != nil {
        if aerr, ok := err.(awserr.Error); ok && aerr.Code() ==
 sqs.ErrCodeQueueDoesNotExist {
            exitErrorf("Unable to find queue %q.", name)
        }
        exitErrorf("Unable to get queue %q, %v.", name, err)
    }
```

Update the queue to enable long polling with a call to `SetQueueAttributes`, passing in the queue URL. Print any errors or a success message.

```
    _, err = svc.SetQueueAttributes(&sqs.SetQueueAttributesInput{
        QueueUrl: resultURL.QueueUrl,
        Attributes: aws.StringMap(map[string]string{
            "ReceiveMessageWaitTimeSeconds": strconv.Itoa(timeout),
        }),
    })
    if err != nil {
        exitErrorf("Unable to update queue %q, %v.", name, err)
    }

    fmt.Printf("Succesffuly updated queue %q.\n", name)
}

func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Enabling Long Polling on Message Receipt

Create a new Go file named `sqs_longpolling_receive_message.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main
```

```
import (
    "flag"
    "fmt"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sqs"
)
```

This example takes two flags, the -n flag is the queue name, and the -t flag contains the timeout value.

```
func main() {
    var name string
    var timeout int64
    flag.StringVar(&name, "n", "", "Queue name")
    flag.Int64Var(&timeout, "t", 20, "(Optional) Timeout in seconds for long polling")
    flag.Parse()

    if len(name) == 0 {
        flag.PrintDefaults()
        exitErrorf("Queue name required")
    }
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a SQS service client.
    svc := sqs.New(sess)
```

You need to convert the queue name into a URL. Make the `GetQueueUrl` API call to retrieve the URL. This is needed for setting attributes on the queue.

```
    resultURL, err := svc.GetQueueUrl(&sqs.GetQueueUrlInput{
        QueueName: aws.String(name),
    })
    if err != nil {
        if aerr, ok := err.(awserr.Error); ok && aerr.Code() ==
 sqs.ErrCodeQueueDoesNotExist {
            exitErrorf("Unable to find queue %q.", name)
        }
        exitErrorf("Unable to queue %q, %v.", name, err)
    }
```

Receive a message from the queue with long polling enabled with a call to `ReceiveMessage`, passing in the queue URL. Print any errors or a success message.

```
    result, err := svc.ReceiveMessage(&sqs.ReceiveMessageInput{
        QueueUrl: resultURL.QueueUrl,
        AttributeNames: aws.StringSlice([]string{
            "SentTimestamp",
        }),
        MaxNumberOfMessages: aws.Int64(1),
        MessageAttributeNames: aws.StringSlice([]string{
            "All",
        }),
```

```
        WaitTimeSeconds: aws.Int64(timeout),
    })
    if err != nil {
        exitErrorf("Unable to receive message from queue %q, %v.", name, err)
    }

    fmt.Printf("Received %d messages.\n", len(result.Messages))
    if len(result.Messages) > 0 {
        fmt.Println(result.Messages)
    }
}

func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

Using Dead Letter Queues in Amazon SQS with Go

This Go example shows you how to:

- Configure source Amazon SQS queues that send messages to a dead letter queue

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

A dead letter queue is one that other (source) queues can target for messages that can't be processed successfully. You can set aside and isolate these messages in the dead letter queue to determine why their processing didn't succeed. You must individually configure each source queue that sends messages to a dead letter queue. Multiple queues can target a single dead letter queue.

The code uses this method of the Amazon SQS client class:

- SetQueueAttributes

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with Amazon SQS dead letter queues. To learn more, see Using Amazon SQS Dead Letter Queues in the *Amazon SQS Developer Guide*.

Configuring Source Queues

After you create a queue to act as a dead letter queue, you must configure the other queues that route unprocessed messages to the dead letter queue. To do this, specify a redrive policy that identifies the queue to use as a dead letter queue and the maximum number of receives by individual messages before they are routed to the dead letter queue.

Create a new Go file with the name `sqs_deadletterqueue.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main
```

```
import (
    "encoding/json"
    "fmt"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sqs"
)
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config.

```
func main() {
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a SQS service client.
    svc := sqs.New(sess)
```

Define the redrive policy for the queue, then marshal the policy to use as input for the `SetQueueAttributes` call.

```
    policy := map[string]string{
        "deadLetterTargetArn": "SQS_QUEUE_ARN",
        "maxReceiveCount":     "10",
    }

    b, err := json.Marshal(policy)
    if err != nil {
        fmt.Println("Failed to marshal policy:", err)
        return
    }
```

Set the policy on the queue.

```
    result, err := svc.SetQueueAttributes(&sqs.SetQueueAttributesInput{
        QueueUrl: aws.String("SQS_QUEUE_URL"),
        Attributes: map[string]*string{
            sqs.QueueAttributeNameRedrivePolicy: aws.String(string(b)),
        },
    })

    if err != nil {
        fmt.Println("Error", err)
        return
    }

    fmt.Println("Success", result)
}
```

Setting Attributes on an Amazon SQS Queue with Go

This Go example shows you how to:

- Set attributes on an existing Amazon SQS queue

You can download complete versions of these example files from the aws-doc-sdk-examples repository on GitHub.

The Scenario

This example updates an existing Amazon SQS queue to use long polling.

Long polling reduces the number of empty responses by allowing Amazon SQS to wait a specified time for a message to become available in the queue before sending a response. Also, long polling eliminates false empty responses by querying all of the servers instead of a sampling of servers. To enable long polling, you must specify a non-zero wait time for received messages. You can do this by setting the *ReceiveMessageWaitTimeSeconds* parameter of a queue or by setting the *WaitTimeSeconds* parameter on a message when it is received.

The code uses these methods of the Amazon SQS client class:

- GetQueueUrl
- SetQueueAttributes

If you are unfamiliar with using Amazon SQS long polling, you should read Long Polling in the *Amazon SQS Developer Guide* before proceeding.

Prerequisites

- You have set up (p. 2) and configured (p. 4) the AWS SDK for Go.
- You are familiar with using Amazon SQS long polling. To learn more, see Long Polling in the *Amazon SQS Developer Guide*.

Set Attributes on Queue

Create a new Go file named `sqs_longpolling_existing_queue.go`.

You must import the relevant Go and AWS SDK for Go packages by adding the following lines.

```
package main

import (
    "flag"
    "fmt"
    "os"
    "strconv"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/awserr"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sqs"
)
```

Get the queue name and timeout passed in by the user.

```
func main() {
    var name string
    var timeout int
    flag.StringVar(&name, "n", "", "Queue name")
    flag.IntVar(&timeout, "t", 20, "(Optional) Timeout in seconds for long polling")
    flag.Parse()

    if len(name) == 0 {
        flag.PrintDefaults()
        exitErrorf("Queue name required")
```

```
    }
```

Initialize a session that the SDK will use to load configuration, credentials, and region information from the shared config file, ~/.aws/config.

```
    sess := session.Must(session.NewSessionWithOptions(session.Options{
        SharedConfigState: session.SharedConfigEnable,
    }))

    // Create a SQS service client.
    svc := sqs.New(sess)
```

Get the queue. You need to convert the queue name into a URL. You can use the `GetQueueUrl` API call to retrieve the URL. This is needed for setting attributes on the queue. Print any errors.

```
    resultURL, err := svc.GetQueueUrl(&sqs.GetQueueUrlInput{
        QueueName: aws.String(name),
    })
    if err != nil {
        if aerr, ok := err.(awserr.Error); ok && aerr.Code() ==
 sqs.ErrCodeQueueDoesNotExist {
            exitErrorf("Unable to find queue %q.", name)
        }
        exitErrorf("Unable to get queue %q, %v.", name, err)
    }
```

Update the queue to enable long polling.

```
    _, err = svc.SetQueueAttributes(&sqs.SetQueueAttributesInput{
        QueueUrl: resultURL.QueueUrl,
        Attributes: aws.StringMap(map[string]string{
            "ReceiveMessageWaitTimeSeconds": strconv.Itoa(timeout),
        }),
    })
    if err != nil {
        exitErrorf("Unable to update queue %q, %v.", name, err)
    }

    fmt.Printf("Succesffuly updated queue %q.\n", name)
}
```

The example uses this utility function.

```
func exitErrorf(msg string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, msg+"\n", args...)
    os.Exit(1)
}
```

SDK Utilities

The AWS SDK for Go includes the following utilities to help you more easily use AWS services. SDK utilities are located in their related AWS service package.

Amazon CloudFront URL Signer

The Amazon CloudFront URL signer simplifies the process of creating signed URLs. A signed URL includes information, such as an expiration date and time, that enables you to control access to your content. Signed URLs are useful when you want to distribute content through the Internet, but want to restrict access to certain users (for example, to users who have paid a fee).

To sign a URL, create a `URLSigner` instance with your Amazon CloudFront key pair ID and the associated private key, and then call the `Sign` or `SignWithPolicy` method and include the URL to sign. For more information about Amazon CloudFront key pairs, see Creating CloudFront Key Pairs for Your Trusted Signers in the *CloudFront Developer Guide*.

The following example creates a signed URL that's valid for one hour after it has been created:

```
signer := sign.NewURLSigner(keyID, privKey)

signedURL, err := signer.Sign(rawURL, time.Now().Add(1*time.Hour))
if err != nil {
    log.Fatalf("Failed to sign url, err: %s\n", err.Error())
    return
}
```

For more information about the signing utility, see the sign package in the AWS SDK for Go API Reference.

Amazon DynamoDB Attributes Converter

The attributes converter simplifies converting Amazon DynamoDB attribute values to and from concrete Go types. Conversions make it easy to work with attribute values in Go and to write values to Amazon DynamoDB tables. For example, you can create records in Go and then use the converter when you want to write those records as attribute values to an Amazon DynamoDB table.

The following example converts a structure to an Amazon DynamoDB`AttributeValues` map and then puts the data to the `exampleTable`:

```
type Record struct {
    MyField string
    Letters []string
    A2Num   map[string]int
}
r := Record{
    MyField: "dynamodbattribute.ConvertToX example",
    Letters: []string{"a", "b", "c", "d"},
    A2Num:   map[string]int{"a": 1, "b": 2, "c": 3},
}

//...

svc := dynamodb.New(session.New(&aws.Config{Region: aws.String("us-west-2")}))
item, err := dynamodbattribute.ConvertToMap(r)
if err != nil {
    fmt.Println("Failed to convert", err)
    return
}
result, err := svc.PutItem(&dynamodb.PutItemInput{
    Item:      item,
    TableName: aws.String("exampleTable"),
})
fmt.Println("Item put to dynamodb", result, err)
```

For more information about the converter utility, see the dynamodbattribute package in the AWS SDK for Go API Reference.

Amazon Elastic Compute Cloud Metadata

`EC2Metadata` is a client that interacts with the Amazon EC2 metadata service. The client can help you easily retrieve information about instances on which your applications run, such as its region or local IP address. Typically, you must create and submit HTTP requests to retrieve instance metadata; instead, create an `EC2Metadata` service client:

```
c := ec2metadata.New(session.New())
```

Then, use the service client to retrieve information from a metadata category like `local-ipv4` (the private IP address of the instance):

```
localip, err := c.GetMetadata("local-ipv4")
if err != nil {
    log.Printf("Unable to retrieve the private IP address from the EC2 instance: %s\n",
 err)
    return
}
```

For a list of all metadata categories, see Instance Metadata Categories in the *Amazon EC2 User Guide for Linux Instances*.

Retrieving an Instance's Region

There's no instance metadata category that returns only the region of an instance. Instead, use the included `Region` method to easily return an instance's region:

```
region, err := ec2metadata.New(session.New()).Region()
if err != nil {
    log.Printf("Unable to retrieve the region from the EC2 instance %v\n", err)
```

```
}
```

For more information about the EC2 metadata utility, see the ec2metadata package in the AWS SDK for Go API Reference.

Amazon Simple Storage Service Transfer Managers

The Amazon Simple Storage Service upload and download managers can break up large objects so they can be transferred in multiple parts, in parallel, which makes it easy to resume interrupted transfers.

Upload Manager

The Amazon Simple Storage Service upload manager determines if a file can be split into smaller parts and uploaded in parallel. You can customize the number of parallel uploads and the size of the uploaded parts.

Uploading

The following example uses the Amazon Simple Storage Service`Uploader` to upload a file. Using `Uploader` is similar to the `s3.PutObject()` operation.

```
mySession, _ := session.NewSession()
uploader := s3manager.NewUploader(mySession)
result, err := uploader.Upload(&s3manager.UploadInput{
    Bucket: &uploadBucket,
    Key:    &uploadFileKey,
    Body:   uploadFile,
})
```

Configuration Options

When you instantiate an `Uploader` instance, you can specify several configuration options (`UploadOptions`) to customize how objects are uploaded:

- `PartSize` specifies the buffer size, in bytes, of each part to be uploaded. The minimum size per part is 5 MB.
- `Concurrency` specifies the number of parts to upload in parallel.
- `LeavePartsOnError` indicates whether to leave successfully uploaded parts in Amazon Simple Storage Service.

Tweak the `PartSize` and `Concurrency` configuration values to find the optimal configuration. For example, systems with high-bandwidth connections can send bigger parts and more uploads in parallel.

For more information about `Uploader` and its configurations, see the s3manager package in the AWS SDK for Go API Reference.

UploadInput Body Field (io.ReadSeeker vs. io.Reader)

The `Body` field of the `s3manager.UploadInput` struct is an `io.Reader` type; however, the field also satisfies the `io.ReadSeeker` interface.

For `io.ReadSeeker` types, the `Uploader` doesn't buffer the body contents before sending it to Amazon Simple Storage Service. `Uploader` calculates the expected number of parts before uploading the file to Amazon Simple Storage Service. If the current value of `PartSize` requires more than 10,000 parts to upload the file, `Uploader` increases the part size value so that fewer parts are required.

For `io.Reader` types, the bytes of the reader must buffer each part in memory before the part is uploaded. When you increase the `PartSize` or `Concurrency` values, the required memory (RAM) for the `Uploader` increases significantly. The required memory is approximately ``*PartSize*`` * ``*Concurrency*``. For example, if you specify 100 MB for `PartSize` and 10 for `Concurrency`, the required memory will be at least 1 GB.

Because an `io.Reader` type cannot determine its size before reading its bytes, `Uploader` cannot calculate how many parts must be uploaded. Consequently, `Uploader` can reach the Amazon Simple Storage Service upload limit of 10,000 parts for large files if you set the `PartSize` too low. If you try to upload more than 10,000 parts, the upload stops and returns an error.

Handling Partial Uploads

If an upload to Amazon Simple Storage Service fails, by default, `Uploader` uses the Amazon Simple Storage Service`AbortMultipartUpload` operation to remove the uploaded parts. This functionality ensures that failed uploads do not consume Amazon Simple Storage Service storage.

You can set `LeavePartsOnError` to true so that the `Uploader` doesn't delete successfully uploaded parts, which is useful for resuming partially completed uploads. To operate on uploaded parts, you must get the `UploadID` of the failed upload. The following example demonstrates how to use the `s3manager.MultiUploadFailure` message to get the `UploadID`:

```
u := s3manager.NewUploader(session.New())
output, err := u.upload(input)
if err != nil {
    if multierr, ok := err.(s3manager.MultiUploadFailure); ok {
        // Process error and its associated uploadID
        fmt.Println("Error:", multierr.Code(), multierr.Message(), multierr.UploadID())
    } else {
        // Process error generically
        fmt.Println("Error:", err.Error())
    }
}
```

Example: Upload Folder to Amazon Simple Storage Service

The following examples use the `path/filepath` package to recursively gather a list of files and upload them to the specified Amazon Simple Storage Service bucket. The keys of the Amazon Simple Storage Service objects are prefixed with the file's relative path.

```
package main

import (
    "log"
    "os"
    "path/filepath"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3/s3manager"
)

var (
    localPath string
    bucket    string
    prefix    string
)

func init() {
    if len(os.Args) != 4 {
```

```
        log.Fatalln("Usage:", os.Args[0], "<local path> <bucket> <prefix>")
    }
    localPath = os.Args[1]
    bucket = os.Args[2]
    prefix = os.Args[3]
}

func main() {
    walker := make(fileWalk)
    go func() {
        // Gather the files to upload by walking the path recursively.
        if err := filepath.Walk(localPath, walker.Walk); err != nil {
            log.Fatalln("Walk failed:", err)
        }
        close(walker)
    }()

    // For each file found walking upload it to S3.
    uploader := s3manager.NewUploader(session.New())
    for path := range walker {
        rel, err := filepath.Rel(localPath, path)
        if err != nil {
            log.Fatalln("Unable to get relative path:", path, err)
        }
        file, err := os.Open(path)
        if err != nil {
            log.Println("Failed opening file", path, err)
            continue
        }
        defer file.Close()
        result, err := uploader.Upload(&s3manager.UploadInput{
            Bucket: &bucket,
            Key:    aws.String(filepath.Join(prefix, rel)),
            Body:   file,
        })
        if err != nil {
            log.Fatalln("Failed to upload", path, err)
        }
        log.Println("Uploaded", path, result.Location)
    }
}

type fileWalk chan string

func (f fileWalk) Walk(path string, info os.FileInfo, err error) error {
    if err != nil {
        return err
    }
    if !info.IsDir() {
        f <- path
    }
    return nil
}
```

Example: Upload File to Amazon Simple Storage Service and Send Location to Amazon Simple Queue Service

The following example uploads a file to an Amazon Simple Storage Service bucket and then sends a notification message of the file's location to an Amazon Simple Queue Service queue:

```
package main

import (
```

```
    "log"
    "os"

    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3/s3manager"
    "github.com/aws/aws-sdk-go/service/sqs"
)

// Uploads a file to a specific bucket in S3 with the filename
// as the Object's key. After it's uploaded a message will be sent
// to a queue.
func main() {
    if len(os.Args) != 4 {
        log.Fatalln("Usage:", os.Args[0], "<bucket> <queue> <file>")
    }

    file, err := os.Open(os.Args[3])
    if err != nil {
        log.Fatal("Open failed:", err)
    }
    defer file.Close()

    // Upload the file to S3 using the S3 Manager
    uploader := s3manager.NewUploader(session.New())
    uploadRes, err := uploader.Upload(&s3manager.UploadInput{
        Bucket: aws.String(os.Args[1]),
        Key:    aws.String(file.Name()),
        Body:   file,
    })
    if err != nil {
        log.Fatalln("Upload failed:", err)
    }

    // Get the Queue's URL that the message will be posted to
    svc := sqs.New(session.New())
    urlRes, err := svc.GetQueueUrl(&sqs.GetQueueUrlInput{
        QueueName: aws.String(os.Args[2]),
    })
    if err != nil {
        log.Fatalln("GetQueueURL failed:", err)
    }

    // Send the Message to the Queue
    _, err = svc.SendMessage(&sqs.SendMessageInput{
        MessageBody: &uploadRes.Location,
        QueueUrl:    urlRes.QueueUrl,
    })
    if err != nil {
        log.Fatalln("SendMessage failed:", err)
    }
}
```

Download Manager

The Amazon Simple Storage Service download manager determines if a file can be split into smaller parts and downloaded in parallel. You can customize the number of parallel downloads and the size of the downloaded parts.

Downloading

The following example uses the Amazon Simple Storage Service`Downloader` to download a file. Using `Downloader` is similar to the `s3.GetObject()` operation.

```
downloader := s3manager.NewDownloader(session.New())
numBytes, err := downloader.Download(downloadFile,
  &s3.GetObjectInput{
    Bucket: &downloadBucket,
    Key:    &downloadFileKey,
})
```

The `downloadFile` parameter is an `io.WriterAt` type. The `WriterAt` interface enables the `Downloader` to write multiple parts of the file in parallel.

Configuration Options

When you instantiate a `Downloader` instance, you can specify several configuration options (`DownloadOptions`) to customize how objects are downloaded:

- `PartSize` specifies the buffer size, in bytes, of each part to be downloaded. The minimum size per part is 5 MB.
- `Concurrency` specifies the number of parts to download in parallel.

Tweak the `PartSize` and `Concurrency` configuration values to find the optimal configuration. For example, systems with high-bandwidth connections can receive bigger parts and more downloads in parallel.

For more information about `Downloader` and its configurations, see the s3manager package in the AWS SDK for Go API Reference.

Example: Download All Objects in a Bucket

The following example uses pagination to gather a list of objects from an Amazon Simple Storage Service bucket and then downloads each object to a local file:

```
package main

import (
    "fmt"
    "os"
    "path/filepath"

    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
    "github.com/aws/aws-sdk-go/service/s3/s3manager"
)

var (
    Bucket         = "MyBucket" // Download from this bucket
    Prefix         = "logs/"    // Using this key prefix
    LocalDirectory = "s3logs"   // Into this directory
)

func main() {
    manager := s3manager.NewDownloader(session.New())
    d := downloader{bucket: Bucket, dir: LocalDirectory, Downloader: manager}

    client := s3.New(session.New())
    params := &s3.ListObjectsInput{Bucket: &Bucket, Prefix: &Prefix}
    client.ListObjectsPages(params, d.eachPage)
}

type downloader struct {
```

```
    *s3manager.Downloader
    bucket, dir string
}

func (d *downloader) eachPage(page *s3.ListObjectsOutput, more bool) bool {
    for _, obj := range page.Contents {
        d.downloadToFile(*obj.Key)
    }

    return true
}

func (d *downloader) downloadToFile(key string) {
    // Create the directories in the path
    file := filepath.Join(d.dir, key)
    if err := os.MkdirAll(filepath.Dir(file), 0775); err != nil {
        panic(err)
    }

    // Set up the local file
    fd, err := os.Create(file)
    if err != nil {
        panic(err)
    }
    defer fd.Close()

    // Download the file using the AWS SDK
    fmt.Printf("Downloading s3://%s/%s to %s...\n", d.bucket, key, file)
    params := &s3.GetObjectInput{Bucket: &d.bucket, Key: &key}
    d.Download(fd, params)
}
```