Developer Guide

AWS SDK for Swift

AWS SDK for Swift: Developer Guide

Table of Contents

AWS SDK for Swift Developer Guide

What is the AWS SDK for Swift?

Welcome to the AWS SDK for Swift, a pure Swift SDK that makes it easier to develop tools that take advantage of AWS services, including Amazon S3, Amazon EC2, DynamoDB, and more, all using the Swift programming language.

Supported targets

Platform	Operating system(s)	Processor	Notes
Apple (Darwin)	macOS, iOS/iPadOS, tvOS, watchOS, and visionOS.	Intel x86 Apple Silicon (ARM).	All Apple targets support 64-bit only. The only exception is watchOS, which also supports the hybrid 64/32-bit processors used in the Series 1 through Series 3 Apple watches.
Linux	Amazon Linux 2 and Ubuntu.	Intel x86.	The SDK may build and function on ARM distributions, but this is not currently supported by Amazon Web Services.
AWS Lambda	Amazon Linux 2	Intel x86.	

The AWS SDK for Swift is not currently built, tested, or supported on Microsoft Windows. Support for Windows targets could be added in the future if customer demand warrants it.

Get started with the SDK

To set up your development environment, see ???. Then you can test drive the AWS SDK for Swift by creating your first project in ???.

For information on making requests to Amazon S3, DynamoDB, Amazon EC2, and other AWS services, see *Use the SDK*.

About this guide

The AWS SDK for Swift Developer Guide covers how to install, configure, and use the preview release of the SDK to create applications and tools in Swift that make use of AWS services.

This guide contains the following sections:

Set up

Covers installing the Swift toolchain if you don't already have it installed and configuring your system for use with the SDK and to have access to your AWS account for development and testing purposes.

Get started

Explains how to create a project that imports the SDK for Swift using the Swift Package Manager in a shell environment on Linux and macOS, as well as how to add the SDK package to an Xcode project. Also included is a guide to building a project that uses the SDK to output a list of a user's Amazon S3 buckets.

Use the SDK

Guides covering typical usage scenarios including creating service clients, performing common tasks, and more.

Apple integration

Guidance and best practices for using the SDK to create applications that take advantage of Apple platform features.

Code examples

Code examples demonstrating how to use various features of the SDK for Swift, as well as how to achieve specific tasks using the SDK.

Security

Guides covering security topics in general, as well as considerations surrounding using the SDK for Swift in various contexts and while performing specific tasks.

Document history

History of this document and of the SDK.

Maintenance and support for SDK major versions

For information about maintenance and support for SDK major versions and their underlying dependencies, see the following in the AWS SDKs and Tools Reference Guide.

- AWS SDKs and Tools Maintenance Policy
- AWS SDKs and Tools Version Support Matrix

Additional resources

In addition to this guide, the following are valuable online resources for AWS SDK for Swift developers:

- AWS SDK for Swift API Reference
- AWS SDK for Swift code examples
- AWS SDK for Swift on GitHub

Contributing to the SDK

Developers can also contribute feedback through the following channels:

- Report bugs in the AWS SDK for Swift

Set up the AWS SDK for Swift

The AWS SDK for Swift is a cross-platform, open source Swift package that lets applications and tools built using Swift make full use of the services offered by AWS. Using the SDK, you can build Swift applications that work with Amazon S3, Amazon EC2, DynamoDB, and more.

The SDK package is imported into your project using the Swift Package Manager (SPM), which is part of the standard Swift toolchain.

When you've finished following the steps in this article, or have confirmed that everything is configured as described, you're ready to begin developing using the AWS SDK for Swift.

Overview

To make requests to AWS using the AWS SDK for Swift, you need the following:

- An active AWS account.
- A user in IAM Identity Center with permission to use the AWS services and resources your application will access.
- A development environment with version 5.9 or later of the Swift toolchain. If you don't have this, you'll install it as part of the steps below.
- Xcode users need version 15 or later of the Xcode application. This includes Swift 5.9.

After finishing these steps, you're ready to use the SDK to develop Swift projects that access AWS services.

Set up AWS access

Before installing the Swift tools, configure your environment to let your project access AWS services. This section covers creating and configuring your AWS account, preparing IAM Identity Center for use, and setting the environment variables used by the SDK for Swift to fetch your access credentials.

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.
2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.

2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

Grant programmatic AWS account access

Users need programmatic access if they want to interact with AWS outside of the AWS Management Console. The way to grant programmatic access depends on the type of user that's accessing AWS.

To grant users programmatic access, choose one of the following options.

Which user needs programmatic access?	To	By
Workforce identity (Users managed in IAM Identity Center)	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. • For the AWS CLI, see Configuring the AWS CLI to use AWS IAM Identity Center in the *AWS Command Line Interface User Guide*. • For AWS SDKs, tools, and AWS APIs, see IAM Identity Center authentication in the *AWS SDKs and Tools Reference Guide*.
IAM	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions in Using temporary credentials with AWS resources in the *IAM User Guide*.
IAM	(Not recommended) Use long-term credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. • For the AWS CLI, see Authenticating using IAM user credentials in the *AWS Command Line Interface User Guide*. • For AWS SDKs and tools, see Authenticate using long-term credentials in

Which user needs programmatic access?	To	By
		the *AWS SDKs and Tools Reference Guide*. • For AWS APIs, see Managing access keys for IAM users in the *IAM User Guide*.

For more advanced cases regarding configuring the credentials and Region, see The .aws/credentials and .aws/config files, AWS Region, and Using environment variables in the AWS SDKs and Tools Reference Guide.

> **Note**
>
> If you plan to develop a macOS desktop application, keep in mind that due to sandbox restrictions, the SDK is unable to access your `~/.aws/config` and `~/.aws/credentials` files, as there is no entitlement available to grant access to the `~/.aws` directory. See the section called "Security and authentication when testing on macOS" for details.

Set up your Swift development environment

The SDK requires at least version 5.9 of Swift. This can be installed either standalone or as part of the Xcode development environment on macOS.

- Swift 5.9 toolchain or newer.
- If you're developing on macOS using Xcode, you need a minimum of Xcode 15.
- An AWS account. If you don't have one already, you can create one using the AWS portal.

Prepare to install Swift

The Swift install process for Linux doesn't automatically install `libcrypto` version 1.1, even though it's required by the compiler. To install it, be sure to install OpenSSL 1.1 or later, as well as its development package. Using the `yum` package manager, for example:

```
$ sudo yum install openssl openssl-devel
```

Using the `apt` package manager:

```
$ sudo apt install openssl libssl-dev
```

This isn't necessary on macOS.

Install Swift

On macOS, the easiest way to install Swift is to simply install Apple's Xcode IDE, which includes Swift and all the standard libraries and tools that go with it. This can be found on the macOS App Store.

If you're using Linux or Windows, or don't want to install Xcode on macOS, the Swift organization's web site has detailed instructions to install and set up the Swift toolchain.

Check the Swift tools version number

If Swift is already installed, you can verify the version number using the command `swift --version`. The output will look similar to one of the following examples.

Checking the Swift version on macOS

```
$ swift --version
swift-driver version: 1.87.1 Apple Swift version 5.9 (swiftlang-5.9.0.128.108
 clang-1500.0.40.1)
Target: x86_64-apple-macosx14.0
```

Checking the Swift version on Linux

```
$ swift --version
Swift version 5.8.1 (swift-5.8.1-RELEASE)
Target: x86_64-unknown-linux-gnu
```

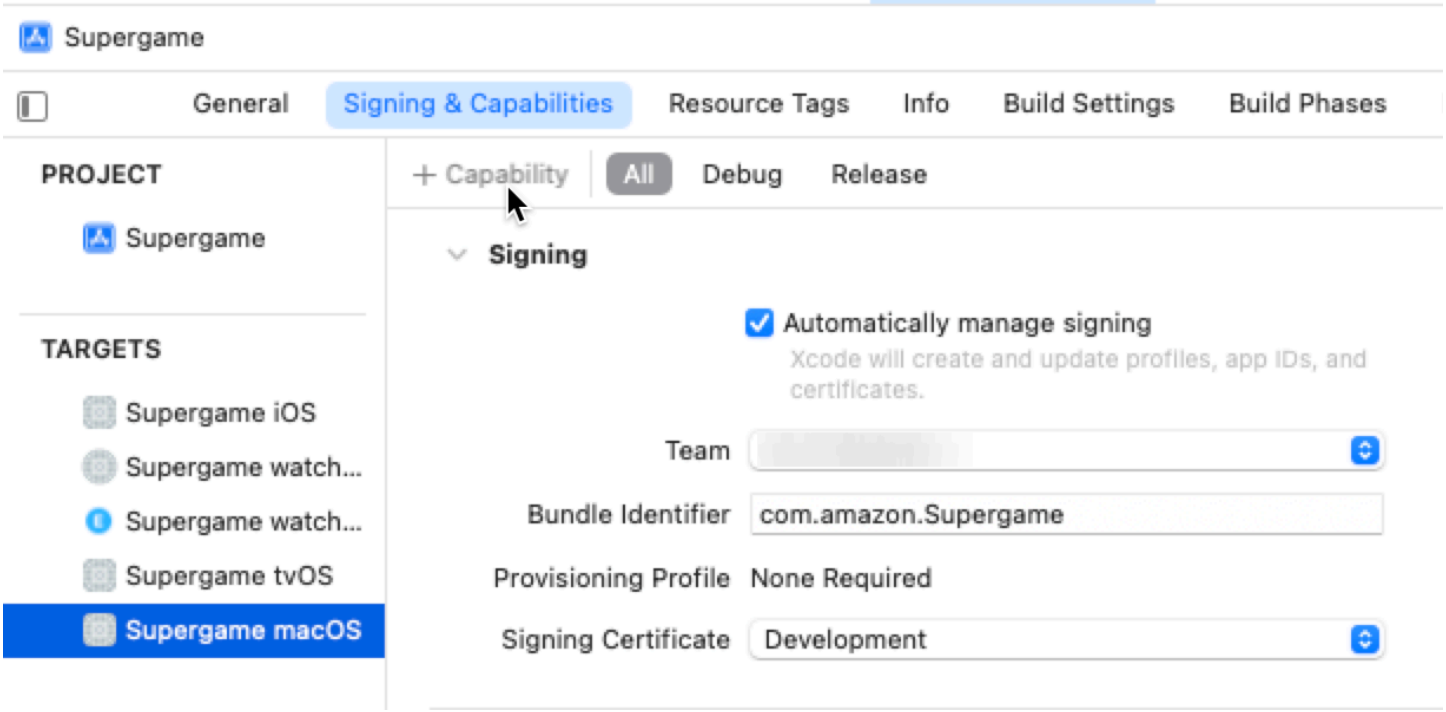
For more information about the configuration and credentials files shared among the AWS Command Line Interface and the various AWS SDKs, see the AWS SDKs and Tools Reference Guide.

Security and authentication when testing on macOS

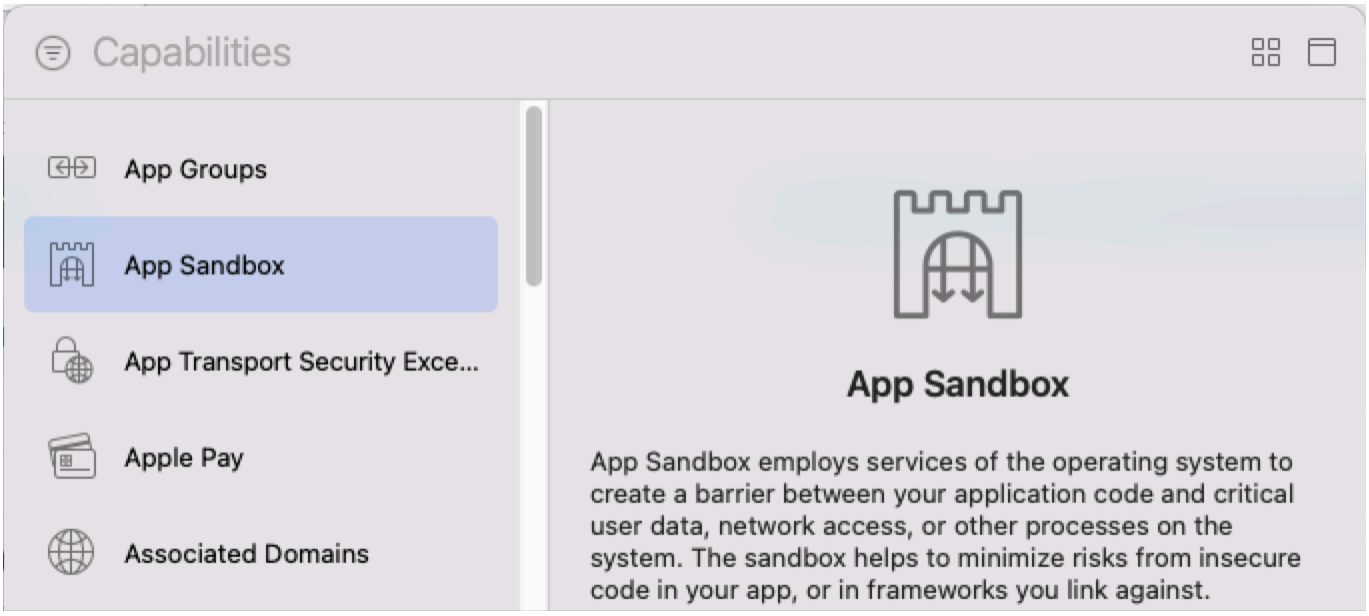
Configuring the App Sandbox

If your SDK for Swift project is a desktop application that you're building in Xcode, you will need to enable the App Sandbox capability and turn on the "Outgoing Connections (Client)" entitlement so that the SDK can communicate with AWS.

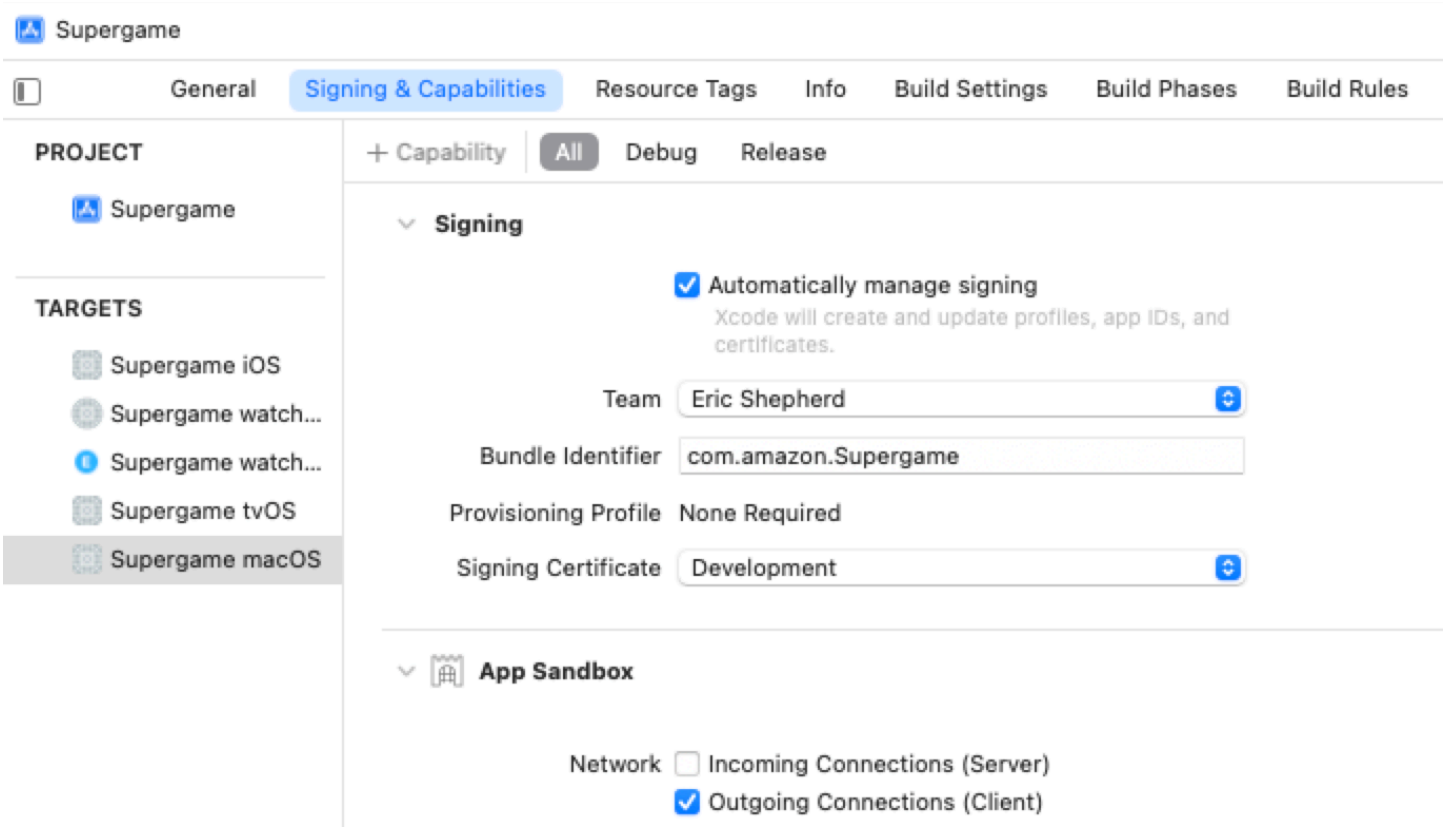
First, open the macOS target's **Signing & Capabilities** panel, shown below.



Click the **+ Capability** button near the top left of this panel to bring up the box listing the available capabilities. In this box, locate the "App Sandbox" capability and double-click on it to add it to your target.



Next, back in your target's **Signing & Capabilities** panel, find the new **App Sandbox** section and make sure that next to **Network**, the **Outgoing Connections (Client)** checkbox is selected in the following image.



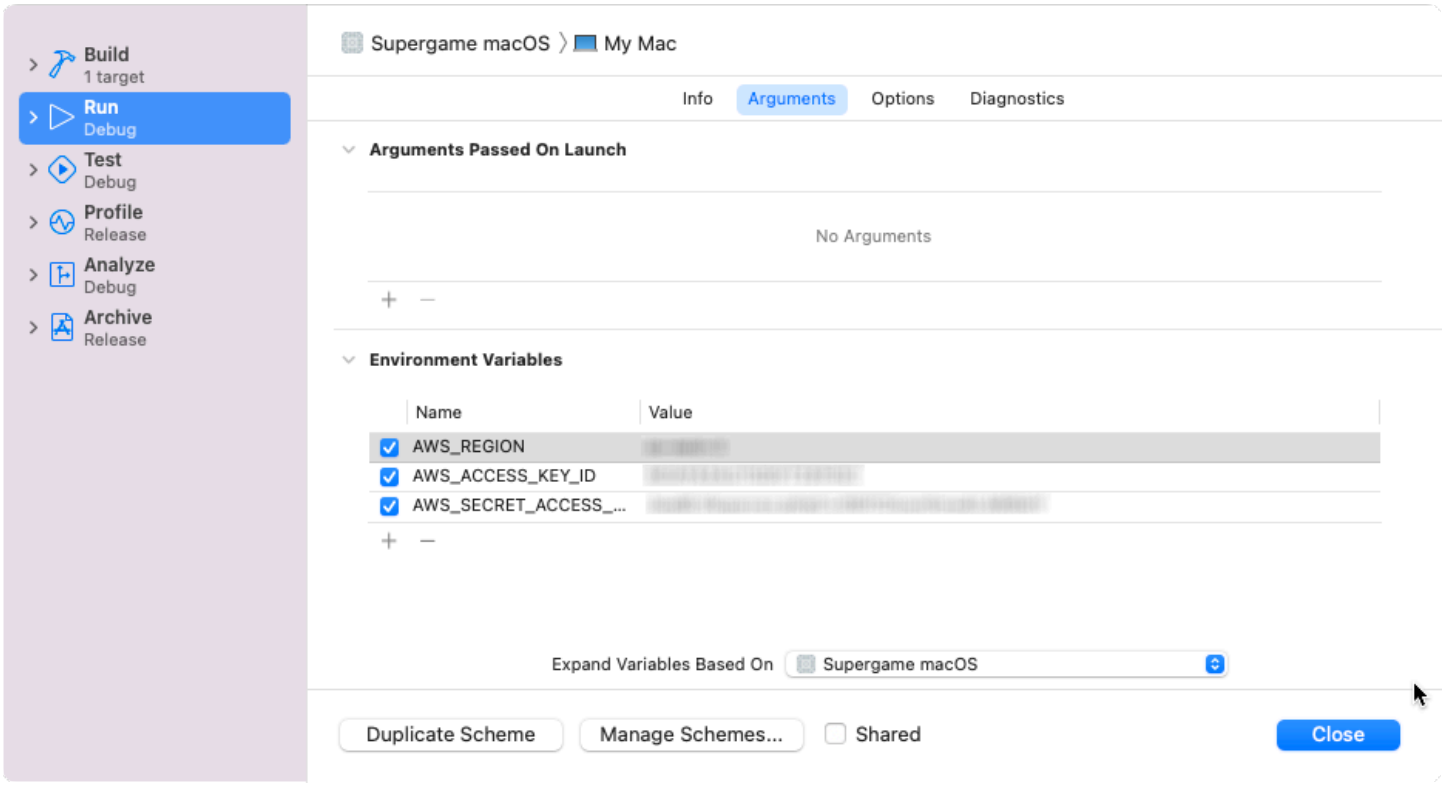
Using AWS access keys on macOS

Although shipping applications should use AWS IAM Identity Center, Amazon Cognito Identity, or similar technologies to handle authentication instead of requiring the direct use of AWS access keys, you may need to use these low-level credentials during development and testing.

macOS security features for desktop applications don't allow applications to access files without express user permission, so the SDK can't automatically configure clients using the contents of the CLI's `~/.aws/config` and `~/.aws/credentials` files. Instead, you need to use the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables to specify the authentication keys for your AWS account.

When running projects from within Xcode, the environment you have set up for your shell is not automatically inherited. Because of this, if you want to test your code using your AWS account's access key ID and secret access key, you need to set up the runtime environment for your application in the Xcode scheme for your development device. To configure the environment for a particular target in your Xcode project, switch to that target, then choose **Edit Scheme** in Xcode's **Product** menu.

This will open the scheme editor window for your project. Click on the **Run** phase in the left sidebar, then **Arguments** in the tab bar near the top of the window.



Under **Environment Variables**, click the **+** icon to add `AWS_REGION` and set its value to the desired region (in the screenshot above, it's set to "us-east-2"). Then add `AWS_ACCESS_KEY_ID` and its value, then `AWS_SECRET_ACCESS_KEY` and its value. Close the window once you have these configured and your scheme's Run configuration looks similar to the above.

> **Important**
>
> Be sure to uncheck the **Shared** checkbox before pushing your code to any public version control repository such as GitHub. Otherwise, your AWS access key and secret access key will be *included* in the publicly shared content. This is important enough to be worth double-checking regularly.

Your project should now be able to use the SDK to connect to AWS services.

Next steps

Now that your tools and environment are ready for you to begin developing with AWS SDK for Swift, see *Get started*, which demonstrates how to create and build a Swift project using AWS services.

Get started with the AWS SDK for Swift

This chapter explores how to use the Swift Package Manager — part of the standard Swift toolchain — to create and build a small project. The project uses the AWS SDK for Swift to output a list of available Amazon Simple Storage Service (Amazon S3) buckets.

Create a project using the AWS SDK for Swift

This chapter demonstrates how to create a small program that lists all the Amazon S3 buckets available on the default user account.

Goals for this project:

- Create a project using Swift Package Manager.
- Add the AWS SDK for Swift to the project.
- Configure the project's `Package.swift` file to describe the project and its dependencies.
- Write code that uses Amazon S3 to get a list of the buckets on the default AWS account, then prints them to the screen.

Before we begin, make sure to prepare your development environment as described in Set up. To make sure you're set up properly, use the following command. This makes sure that Swift is available and which version it is.

```
$ swift --version
```

On macOS, you should see output that looks like the following (with possibly different version and build numbers):

```
swift-driver version: 1.87.1 Apple Swift version 5.9 (swiftlang-5.9.0.128.108
 clang-1500.0.40.1)
Target: x86_64-apple-macosx14.0
```

On Linux, the output should look something like the following:

```
Swift version 5.9.0 (swift-5.9.0-RELEASE)
```

```
Target: x86_64-unknown-linux-gnu
```

If Swift is not installed, or is older than version 5.9, follow the instructions in ??? to install or reinstall the tools.

> **Get this example on GitHub**
>
> You can fork or download this example from the AWS SDK for Swift code examples repository.

Create the project

With the Swift tools installed, open a terminal session using your favorite terminal application (such as Terminal, iTerm, or the integrated terminal in your editor).

At the terminal prompt, go to the directory where you want to create the project. Then, enter the following series of commands to create an empty Swift project for a standard executable program.

```
$ mkdir ListBuckets
$ cd ListBuckets
$ swift package init --type executable
$ mv Sources/main.swift Sources/entry.swift
```

This creates the directory for the examples project, moves into that directory, and initializes that directory with the Swift Package Manager. The result is the basic file system structure for a simple executable program. The Swift source code file `main.swift` is also renamed to `entry.swift` to work around a bug in the Swift tools that involves the use of asynchronous code in the main function of a source file named `main.swift`.

```
ListBuckets/
### Package.swift
### Sources/
   ### entry.swift
```

Open your created project in your preferred text editor or IDE. On macOS, you can open the project in Xcode with the following:

```
$ xed .
```

As another example, you can open the project in Visual Studio Code with the following:

```
$ code .
```

Configure the package

After opening the project in your editor, open the `Package.swift` file. This is a Swift file that defines an SPM `Package` object that describes the project, its dependencies, and its build rules.

The first line of every `Package.swift` file must be a comment specifying the minimum version of the Swift toolchain needed to build the project. This isn't only informational. The version specified here can change the behavior of the tools for compatibility purposes. The AWS SDK for Swift requires at least version 5.9 of the Swift tools.

```
// swift-tools-version:5.9
```

Specify supported platforms

For projects whose target operating systems include any Apple platform, add or update the `platforms` property to include a list of the supported Apple platforms and minimum required versions. This list only specifies support for Apple platforms and doesn't preclude building for other platforms.

```
    // Let Xcode know the minimum Apple platforms supported.
    platforms: [
        .macOS(.v11),
        .iOS(.v13)
    ],
```

In this excerpt, the supported Apple platforms are macOS (version 11.0 and higher) and iOS/iPadOS (version 13 and higher).

Set up dependencies

The package dependency list needs to include the AWS SDK for Swift. This tells the Swift compiler to fetch the SDK and its dependencies before attempting to build the project.

```
    dependencies: [
```

```
        // Dependencies declare other packages that this package depends on.
        .package(
            url: "https://github.com/awslabs/aws-sdk-swift",
            from: "1.0.0"
        )
    ],
```

Configure the target

Now that the package depends on the AWS SDK for Swift, add a dependency to the executable program's target. Indicate that it relies on Amazon S3, which is offered by the SDK's `AWSS3` product.

```
    targets: [
        // Targets are the basic building blocks of a package, defining a module or a
test suite.
        // Targets can depend on other targets in this package and products from
dependencies.
        .executableTarget(
            name: "ListBuckets-Simple",
            dependencies: [
                .product(name: "AWSS3", package: "aws-sdk-swift")
            ],
            path: "Sources")
    ]
```

Access AWS services using Swift

The example program's Swift code is found in the `Source/entry.swift` file. This file begins by importing the needed Swift modules, using the `import` statement.

```
import AWSClientRuntime
import AWSS3
import Foundation
```

- `Foundation` is the standard Apple Foundation package.
- `AWSS3` is the SDK module that's used to access Amazon S3.
- `AWSClientRuntime` contains runtime components used by the SDK.

After you add the SDK for Swift to your project and import the service you want to use into your source code, you can create an instance of the client representing the service and use it to issue AWS service requests.

Create a service client object

Each AWS service is represented by a specific client class in the AWS SDK for Swift. For example, while the Amazon DynamoDB client class is called `DynamoDBClient`, the class for Amazon Simple Storage Service is `S3Client`. To use Amazon S3 in this example, first create an `S3Client` object on which to call the SDK's Amazon S3 functions.

```
        let configuration = try await S3Client.S3ClientConfiguration()
        //   configuration.region = "us-east-2" // Uncomment this to set the region
programmatically.
        let client = S3Client(config: configuration)
```

Issue AWS service requests

To issue a request to an AWS service, call the corresponding function on the service's client object. Each function's inputs are specified using a function-specific input structure as the value of the function's `input` parameter. For example, when calling `S3Client.listBuckets(input:)`, `input` is a structure of type `ListBucketsInput`.

```
        // Use "Paginated" to get all the buckets.
        // This lets the SDK handle the 'continuationToken' in "ListBucketsOutput".
        let pages = client.listBucketsPaginated(
            input: input
        )
```

Functions defined by the AWS SDK for Swift run asynchronously, so the example uses `await` to block the program's execution until the result is available. If SDK functions encounter errors, they throw them so your code can handle them using a `do-catch` statement or by propagating them back to the caller.

Get all bucket names

This example's main program calls `getBucketNames()` to get an array containing all of the bucket names. That function is defined as follows.

```
// Return an array containing the names of all available buckets.
//
// - Returns: An array of strings listing the buckets.
func getBucketNames() async throws -> [String] {
    do {
        // Get an S3Client with which to access Amazon S3.
        let configuration = try await S3Client.S3ClientConfiguration()
        //   configuration.region = "us-east-2" // Uncomment this to set the region
 programmatically.
        let client = S3Client(config: configuration)

        // Use "Paginated" to get all the buckets.
        // This lets the SDK handle the 'continuationToken' in "ListBucketsOutput".
        let pages = client.listBucketsPaginated(
            input: ListBucketsInput( maxBuckets: 10)
        )

        // Get the bucket names.
        var bucketNames: [String] = []

        do {
            for try await page in pages {
                guard let buckets = page.buckets else {
                    print("Error: no buckets returned.")
                    continue
                }

                for bucket in buckets {
                    bucketNames.append(bucket.name ?? "<unknown>")
                }
            }

            return bucketNames
        } catch {
            print("ERROR: listBuckets:", dump(error))
            throw error
        }
    }
}
```

This function starts by creating an Amazon S3 client and calling its `listBuckets(input:)` function to request a list of all of the available buckets. The list is returned asynchronously. After

it's returned, the bucket list is fetched from the output structure's `buckets` property. If it's `nil`, an empty array is immediately returned to the caller. Otherwise, each bucket name is added to the array of bucket name strings, which is then returned to the caller.

Add the example entry point

To allow the use of asynchronous functions from within `main()`, use Swift's `@main` attribute to create an object that contains a static `async` function called `main()`. Swift will use this as the program entry point.

```
/// The program's asynchronous entry point.
@main
struct Main {
    static func main() async {
        do {
            let names = try await getBucketNames()

            print("Found \(names.count) buckets:")
            for name in names {
                print("  \(name)")
            }
        } catch let error as AWSServiceError {
            print("An Amazon S3 service error occurred: \(error.message ?? "No details
 available")")
        } catch {
            print("An unknown error occurred: \(dump(error))")
        }
    }
}
```

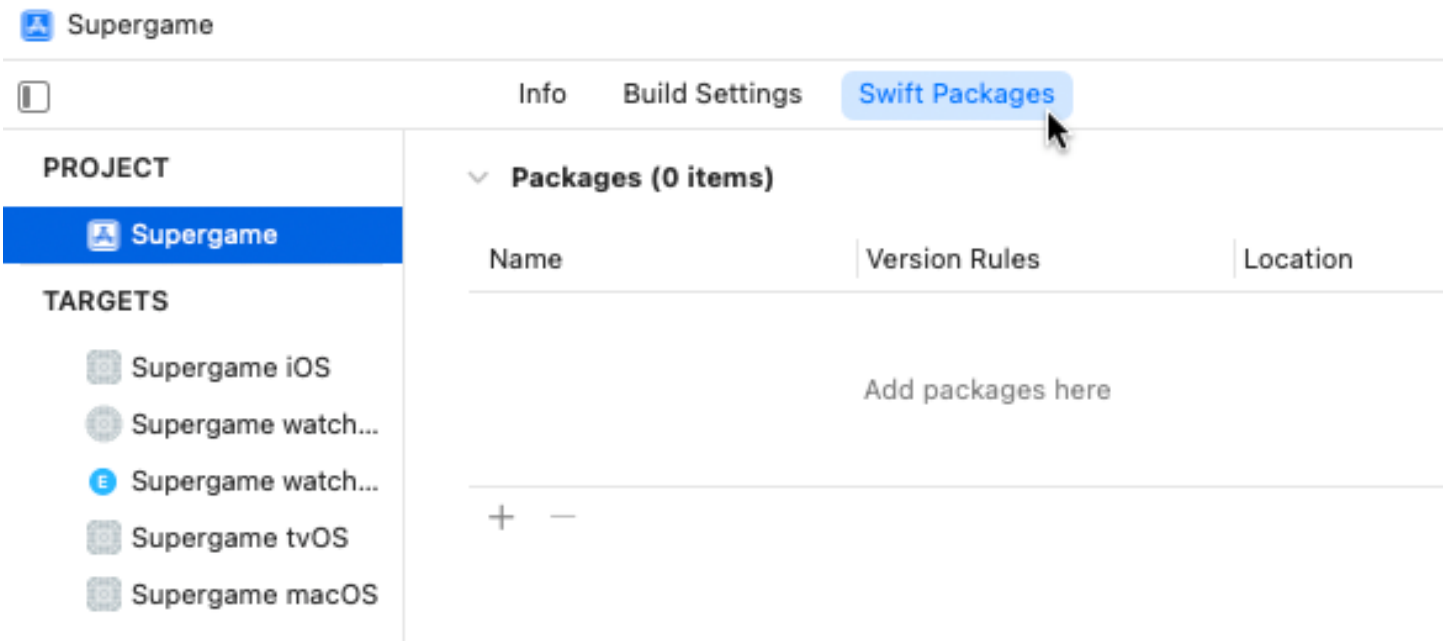
`main()` calls `getBucketNames()`, then outputs the returned list of names. Errors thrown by `getBucketNames()` are caught and handled. Errors of type `ServiceError`, which represent errors reported by the AWS service, are handled specially.

Add the AWS SDK for Swift to an existing Xcode project

If you have an existing Xcode project, you can add the SDK for Swift to it. Open your project's main configuration pane and choose the **Swift Packages** tab at the right end of the tab bar. The

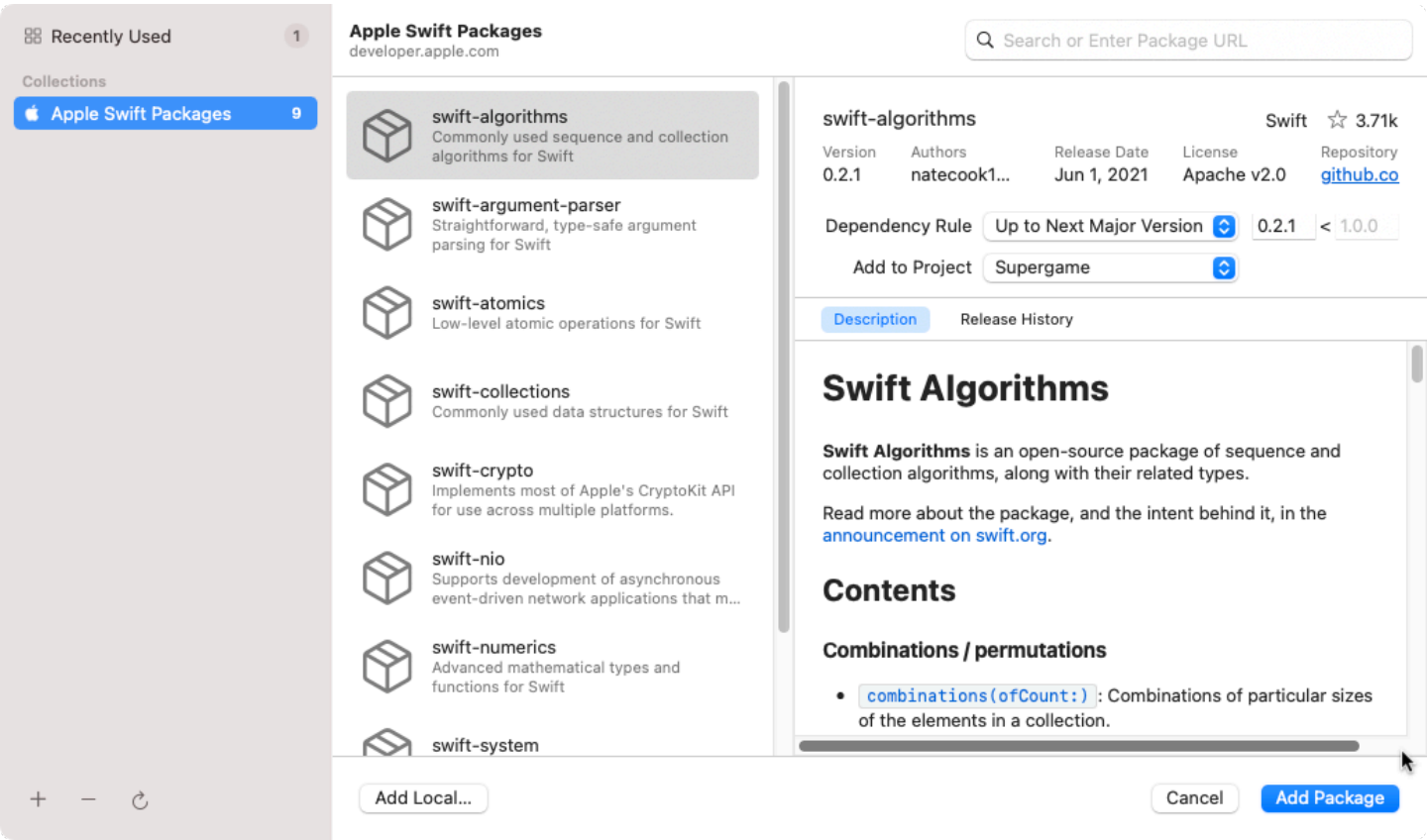
following image shows how to do this for an Xcode project called "Supergame," which will use Amazon S3 to get game data from a server.

Swift Packages tab in Xcode



This shows a list of the Swift packages currently in use by your project. If you haven't added any Swift packages, the list will be empty, as shown in the preceding image. To add the AWS SDK for Swift package to your project, choose the **+** button under the package list.

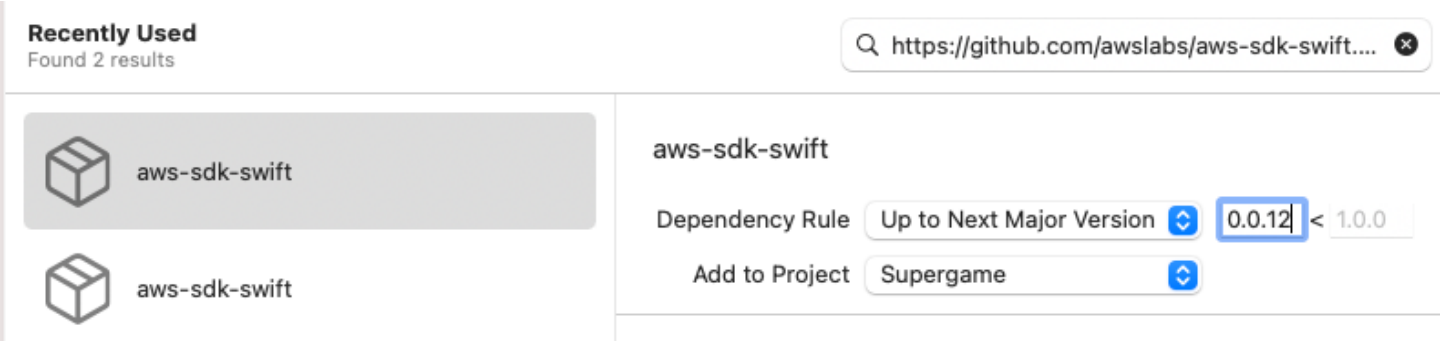
Find and select packages to import



Next, specify the package or packages to add to your project. You can choose from standard Apple-provided packages or enter the URL of a custom package in the search box at the top of the window. Enter the URL of the AWS SDK for Swift as follows: `https://github.com/awslabs/aws-sdk-swift.git` .

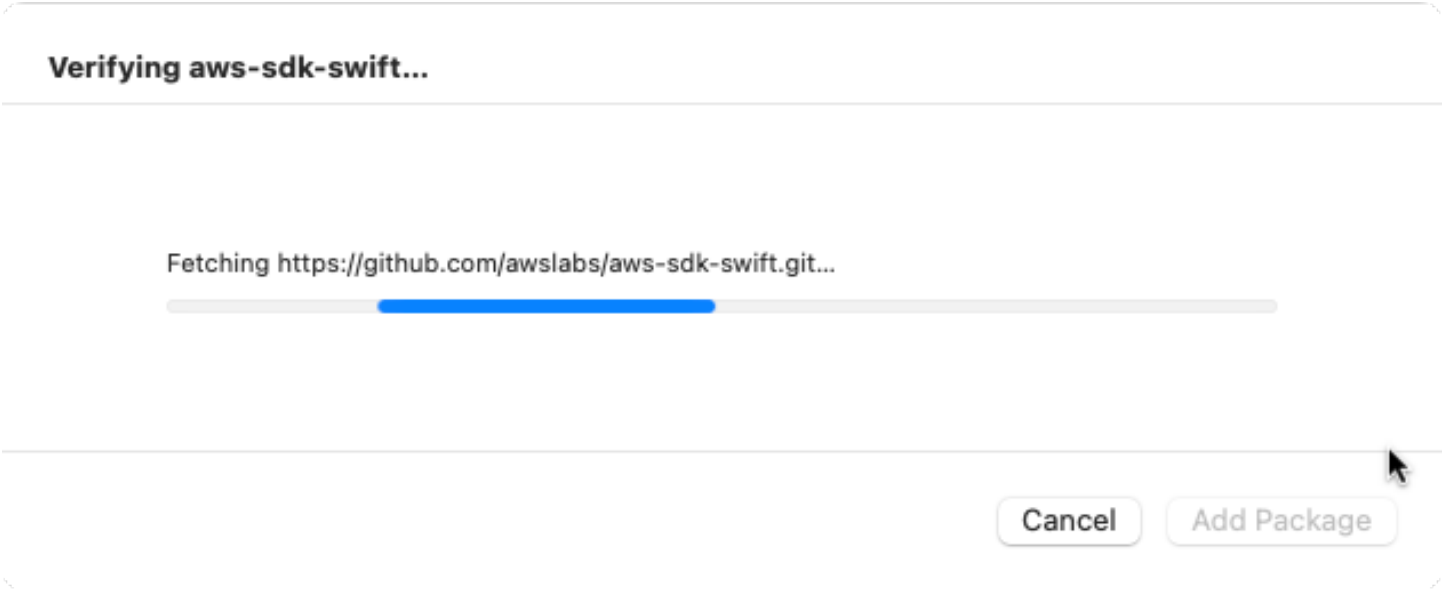
After you enter the SDK URL, you can configure version requirements and other options for the SDK package import.

Configure dependency rules for the SDK for Swift package



Configure the dependency rule. Make sure that **Add to Project** is set to your project — "Supergame" in this case — and choose **Add Package**. You will see a progress bar while the SDK and all its dependencies are processed and retrieved.

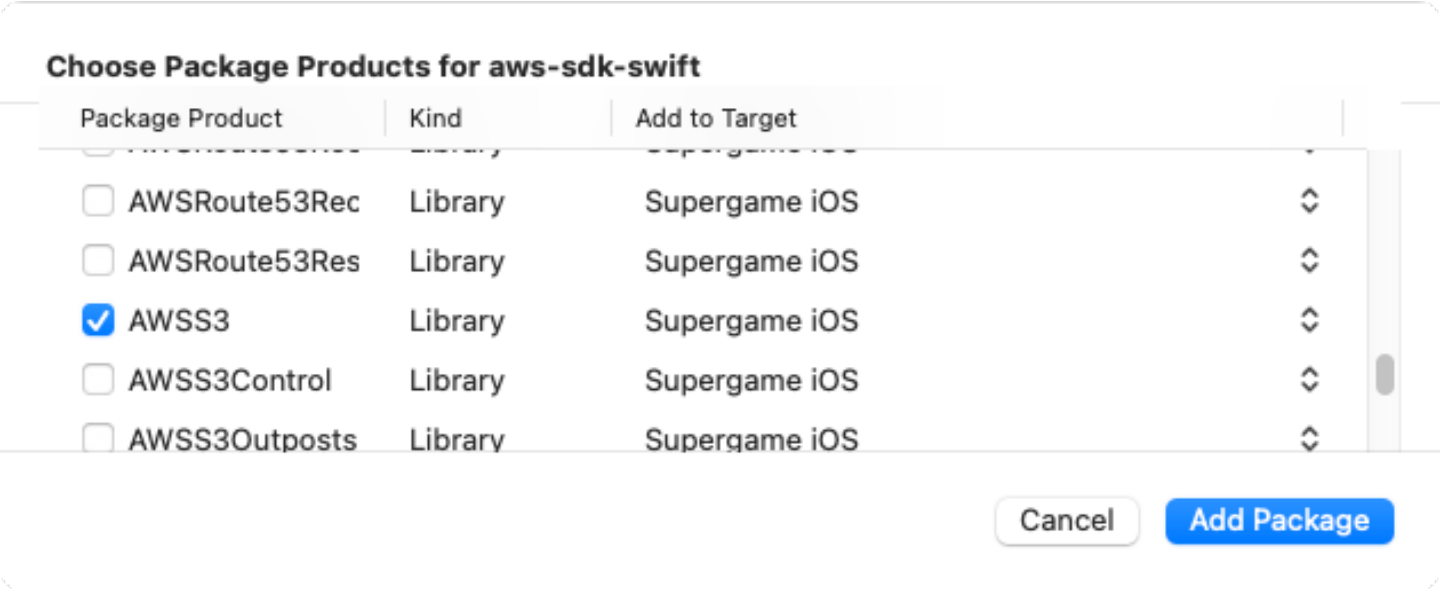
Fetching the AWS SDK for Swift package and its product list



Next, select specific *products* from the AWS SDK for Swift package to include in your project. Each product is generally one AWS API or service. Each package is listed by package name, starting with `AWS` and followed by the shorthand name of the service or toolkit.

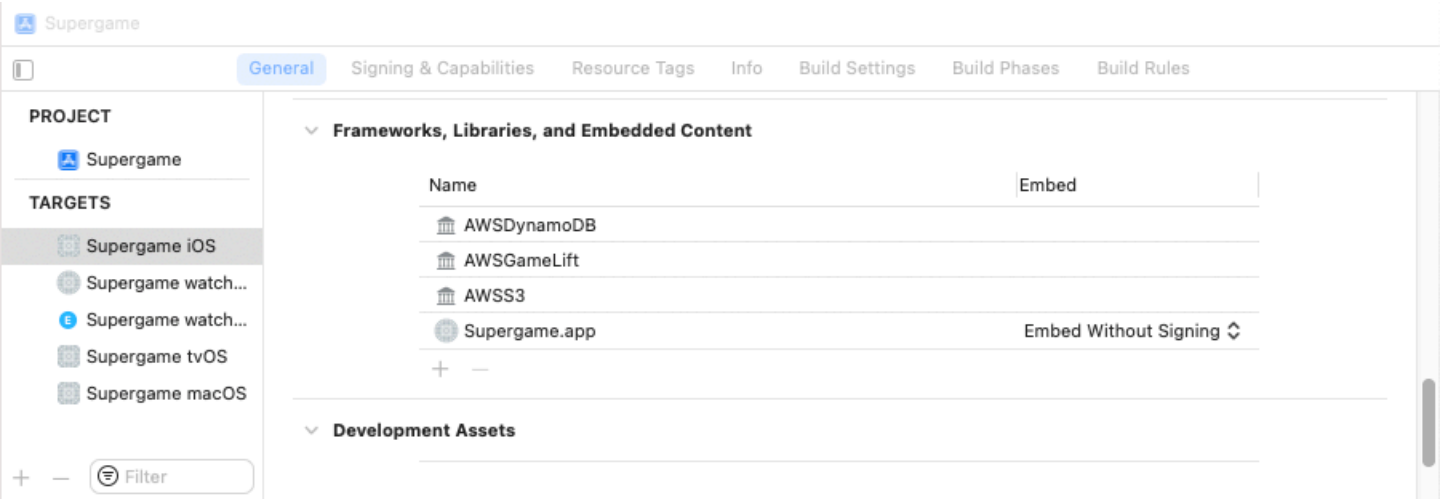
For the Supergame project, select `AWSS3`, `AWSDynamoDB`, and `AWSGameLift`. Assign them to the correct target (iOS in this example), and choose **Add Package**.

Choose package products for specific AWS services and toolkits



Your project is now configured to import the AWS SDK for Swift package and to include the desired APIs in the build for that target. To see a list of the AWS libraries, open the target's **General** tab and scroll down to **Frameworks, Libraries, and Embedded Content**.

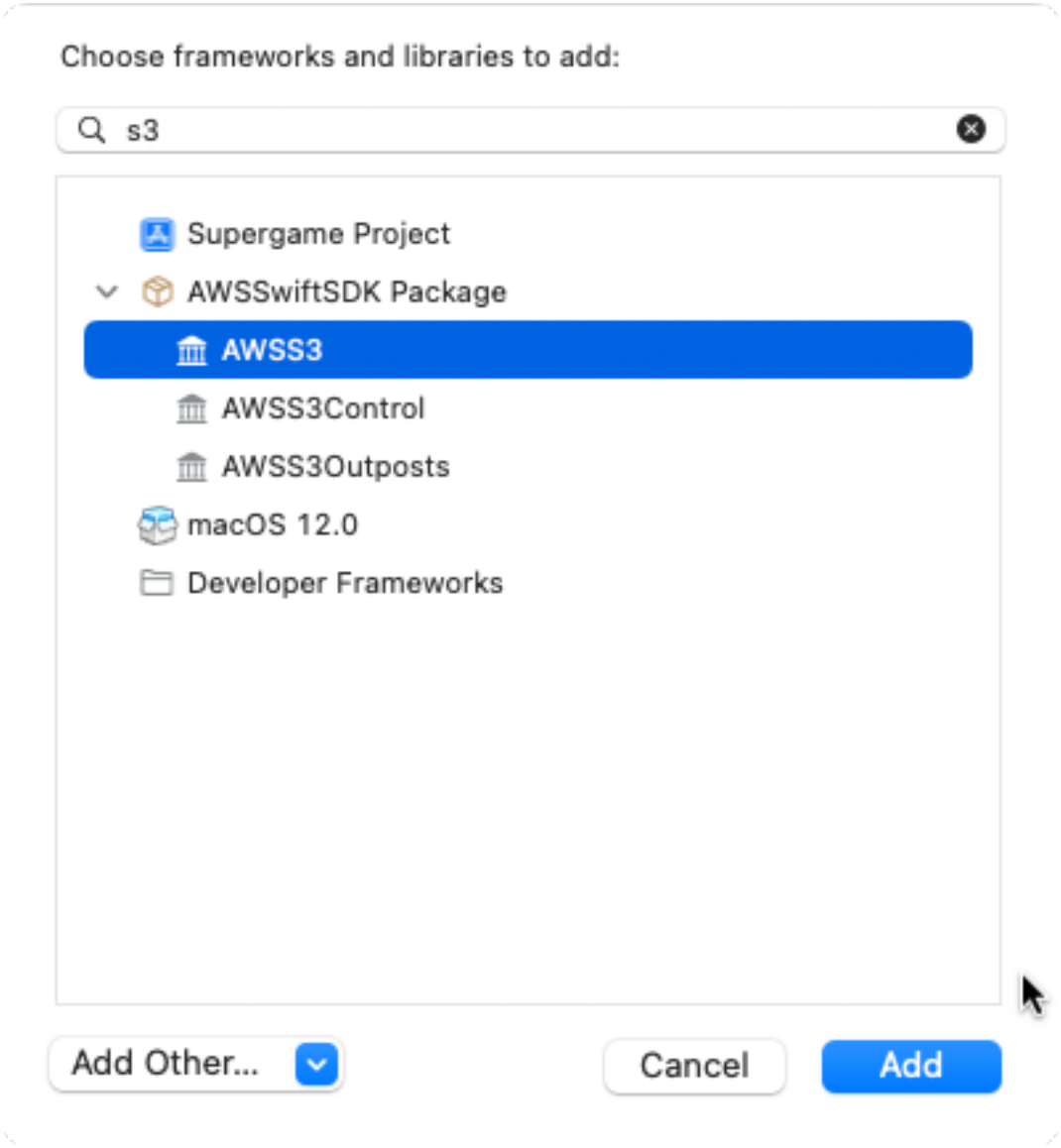
AWS SDK for Swift libraries in the Xcode target



If your project is a multi-platform project, you also need to add the AWS libraries to the other targets in your project. For each platform's target, navigate to the **Frameworks, Libraries, and Embedded Content** section under the **General** tab and choose **+** to open the library picker window.

Then, you can scroll to find and select all of the needed libraries and choose **Add** to add them all at once. Alternatively, you can search for and select each library, then choose **Add** to add them to the target one at a time.

Find and add SDK for Swift libraries using the Xcode library picker window



You're now ready to import the libraries and any needed dependencies into individual Swift source code files and start using the AWS services in your project. Build your project by using the Xcode **Build** option in the **Product** menu.

Build and run an SPM project

To build and run a Swift Package Manager project from a Linux or macOS terminal prompt, use the following commands.

Build a project

```
$ swift build
```

Run a project

```
$ swift run
$ swift run executable-name
$ swift run executable-name arg1, ...
```

If your project builds only one executable file, you can type **swift run** to build and run it. If your project outputs multiple executables, you can specify the file name of the executable you want to run. If you want to pass arguments to the program when you run it, you *must* specify the executable name before listing the arguments.

Get the built product output directory

```
$ swift build --show-bin-path
/home/janice/MyProject/.build/x86_64-unknown-linux-gnu/debug
```

Delete build artifacts

```
$ swift package clean
```

Delete build artifacts and all build caches

```
$ swift package reset
```

Import AWS SDK for Swift libraries into source files

After the libraries are in place, you can use the Swift `import` directive to import the individual libraries into each source file that needs them. To use the functions for a given service, import its library from the AWS SDK for Swift package into your source code file. Also import the `ClientRuntime` library, which contains utility functions and type definitions.

```
import Foundation
import ClientRuntime
import AWSS3
```

The standard Swift library `Foundation` is also imported because it's used by many features of the AWS SDK for Swift.

Additional information

- *Set up*
- *Use the SDK*
- SDK for Swift code examples

Use the SDK for Swift

After completing the steps in *Set up*, you're ready to make requests to AWS services such as Amazon S3, DynamoDB, IAM, Amazon EC2, and more by instantiating service objects and making calls on them using the AWS SDK for Swift. The guides below cover setting up and using AWS services for common use cases.

Customize AWS service client configurations

By default, AWS SDK for Swift uses a basic default configuration for each AWS service. If the user doesn't configure a credential resolver, the default one gets used. The default credential resolver is a chain of resolvers that looks for credentials in the following order:

1. The environment variables `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY_ID`, and `AWS_SESSION_TOKEN`. While these are suitable for development and testing, they shouldn't be relied on for shipped applications.
2. The default AWS profile, as described in the AWS configuration file (located at `~/.aws/config` on Linux and macOS, and the credentials found in the file `~/.aws/credentials` on Linux and macOS.
3. Optionally, the configuration is looked for on Amazon Elastic Container Service.
4. Optionally, the configuration can be taken from Amazon EC2 instance metadata.

Each service might have other options available through its configuration class, depending on that service's needs.

Configuration data types

Each AWS service has its own configuration class that you use to specify adjust options for that client type. All of these classes are based on a number of protocols that together describe all the configuration options available for that client. This includes options such as the Region and credentials, which are always needed in order to access an AWS service.

For Amazon S3, the configuration class is called `S3Client.S3ClientConfiguration` which is defined like this:

```
extension S3Client {
```

```
    public class S3ClientConfiguration: AWSClientRuntime.AWSDefaultClientConfiguration
 &
            AWSClientRuntime.AWSRegionClientConfiguration &
            ClientRuntime.DefaultClientConfiguration &
            ClientRuntime.DefaultHttpClientConfiguration {

        // Service-specific properties are defined here, including:

        public var region: Swift.String?

        // ... and so on.
    }
}
```

As a result, `S3ClientConfiguration` includes its own properties and also all the properties defined in those protocols, making it a complete description of an Amazon S3 client's configuration. Corresponding configuration classes are defined by every AWS service.

By creating a custom configuration object and using it when creating a service client object, you can customize the client by specifying a configuration source from which credentials and other options are taken. Otherwise, you can directly specify the credentials instead of letting the SDK obtain them automatically.

Configure a client

When only changing common options, you can often specify the custom values for your configuration when instantiating the service's client object. If the client class constructor doesn't support the option you need to change, then create and specify a configuration object with the type that corresponds to the client class.

Create a client with only a custom Region

Most services let you directly specify the Region when you call their constructors. For example, to create an Amazon S3 client configured for the Region `af-south-1`, specify the `region` parameter when creating the client, as shown.

```
do {
    let s3 = try S3Client(region: "af-south-1")

    // Use the client.
} catch {
```

```
    // Handle the error.
    dump(error, name: "Error accessing S3 service")
}
```

This lets you handle a common client configuration scenario (specifying a Region while using the default values for all other options) without going through the full configuration process. That process is covered in the next topic.

Create and use a custom configuration

To customize the configuration of an AWS service, create a configuration object of the appropriate type for the service. Then pass that configuration object into the service client's constructor as the value of its `config` parameter.

For example, to configure an Amazon S3 client, create an object of type `S3Client.S3ClientConfiguration`. Set the properties that you want to change, then pass the configuration object into `S3Client(config:)`.

The following example creates a new Amazon S3 client configured with the following options:

- The AWS Region is set to `us-east-1`.
- An exponential backoff strategy with default options is selected instead of the default backoff strategy.
- The retry mode is set to `RetryStrategyOptions.RateLimitingMode.adaptive`. See Retry behavior in the *AWS SDKs and Tools Reference Guide* for details.
- The maximum number of retries is set to 5.

```
        // Create an Amazon S3 client configuration object that specifies the
        // region as "us-east-1", an exponential backoff strategy, the
        // adaptive retry mode, and the maximum number of retries as 5.

        await SDKLoggingSystem().initialize(logLevel: .debug)

        let config: S3Client.S3ClientConfiguration

        do {
            config = try await S3Client.S3ClientConfiguration(
                awsRetryMode: .standard,
                maxAttempts: 3,
                region: "us-east-1"
```

```
        )
    } catch {
        print("Error: Unable to create configuration")
        dump(error)
        exit(1)
    }

    // Create an Amazon S3 client using the configuration created above.

    let client = S3Client(config: config)
```

If an error occurs creating the configuration, an error message is displayed and the details are dumped to the console. The program then exits. In a real world application, a more constructive approach should be taken when handling this error, such as falling back to a different configuration or using the default configuration, if doing so is appropriate for your application.

Use client services with the AWS SDK for Swift

Each AWS service is exposed through a corresponding Swift class. Each class provides a number of functions that you can call to issue requests to AWS services, including Amazon S3, DynamoDB, IAM, and others. Functions that access the network are designed to operate in the background so that your application can continue to run while awaiting the response. The SDK then notifies you when the response arrives.

The process of sending requests to AWS services is as follows:

1. Create a service client object with the desired configuration, such as the specific AWS Region.
2. Create an input object with the values and data needed to make the request. For example, when sending a `GetObject` request to Amazon S3, you need to specify the bucket name and the key of the Amazon S3 object that you want to access. For a request method named `SomeOperation`, the input parameters object is created using a function called `SomeOperationInput()`.
3. Call the service object method on the client object that sends the desired request, with the input object created in the previous step.
4. Use `await` to wait for the response, and handle thrown exceptions to appropriately handle error conditions.
5. Examine the contents of the returned structure for the results you need. Every SDK function returns a structure with a type whose name is the same as the service action performed by the

function, followed by the word `Output`. For example, when calling the Amazon S3 function `S3Client.createBucket(input:)`, the return type is `CreateBucketOutput`.

Create and use AWS client objects

Before you can send requests to an AWS service, you must first instantiate a client object corresponding to the service. These client classes are helpfully named using the service name and the word `Client`. Examples include `S3Client` and `IAMClient`.

After creating the client object, use it to make your requests. When you're done, release the object. If the service connection is open, it is closed for you automatically.

```
do {
    let s3 = try await S3Client()

    // ...
} catch {
    dump(error)
}
```

If an error occurs while trying to instantiate an AWS service — or at any time while using the service — an exception is thrown. Your `catch` block should handle the error appropriately.

Unless you are in a testing environment in which you expect a knowledgeable user to have configured reasonable default options, specify the appropriate service configuration when instantiating the client object. This is described in ???.

Specify service client function parameters

When calling service client methods, you pass the input object corresponding to that operation. For example, before calling the `getObject()` method on the Amazon S3 service class `S3Client`, you need to create the input parameter object using the initializer `GetObjectInput()`.

```
do {
    let s3 = try S3Client()
    let inputObject = GetObjectInput(bucket: "amzn-s3-demo-bucket", key: "keyName")
    let output = try await s3.getObject(input: inputObject)

    // ...
```

```
} catch {
    dump(error)
}
```

In this example, `GetObjectInput()` is used to create an input object for the `getObject(input:)` method. The resulting input object specifies that the desired data has the `keyName` key and should be fetched from the Amazon S3 bucket named `amzn-s3-demo-bucket`.

Call SDK functions

Nearly all AWS SDK for Swift functions are asynchronous and can be called using Swift's `async/await` model.

To call one of the SDK's asynchronous functions from synchronous code, call the function from a Swift `Task` created and run from your synchronous code.

Call SDK functions asynchronously

The following function fetches and returns the content of a file named `text/motd.txt` from a bucket named amzn-s3-demo-bucket, using Amazon S3.

```
func getMOTD() async throws -> String? {
    let s3 = try S3Client()
    let motdInput = GetObjectInput(bucket: "amzn-s3-demo-bucket",
                                   key: "text/motd.txt")
    let output = try await s3.getObject(input: motdInput)

    guard let data = output.body?.toBytes().toData() else {
        return nil
    }
    return String(decoding: data, as: UTF8.self)
}
```

The `getMOTD()` function can only be called from another `async` function, and returns a string that contains the text in the `MOTD` file or `nil` if the file is empty. It throws an exception on errors. Thus, you call the `getMOTD()` function.

```
do {
    let motd = try await getMOTD()
    // ...
```

```
} catch {
    dump(error)
}
```

Here, the fetched "message of the day" text is available in the variable `motd` immediately following the call to `getMOTD()`. If an error occurs attempting to fetch the text, an appropriate exception is delivered to the `catch` clause. The standard Swift variable `error` describes the problem that occurred.

Call SDK functions from synchronous code

To call AWS SDK for Swift functions from synchronous code, enclose the code that needs to run asynchronously in a `Task`. The `Task` uses `await` for each SDK function call that returns its results asynchronously. You might need to use an atomic flag or other means to know that the operation has finished.

> **Note**
>
> It's important to properly manage asynchronous requests. Be sure that any operation that's dependent on a previous result waits until that result is available before it begins. When used properly, the `async/await` model handles most of this for you.

```
func updateMOTD() {
    Task() {
        var motd: String = ""

        do {
            let s3 = try S3Client()
            let motdInput = GetObjectInput(bucket: "amzn-s3-demo-bucket",
                           key: "text/motd.txt")
            let output = try await s3.getObject(input: motdInput)

            if let bytes = output.body?.toBytes() {
                motd = String(decoding: bytes.toData(), as: UTF8.self)
            }
        } catch {
            motd = ""
        }
```

```
        setMOTD(motd)
    }
}
```

In this example, the code inside the `Task` block runs asynchronously, returning no output value to the caller. It fetches the contents of a text file with the key `text/motd.txt` and calls a function named `setMOTD()`, with the contents of the file decoded into a UTF-8 string.

A `do`/`catch` block is used to capture any thrown exceptions and set the `motd` variable to an empty string, which indicates that no message is available.

A call to this `updateMOTD()` function will spawn the task and return immediately. In the background, the program continues to run while the asynchronous task code fetches and uses the text from the specified file on Amazon S3. When the task has completed, the `Task` automatically ends.

Use credential identity resolvers with AWS SDK for Swift

Most AWS service calls require that your app verify users' identity before accessing them. In AWS SDK for Swift applications, this is done using credential identity resolvers. This section of the Guide covers what these are and how to use them.

Overview

A **credential identity resolver** is an object that takes some form of identity, verifies that it's valid for use by the application, and returns credentials that can be used when using an AWS service . There are several supported ways to obtain a valid identity, and each has a corresponding credential identity resolver type available for you to use, depending on which authorization methods you want to use.

The credential identity resolver acts as an adaptor between the identity and the AWS service. By providing a credential identity resolver to the service instead of directly providing the user's credentials, the service is able to fetch currently-valid credentials for the identity at any time, as long as the identity provider allows it.

Identity features in the AWS SDK for Swift are defined in the module `AWSSDKIdentity`. In the `AWSSDKIdentity` module, credentials are represented by the struct `AWSCredentialIdentity`. See AWS security credentials in the IAM User Guide for further information about AWS credentials.

Credential identity resolver types

There are several credential identity resolver types available as a means of obtaining an identity to use for authentication. Some credential identity resolvers are specific to a given source while others encompass an assortment of identity sources that share similar technologies. For example, the `STSWebIdentityCredentialIdentityResolver`, which uses a JSON Web Token (JWT) as the source identity for which to return AWS credentials. The JWT can come from a number of different services, including Amazon Cognito federated identities, Sign In With Apple, Google, or Facebook. See Identity pools third-party identity providers for information on third-party identity providers.

`CachedAWSCredentialIdentityResolver`

A credential identity resolver that is chained with another one so it can cache the resolved identity for re-use until an expiration time elapses.

`CustomAWSCredentialIdentityResolver`

A credential identity resolver that uses another credential identity resolver's output to resolve the credentials in a custom way.

`DefaultAWSCredentialIdentityResolverChain`

Represents a chain of credential identity resolvers that attempt to resolve the identity following the standard search order. See Credential provider chain in the AWS SDKs and Tools Reference Guide for details on the credential provider chain.

`ECSAWSCredentialIdentityResolver`

Obtains credentials from an Amazon Elastic Container Service container's metadata.

`EnvironmentAWSCredentialIdentityResolver`

Resolves credentials using the environment variables `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_SESSION_TOKEN`.

`IMDSAWSCredentialIdentityResolver`

Uses IMDSv2 to fetch credentials within an Amazon Elastic Compute Cloud instance.

`ProcessAWSCredentialIdentityResolver`

Resolves credentials by running a command or process. The process to run is sourced from a profile in the AWS `config` file. The profile key that identifies the process to use is `credential_process`.

`ProfileAWSCredentialIdentityResolver`

Uses the specified profile from an AWS `config` file to resolve credentials.

`SSOAWSCredentialIdentityResolver`

Resolves credentials using a single-sign-on login with AWS IAM Identity Center.

`StaticAWSCredentialIdentityResolver`

A credential resolver that uses specified credentials in the form of an `AWSCredentialIdentity` object.

`STSAssumeRoleAWSCredentialIdentityResolver`

Uses another credential identity resolver to assume a specified AWS Identity and Access Management role, then fetch the assumed credentials using AWS Security Token Service.

`STSWebIdentityAWSCredentialIdentityResolver`

Exchanges a JSON Web Token (JWT) for credentials using AWS Security Token Service.

Getting credentials from an identity

The process of using a credential identity resolver involves four primary steps:

1. Use an appropriate sign-in service to obtain an identity in a form supported by AWS.
2. Create a credential identity resolver of the type that corresponds to the given identity.
3. When creating an AWS service client object, provide the credential identity resolver as the value of its configuration's `awsCredentialIdentityResolver` property.
4. Call service functions using the service client object.

The following sections provide examples using some of the credential identity providers supported by AWS.

SSO credential identity resolvers with AWS IAM Identity Center

Authenticating for an AWS service using SSO requires first configuring SSO access using AWS IAM Identity Center. See IAM Identity Center authentication for your SDK or tool in the AWS SDKs and Tools Reference Guide for instructions on setting up IAM Identity Center and configuring SSO access on computers that will use your application.

Once a user has authenticated with the AWS Command Line Interface (AWS CLI) command **aws sso login** or **aws sso login --profile *profile-name***, your application can use an `SSOAWSCredentialIdentityResolver` to obtain credentials using the established IAM Identity Center identity.

To create an SSO credential identity resolver, create a new `SSOAWSCredentialIdentityResolver` that uses the desired settings for the profile name, `config` file path, and `credentials` file path. Any of these can be `nil` to use the same default value the AWS CLI would use.

```
        let identityResolver = try SSOAWSCredentialIdentityResolver(
            profileName: profile,
            configFilePath: config,
            credentialsFilePath: credentials
        )
```

To use the IAM Identity Center identity resolver to provide credentials to an AWS service, set the service configuration's `awsCredentialIdentityResolver` to the created credential identity resolver.

```
    // Get an S3Client with which to access Amazon S3.
    let configuration = try await S3Client.S3ClientConfiguration(
        awsCredentialIdentityResolver: identityResolver
    )
    let client = S3Client(config: configuration)

    // Use "Paginated" to get all the buckets. This lets the SDK handle
    // the 'continuationToken' in "ListBucketsOutput".
    let pages = client.listBucketsPaginated(
        input: ListBucketsInput(maxBuckets: 10)
    )
```

With the service configured this way, each time the SDK accesses the AWS service, it uses the credentials returned by the SSO credential identity resolver to authenticate the request.

Static credential identity resolvers

> **Warning**
>
> Static credential identity resolvers can be highly unsafe unless used with care. They can return hard-coded credentials, which are inherently unsafe to use. Only use static credential identity resolvers when experimenting, testing, or generating safe credentials from another source before using them.

Static credential identity resolvers use AWS credentials as an identity.

To create a static credential identity resolver, create an `AWSCredentialIdentity` object with the static credentials, then create a new `StaticAWSCredentialIdentityResolver` that uses that identity.

```
        let credentials = AWSCredentialIdentity(
            accessKey: accessKey,
            secret: secretKey,
            sessionToken: sessionToken
        )

        let identityResolver = try StaticAWSCredentialIdentityResolver(credentials)
```

To use the static credential identity resolver to provide credentials to an AWS service, use it as the service configuration's `awsCredentialIdentityResolver`.

```
            let s3Configuration = try await S3Client.S3ClientConfiguration(
                awsCredentialIdentityResolver: identityResolver,
                region: region
            )
            let client = S3Client(config: s3Configuration)

            // Use "Paginated" to get all the buckets. This lets the SDK handle
            // the 'continuationToken' in "ListBucketsOutput".
            let pages = client.listBucketsPaginated(
                input: ListBucketsInput( maxBuckets: 10)
            )
```

When the service client asks the credential identity resolver for its credentials, the resolver returns the `AWSCredentialIdentity` struct's access key, secret, and session token.

The complete example is available on GitHub.

Additional information

- AWS SDKs and Tools Reference Guide: SSO token provider configuration

Create AWS Lambda functions using the AWS SDK for Swift

Overview

You can use the AWS SDK for Swift from within an AWS Lambda function by using the Swift AWS Lambda Runtime package in your project. This package is part of Apple's `swift-server` repository of packages that can be used to develop server-side Swift projects.

See the documentation for the `swift-aws-lambda-runtime` repository on GitHub for more information about the runtime package.

Set up a project to use AWS Lambda

If you're starting a new project, create the project in Xcode or open a shell session and use the following command to use Swift Package Manager (SwiftPM) to manage your project:

```
$ swift package init --type executable --name LambdaExample
```

Remove the file `Sources/main.swift`. The source code file will have be `Sources/lambda.swift` to work around a known Swift bug that can cause problems when the entry point is in a file named `main.swift`.

Add the `swift-aws-lambda-runtime` package to the project. There are two ways to accomplish this:

- If you're using Xcode, choose the **Add package dependencies...** option in the **File** menu, then provide the package URL: `https://github.com/swift-server/swift-aws-lambda-runtime.git`. Choose the `AWSLambdaRuntime` module.

- If you're using SwiftPM to manage your project dependencies, add the runtime package and its `AWSLambdaRuntime` module to your `Package.swift` file to make the module available to your project:

```
import PackageDescription

let package = Package(
    name: "LambdaExample",
    platforms: [
        .macOS(.v12)
    ],
    // The product is an executable named "LambdaExample", which is built
    // using the target "LambdaExample".
    products: [
        .executable(name: "LambdaExample", targets: ["LambdaExample"])
    ],
    // Add the dependencies: these are the packages that need to be fetched
    // before building the project.
    dependencies: [
        .package(
            url: "https://github.com/swift-server/swift-aws-lambda-runtime.git",
            from: "1.0.0-alpha"),
        .package(url: "https://github.com/awslabs/aws-sdk-swift.git",
            from: "1.0.0"),
    ],
    targets: [
        // Add the executable target for the main program. These are the
        // specific modules this project uses within the packages listed under
        // "dependencies."
        .executableTarget(
            name: "LambdaExample",
            dependencies: [
                .product(name: "AWSLambdaRuntime", package: "swift-aws-lambda-
runtime"),
                .product(name: "AWSS3", package: "aws-sdk-swift"),
            ]
        )
    ]
)
```

This example adds a dependency on the Amazon S3 module of the AWS SDK for Swift in addition to the Lambda runtime.

You may find it useful to build the project at this point. Doing so will pull the dependencies and may make them available for your editor or IDE to generate auto-completion or inline help:

```
$ swift build
```

> **Warning**
>
> As of this article's last revision, the Swift AWS Lambda Runtime package is in a preview state and may have flaws. It also may change significantly before release. Keep this in mind when making use of the package.

Create a Lambda function

To create a Lambda function in Swift, you generally need to define several components:

- A `struct` that represents the data your Lambda function will receive from the client. It must implement the `Decodable` protocol. The Swift AWS Lambda Runtime Events library contains a variety of struct definitions that represent common messages posted to a Lambda function by other AWS services.
- An `async` handler function that performs the Lambda function's work.
- An optional `init(context:)` function that configures logging and sets up other variables or services that must be created once per execution environment.
- An optional `struct` that represents the data returned by your Lambda function. This is usually an `Encodable` `struct` describing the contents of a JSON document returned to the client.

Imports

Create the file `Sources/lambda.swift` and begin by importing the needed modules and types:

```
import Foundation
import AWSLambdaRuntime
import AWSS3

import protocol AWSClientRuntime.AWSServiceError
import enum Smithy.ByteStream
```

These imports are:

1. The standard Apple `Foundation` API.
2. `AWSLambdaRuntime`is the Swift Lambda Runtime's main module.
3. `AWSS3` is the Amazon S3 module from the AWS SDK for Swift.
4. The `AWSClientRuntime.AWSServiceError` protocol describes service errors returned by the SDK.
5. The `Smithy.ByteStream` enum is a type that represents a stream of data. The Smithy library is one of the SDK's core modules.

Define structs and enums

Next, define the structs that represent the incoming requests and the responses sent back by the Lambda function, along with the enum used to identify errors thrown by the handler function:

```
/// Represents the contents of the requests being received from the client.
/// This structure must be `Decodable` to indicate that its initializer
/// converts an external representation into this type.
struct Request: Decodable, Sendable {
    /// The request body.
    let body: String
}

/// The contents of the response sent back to the client. This must be
/// `Encodable`.
struct Response: Encodable, Sendable {
    /// The ID of the request this response corresponds to.
    let req_id: String
    /// The body of the response message.
    let body: String
}

/// The errors that the Lambda function can return.
enum S3ExampleLambdaErrors: Error {
    /// A required environment variable is missing. The missing variable is
    /// specified.
    case noEnvironmentVariable(String)
    /// The Amazon Simple Storage Service (S3) client couldn't be created.
    case noS3Client
}
```

The Lambda handler struct

This example creates a Lambda function that writes a string to an Amazon Simple Storage Service object. To do so, a struct of type `LambdaHandler` is needed:

```
@main
struct S3ExampleLambda: LambdaHandler {
    let s3Client: S3Client?

    ...

}
```

Initializer

The `S3ExampleLambda` struct's initializer `init(context:)` prepares the handler by logging the configured logging level (as specified by the environment variable `LOG_LEVEL`), fetches the AWS Region configured by the environment variable `AWS_REGION`, and creates an instance of the SDK for Swift `S3Client`, which will be used for all Amazon S3 requests:

```
    /// Initialize the AWS Lambda runtime.
    ///
    /// ^ The logger is a standard Swift logger. You can control the verbosity
    /// by setting the `LOG_LEVEL` environment variable.
    init(context: LambdaInitializationContext) async throws {
        // Display the `LOG_LEVEL` configuration for this process.
        context.logger.info(
            "Log Level env var : \(ProcessInfo.processInfo.environment["LOG_LEVEL"] ??
 "info" )"
        )

        // Initialize the Amazon S3 client. This single client is used for every
        // request.
        let currentRegion = ProcessInfo.processInfo.environment["AWS_REGION"] ?? "us-
east-1"
        self.s3Client = try? S3Client(region: currentRegion)
    }
```

Function handler

To receive, process, and respond to incoming requests, the Swift Lambda Runtime's `LambdaHandler` protocol requires you to implement the `handle(event:context:)` function:

```
    /// The Lambda function's entry point. Called by the Lambda runtime.
    ///
    /// - Parameters:
    ///   - event: The `Request` describing the request made by the
    ///     client.
    ///   - context: A `LambdaContext` describing the context in
    ///     which the lambda function is running.
    ///
    /// - Returns: A `Response` object that will be encoded to JSON and sent
    ///   to the client by the Lambda runtime.
    func handle(_ event: Request, context: LambdaContext) async throws -> Response {
        // Get the bucket name from the environment.
        guard let bucketName = ProcessInfo.processInfo.environment["BUCKET_NAME"] else
{
            throw S3ExampleLambdaErrors.noEnvironmentVariable("BUCKET_NAME")
        }

        // Make sure the `S3Client` is valid.
        guard let s3Client else {
            throw S3ExampleLambdaErrors.noS3Client
        }

        // Call the `putObject` function to store the object on Amazon S3.
        var responseMessage: String
        do {
            let filename = try await putObject(
                            client: s3Client,
                            bucketName: bucketName,
                            body: event.body)

            // Generate the response text.
            responseMessage = "The Lambda function has successfully stored your data in
S3 with name \(filename)'"

            // Send the success notification to the logger.
            context.logger.info("Data successfully stored in S3.")
        } catch let error as AWSServiceError {
            // Generate the error message.
            responseMessage = "The Lambda function encountered an error and your data
was not saved. Root cause: \(error.errorCode ?? "") - \(error.message ?? "")"

            // Send the error message to the logger.
            context.logger.error("Failed to upload data to Amazon S3.")
```

```
        }

        // Return the response message. The AWS Lambda runtime will send it to the
        // client.
        return Response(
            req_id: context.requestID,
            body: responseMessage)
    }
```

The name of the bucket to use is first fetched from the environment variable `BUCKET_NAME`. Then the `putObject(client:bucketName:body:)` function is called to write the text into an Amazon S3 object, and the text of the response is set depending on whether or not the object is successfully written to storage. The response is created with the response message and the request ID string that was in the original Lambda request.

Helper function

This example uses a helper function, `putObject(client:bucketName:body:)`, to write strings to Amazon S3. The string is stored in the specified bucket by creating an object whose name is based on the current number of seconds since the Unix epoch:

```
    /// Write the specified text into a given Amazon S3 bucket. The object's
    /// name is based on the current time.
    ///
    /// - Parameters:
    ///   - s3Client: The `S3Client` to use when sending the object to the
    ///     bucket.
    ///   - bucketName: The name of the Amazon S3 bucket to put the object
    ///     into.
    ///   - body: The string to write into the new object.
    ///
    /// - Returns: A string indicating the name of the file created in the AWS
    ///   S3 bucket.
    private func putObject(client: S3Client,
                           bucketName: String,
                           body: String) async throws -> String {
        // Generate an almost certainly unique object name based on the current
        // timestamp.
        let objectName = "\(Int(Date().timeIntervalSince1970*1_000_000)).txt"

        // Create a Smithy `ByteStream` that represents the string to write into
        // the bucket.
```

```
        let inputStream = Smithy.ByteStream.data(body.data(using: .utf8))

        // Store the text into an object in the Amazon S3 bucket.
        let putObjectRequest = PutObjectInput(
            body: inputStream,
            bucket: bucketName,
            key: objectName
        )
        let _ = try await client.putObject(input: putObjectRequest)

        // Return the name of the file.
        return objectName
    }
```

Build and test locally

While you can test your Lambda function by adding it in the Lambda console, the Swift AWS Lambda Runtime provides an integrated Lambda server you can use for testing. This server accepts requests and dispatches them to your Lambda function.

To use the integrated web server for testing, define the environment variable `LOCAL_LAMBDA_SERVER_ENABLED` before running the program.

In this example, the program is built and run with the Region set to `eu-west-1`, the bucket name set to `amzn-s3-demo-bucket`, and the local Lambda server enabled:

```
$ AWS_REGION=eu-west-1             \
BUCKET_NAME=amzn-s3-demo-bucket    \
LOCAL_LAMBDA_SERVER_ENABLED=true \
swift run
```

After running this command, the Lambda function is available on the local server.
Test it by opening another terminal session and using it to send a Lambda request to `http://127.0.0.1:7000/invoke`, or to port 7000 on `localhost`:

```
$ curl -X POST                          \
     --data '{"body":"This is the message to store on Amazon S3."}' \
   http://127.0.0.1:7000/invoke
```

Upon success, a JSON object similar to this is returned:

```
{
  "req_id": "290935198005708",
  "body": "The Lambda function has successfully stored your data in S3 with name
 '1720098625801368.txt'"
}
```

You can remove the created object from your bucket using this AWS CLI command:

```
$ aws s3 rm s3://amzn-s3-demo-bucket/file-name
```

Package and upload the app

To use a Swift app as a Lambda function, compile it for an x86_64 or ARM Linux target depending on the build machine's architecture. This may involve cross-compiling, so you may need to resolve dependency issues, even if they don't happen when building for your build system.

The Swift Lambda Runtime includes an `archive` command as a plugin for the Swift compiler. This plugin lets you cross-compile from macOS to Linux just using the standard `swift` command. The plugin uses a Docker container to build the Linux executable, so you'll need Docker installed.

To build your app for use as a Lambda function:

1. Build the app using the SwiftPM `archive` plugin. This automatically selects the architecture based on that of your build machine (x86_64 or ARM).

   ```
   $ swift package archive --disable-sandbox
   ```

   This creates a ZIP file containing the function executable, placing the output in `.build/plugins/AWSLambdaPackager/outputs/AWSLambdaPackager/`*`target-name`*`/`*`executable-name`*`.zip`
2. Create a Lambda function using the method appropriate for your needs, such as:
   - AWS Lambda Developer Guide
   - AWS Command Line Interface User Guide
   - AWS Cloud Development Kit (AWS CDK) Developer Guide

   Warning

   Things to keep in mind when deploying the Lambda function:

- Use the same architecture (x86_64 or ARM64) for your function and your binary.
- Use the Amazon Linux 2 runtime.
- Define any environment variables required by the function. In this example, the `BUCKET_NAME` variable needs to be set to the name of the bucket to write objects into.
- Give your function the needed permissions to access AWS resources. For this example, the function needs IAM permission to use `PutObject` on the bucket specified by `BUCKET_NAME`.

3. Once you've created and deployed the Swift-based Lambda function, it should be ready to accept requests. You can invoke the function using the `Invoke` Lambda API.

```
$ aws lambda invoke                     \
--region eu-west-1                      \
--function-name LambdaExample           \
--cli-binary-format raw-in-base64-out   \
--payload '{"body":"test message"}'     \
output.json
```

The file `output.json` contains the results of the invocation (or the error message injected by our code).

Additional information

- Swift AWS Lambda Runtime package on GitHub
- Swift AWS Lambda Events package on GitHub
- AWS Lambda Developer Guide
- AWS Lambda API Reference

Event streaming with AWS SDK for Swift

Overview

Some AWS services provide timely feedback about the state of a system or process by streaming to your application a series of events describing that state, if your application wants to receive them. Likewise, other services may be able to receive a stream of events from your application to provide

needed data as it becomes available. The AWS SDK for Swift provides support for sending and receiving streams of events with services that support this feature.

This section of the guide demonstrates how to stream events to a service and receive events from a service, with an example that uses Amazon Transcribe to transcribe voice content from an audio file into text displayed on the screen.

Event streaming example

This Amazon Transcribe example uses the swift-argument-parser package from Apple to parse the command line neatly, as well as the `AWSTranscribeStreaming` module from the AWS SDK for Swift.

The example's complete source code is available on GitHub.

Import modules

The example begins by importing the modules it needs:

```
import ArgumentParser
import AWSClientRuntime
import AWSTranscribeStreaming
import Foundation
```

Enum definition

Then an enum is defined to represent the three audio formats Amazon Transcribe supports for streaming. These are used to match against the format specified on the command line using the **--format** option.:

```
/// Identify one of the media file formats supported by Amazon Transcribe.
enum TranscribeFormat: String, ExpressibleByArgument {
    case ogg = "ogg"
    case pcm = "pcm"
    case flac = "flac"
}
```

Create the audio stream

A function named `createAudioStream()` returns an `AsyncThrowingStream` that contains the audio file's contents, broken into 125ms chunks. The `AsyncThrowingStream` supplies audio

data to Amazon Transcribe. The stream is specified as an input property when calling the client's `startStreamTranscription(input:)` function.

```
    /// Create and return an Amazon Transcribe audio stream from the file
    /// specified in the arguments.
    ///
    /// - Throws: Errors from `TranscribeError`.
    ///
    /// - Returns: `AsyncThrowingStream<TranscribeStreamingClientTypes.AudioStream,
Error>`
    func createAudioStream() async throws
                -> AsyncThrowingStream<TranscribeStreamingClientTypes.AudioStream,
Error> {

        let fileURL: URL = URL(fileURLWithPath: path)
        let audioData = try Data(contentsOf: fileURL)

        // Properties defining the size of audio chunks and the total size of
        // the audio file in bytes. You should try to send chunks that last on
        // average 125 milliseconds.

        let chunkSizeInMilliseconds = 125.0
        let chunkSize = Int(chunkSizeInMilliseconds  / 1000.0 * Double(sampleRate) *
2.0)
        let audioDataSize = audioData.count

        // Create an audio stream from the source data. The stream's job is
        // to send the audio in chunks to Amazon Transcribe as
        // `AudioStream.audioevent` events.

        let audioStream =
AsyncThrowingStream<TranscribeStreamingClientTypes.AudioStream,
                                Error> { continuation in
            Task {
                var currentStart = 0
                var currentEnd = min(chunkSize, audioDataSize - currentStart)

                // Generate and send chunks of audio data as `audioevent`
                // events until the entire file has been sent. Each event is
                // yielded to the SDK after being created.

                while currentStart < audioDataSize {
                    let dataChunk = audioData[currentStart ..< currentEnd]
```

```
                    let audioEvent =
TranscribeStreamingClientTypes.AudioStream.audioevent(
                        .init(audioChunk: dataChunk)
                    )
                    let yieldResult = continuation.yield(audioEvent)
                    switch yieldResult {
                        case .enqueued(_):
                            // The chunk was successfully enqueued into the
                            // stream. The `remaining` parameter estimates how
                            // much room is left in the queue, but is ignored here.
                            break
                        case .dropped(_):
                            // The chunk was dropped because the queue buffer
                            // is full. This will cause transcription errors.
                            print("Warning: Dropped audio! The transcription will be
incomplete.")
                        case .terminated:
                            print("Audio stream terminated.")
                            continuation.finish()
                            return
                        default:
                            print("Warning: Unrecognized response during audio
streaming.")
                    }

                    currentStart = currentEnd
                    currentEnd = min(currentStart + chunkSize, audioDataSize)
                }

                // Let the SDK's continuation block know the stream is over.

                continuation.finish()
            }
        }

        return audioStream
    }
```

This function returns an `AsyncThrowingStream<TranscribeStreamingClientTypes.AudioStream, Error>`. This is a function that asynchronously generates chunks of audio data, yielding them to the caller, until there's no audio left to process.

The function begins by creating a Foundation `URL` from the path of the audio file. Then it reads the audio into a `Data` object (to support larger audio files, this would need to be changed to load the audio from disk in chunks). The size of each audio chunk to send to the SDK is calculated so it will hold 125 milliseconds of audio, and the total size of the audio file in bytes is obtained.

The audio stream is generated by iterating over the audio data, taking the next chunk of audio and creating a `TranscribeStreamingClientTypes.AudioStream.audioevent` that represents it. The event is sent to the SDK using the continuation object's `yield()` function. The yield result is checked to see if any problems occurred, such as the event being dropped because the event queue is full.

This continues until the last chunk of audio is sent; then the continuation's `finish()` function is executed to let the SDK know the file has been fully transmitted.

Transcribe audio

Transcription is handled by the `transcribe()` function:

```
    /// Run the transcription process.
    ///
    /// - Throws: An error from `TranscribeError`.
    func transcribe() async throws {
        // Convert the value of the `--format` option into the Transcribe
        // Streaming `MediaEncoding` type.

        let mediaEncoding: TranscribeStreamingClientTypes.MediaEncoding
        switch format {
        case .flac:
            mediaEncoding = .flac
        case .ogg:
            mediaEncoding = .oggOpus
        case .pcm:
            mediaEncoding = .pcm
        }

        // Create the Transcribe Streaming client.

        let client = TranscribeStreamingClient(
            config: try await
TranscribeStreamingClient.TranscribeStreamingClientConfiguration(
                                region: region
            )
```

```
        )

        // Start the transcription running on the audio stream.

        let output = try await client.startStreamTranscription(
            input: StartStreamTranscriptionInput(
                audioStream: try await createAudioStream(),
                languageCode: TranscribeStreamingClientTypes.LanguageCode(rawValue:
lang),
                mediaEncoding: mediaEncoding,
                mediaSampleRateHertz: sampleRate
            )
        )

        // Iterate over the events in the returned transcript result stream.
        // Each `transcriptevent` contains a list of result fragments which
        // need to be concatenated together to build the final transcript.
        for try await event in output.transcriptResultStream! {
            switch event {
            case .transcriptevent(let event):
            for result in event.transcript?.results ?? [] {
                guard let transcript = result.alternatives?.first?.transcript else {
                    continue
                }

                // If showing partial results is enabled and the result is
                // partial, show it. Partial results may be incomplete, and
                // may be inaccurate, with upcoming audio making the
                // transcription complete or by giving more context to make
                // transcription make more sense.

                if (result.isPartial && showPartial) {
                    print("[Partial] \(transcript)")
                }

                // When the complete fragment of transcribed text is ready,
                // print it. This could just as easily be used to draw the
                // text as a subtitle over a playing video, though timing
                // would need to be managed.

                if !result.isPartial {
                    if (showPartial) {
                        print("[Final  ] ", terminator: "")
                    }
```

```
                    print(transcript)
                }
            }
            default:
                print("Error: Unexpected message from Amazon Transcribe:")
            }
        }
    }
```

This function first looks at the value of the **--format** option passed into the program on the command line and prepares a constant of type `TranscribeStreamingClientTypes.MediaEncoding` that indicates the format of the incoming audio. Then it calls `client.startStreamTranscription(input:)` to start the transcription process. The audio stream is specified by a function named `createAudioStream()`, which is described below.

The event stream returned by `startStreamTranscription(input:)` is monitored using a `for await` loop. Each `transcriptevent` is handled by pulling the first available transcription from the result stream. If the transcript is flagged as partial and the user specified the **--show-partial** option on the command line, the partial output is printed to the console.

If it's a completed transcription of a section of the audio, the transcription is output to the screen. The importance of checking the value of the result's `isPartial` property is simple: as chunks of audio are processed, they may contain partial words that need to be completed by referring to other chunks. Similarly, if a transcription's certainty is low, it might be higher if subsequent chunks provide additional context. For example, if the transcription includes the word "its," the following chunk may help determine if the word should actually be "it's" instead.

Run the example

If you download and build the complete example, you can run it using the **tsevents** executable. For example, if you have a 44,100Hz audio file named `audio-sample.flac`, you can process it with the command:

```
$ tsevents --path audio-sample.flac --format flac --sample-rate 44100
```

If the language of the audio file isn't US English, you can specify the file's language using the **--lang** option. For example, for modern Arabic, you can use:

```
$ tsevents --path audio-sample.flac --format flac --sample-rate 44100 --lang ar-SA
```

For complete usage information, simply run the command **tsevents --help**.

Additional information

- Amazon Transcribe Developer Guide
- Amazon Transcribe API Reference

Binary streaming with AWS SDK for Swift

Overview

In the AWS SDK for Swift, binary data can be represented as a stream of data directly from a file or other resource. This is done using the Smithy `ByteStream` type, which represents an abstract stream of bytes.

Streams are handled automatically by the SDK. Incoming data from downloads is accepted using a stream so you don't have to manage receiving and combining multiple inbound chunks of an object yourself. Similarly, you can choose to upload a file using a stream as the data source, which lets you avoid needing to write multipart upload code manually.

The example code in this section can be found in its entirety on GitHub.

Streaming incoming data

The `*Output` structs returned by functions that receive incoming data contain a `body` property whose value is a `ByteStream`. The `ByteStream` returns its data either as a Swift `Data` object or as a Smithy `ReadableStream`, from which you can read the data.

```
    /// Download a file from the specified bucket.
    ///
    /// - Parameters:
    ///   - bucket: The Amazon S3 bucket name to get the file from.
    ///   - key: The name (or path) of the file to download from the bucket.
    ///   - destPath: The pathname on the local filesystem at which to store
    ///     the downloaded file.
    func downloadFile(bucket: String, key: String, destPath: String?) async throws {
        let fileURL: URL

        // If no destination path was provided, use the key as the name to use
        // for the file in the downloads folder.
```

```
        if destPath == nil {
            do {
                try fileURL = FileManager.default.url(
                    for: .downloadsDirectory,
                    in: .userDomainMask,
                    appropriateFor: URL(string: key),
                    create: true
                ).appendingPathComponent(key)
            } catch {
                throw TransferError.directoryError
            }
        } else {
            fileURL = URL(fileURLWithPath: destPath!)
        }

        let config = try await S3Client.S3ClientConfiguration(region: region)
        let s3Client = S3Client(config: config)

        // Create a `FileHandle` referencing the local destination. Then
        // create a `ByteStream` from that.

        FileManager.default.createFile(atPath: fileURL.path, contents: nil, attributes:
nil)
        let fileHandle = try FileHandle(forWritingTo: fileURL)

        // Download the file using `GetObject`.

        let getInput = GetObjectInput(
            bucket: bucket,
            key: key
        )

        do {
            let getOutput = try await s3Client.getObject(input: getInput)

            guard let body = getOutput.body else {
                throw TransferError.downloadError("Error: No data returned for
download")
            }

            // If the body is returned as a `Data` object, write that to the
            // file. If it's a stream, read the stream chunk by chunk,
            // appending each chunk to the destination file.
```

```
            switch body {
            case .data:
                guard let data = try await body.readData() else {
                    throw TransferError.downloadError("Download error")
                }

                // Write the `Data` to the file.

                do {
                    try data.write(to: fileURL)
                } catch {
                    throw TransferError.writeError
                }
                break

            case .stream(let stream as ReadableStream):
                while (true) {
                    let chunk = try await stream.readAsync(upToCount: 5 * 1024 * 1024)
                    guard let chunk = chunk else {
                        break
                    }

                    // Write the chunk to the destination file.

                    do {
                        try fileHandle.write(contentsOf: chunk)
                    } catch {
                        throw TransferError.writeError
                    }
                }

                break
            default:
                throw TransferError.downloadError("Received data is unknown object
type")
            }
        } catch {
            throw TransferError.downloadError("Error downloading the file: \(error)")
        }

        print("File downloaded to \(fileURL.path).")
    }
```

This function builds a Swift URL representing the destination path for the downloaded file, using either the path specified by `destPath`, or by creating a file in the user's `Downloads` directory with the same name as the object being downloaded. Then an Amazon S3 client is created, the file is created using the Foundation `FileManager` class, and the download is started by calling the `S3Client.getObject(input:)` function.

If the returned `GetObjectOutput` object's `body` property matches `.data`, the stream's contents are retrieved using the `ByteStream` function `readData()`, and the data is written to the file. If `body` matches `.stream`, the associated `ReadableStream` is read by repeatedly calling the stream's `readAsync(upToCount:)` function to get chunks of the file, writing each chunk to the file until no chunk is received, at which point the download ends. Any other type of `body` is unknown, causing an error to throw.

Streaming outgoing data

When sending data to an AWS service, you can use a `ByteStream` to send data too large to reasonably store in memory in its entirety, or data that is being generated in real time.

```
    /// Upload a file to the specified bucket.
    ///
    /// - Parameters:
    ///   - bucket: The Amazon S3 bucket name to store the file into.
    ///   - key: The name (or path) of the file to upload to in the `bucket`.
    ///   - sourcePath: The pathname on the local filesystem of the file to
    ///     upload.
    func uploadFile(sourcePath: String, bucket: String, key: String?) async throws {
        let fileURL: URL = URL(fileURLWithPath: sourcePath)
        let fileName: String

        // If no key was provided, use the last component of the filename.

        if key == nil {
            fileName = fileURL.lastPathComponent
        } else {
            fileName = key!
        }

        let s3Client = try await S3Client()

        // Create a FileHandle for the source file.

        let fileHandle = FileHandle(forReadingAtPath: sourcePath)
```

```
        guard let fileHandle = fileHandle else {
            throw TransferError.readError
        }

        // Create a byte stream to retrieve the file's contents. This uses the
        // Smithy FileStream and ByteStream types.

        let stream = FileStream(fileHandle: fileHandle)
        let body = ByteStream.stream(stream)

        // Create a `PutObjectInput` with the ByteStream as the body of the
        // request's data. The AWS SDK for Swift will handle sending the
        // entire file in chunks, regardless of its size.

        let putInput = PutObjectInput(
            body: body,
            bucket: bucket,
            key: fileName
        )

        do {
            _ = try await s3Client.putObject(input: putInput)
        } catch {
            throw TransferError.uploadError("Error uploading the file: \(error)")
        }

        print("File uploaded to \(fileURL.path).")
    }
```

This function opens the source file for reading using the Foundation `FileHandle` class. The `FileHandle` is then used to create a Smithy `ByteStream` object. The stream is specified as the `body` when setting up the `PutObjectInput`, which is in turn used to upload the file. Since a stream was specified as the `body`, the SDK automatically continues to send the stream's data until the end of the file is reached.

Use AWS SDK for Swift paginators

Many AWS operations return paginated results when the payload is too large to return in a single response. The AWS SDK for Swift provides specialized versions of the functions that do this. These special functions end with the word **`Paginated`**. All your code needs to do is process the results as they arrive.

Each paginator is a function that returns an object of type `PaginatorSequence<`*`input-type`*`, `*`output-type`*`>`. The `PaginatorSequence<>` is an `AsyncSequence`; `AsyncSequence` is a "lazy" sequence, so no AWS service requests are made until you start iterating over the pages. This also means that any errors that occur during the operation don't reach you until iteration begins.

> **Note**
>
> The examples in this section of the developer guide use Amazon S3. However, the concept is the same for any service that has one or more paginated APIs.

For example, the paginated version of the `S3Client` function `listBuckets(input:)`, `listBucketsPaginated(input:)`, returns an object of type `PaginatorSequence<ListBucketsInput, ListBucketsOutput>`:

```
        let pages = client.listBucketsPaginated(
            input: ListBucketsInput(maxBuckets: PAGE_SIZE)
        )
```

In this example, the number of results in each page is specified by adding a `maxBuckets` property to the `ListBucketsInput` object. Each paginator uses an appropriate name for this property. As of the time `listBucketsPaginated(input:)` returns, no requests have been sent to the Amazon S3 service.

The `PaginatorSequence<>` is a sequence of pages which are asynchronously added to the sequence as the results are received. The type of each entry in the sequence is the `Output` `struct` corresponding to the function called. For example, if you call `S3Client.listBucketsPaginated(input:)`, each entry in the sequence is a `ListBucketsOutput` object. Each entry's buckets can be found in the its `ListBucketsOutput.buckets` property, which is an array of objects of type `S3ClientTypes.Bucket`.

To begin sending requests and receiving results, asynchronously iterate over each page, then iterate over each page's items:

```
        var pageNumber = 0

        do {
            for try await page in pages {
```

```
            pageNumber += 1

            guard let pageBuckets = page.buckets else {
                print("ERROR: No buckets returned in page \(pageNumber)")
                continue
            }

            print("\nPage \(pageNumber):")

            // Print this page's bucket names.

            for bucket in pageBuckets {
                print("  " + (bucket.name ?? "<unknown>"))
            }
        }
    } catch {
        print("ERROR: Unable to process bucket list pages.")
    }
```

The outer `for` loop uses `await` to process pages of results as they're delivered, asynchronously. Once a page is received, the inner loop iterates over the buckets found in each entry's `buckets` property. The full example is available on GitHub.

Presigned URLs and requests with AWS SDK for Swift

Overview

You can presign certain AWS API operations in advance of their use to let the request be used at a later time without the need to provide credentials. With a presigned URL, the owner of a resource can grant an unauthorized person access to a resource for a limited time by simply sending the other user a presigned URL to use the resource.

Presigning basics

The AWS SDK for Swift provides functions that create presigned URLs or requests for each of the service actions that support presigning. These functions take as their `input` parameter the same input struct used by the unsigned action, plus an expiration time that restricts how long the presigned request will be valid and usable.

For example, to create a presigned request for the Amazon S3 action `GetObject`:

```
        let getInput = GetObjectInput(
            bucket: bucket,
            key: key
        )

        let presignedRequest: URLRequest
        do {
            presignedRequest = try await s3Client.presignedRequestForGetObject(
                input: getInput,
                expiration: TimeInterval(5 * 60)
            )
        } catch {
            throw TransferError.signingError
        }
```

This first creates a `GetObjectInput` struct to identify the object to retrieve, then creates a Foundation `URLRequest` with the presigned request by calling the SDK for Swift function `presignedRequestForGetObject(input: expiration:)`, specifying the input struct and a five-minute expiration period. The resulting request can then be sent by anyone, up to five minutes after the request was created. The codebase that sends the request can be in a different application, and even written in a different programming language.

Advanced presigning configuration

The SDK's input structs used to pass options into presignable operations have two methods you can call to generate presigned requests or URLs. For example, the `PutObjectInput` struct has the methods `presign(config: expiration:)` and `presignURL(config: expiration:)`. These are useful if you need to apply to the operation a configuration other than the one used when initializing the service client.

In this example, the `AsyncHTTPClient` package from Apple's `swift-server` project is used to create an `HTTPClientRequest`, which in turn is used to send the file to Amazon S3. A custom configuration is created that lets the SDK make up to six attempts to send an object to Amazon S3:

```
        let fileData = try Data(contentsOf: fileURL)
        let dataStream = ByteStream.data(fileData)
        let presignedURL: URL

        // Create a presigned URL representing the `PutObject` request that
        // will upload the file to Amazon S3. If no URL is generated, a
```

```
        // `TransferError.signingError` is thrown.

        let putConfig = try await S3Client.S3ClientConfiguration(
            maxAttempts: 6,
            region: region
        )

        do {
            let url = try await PutObjectInput(
                body: dataStream,
                bucket: bucket,
                key: fileName
            ).presignURL(
                config: putConfig,
                expiration: TimeInterval(10 * 60)
            )

            guard let url = url else {
                throw TransferError.signingError
            }
            presignedURL = url
        } catch {
            throw TransferError.signingError
        }

        // Send the HTTP request and upload the file to Amazon S3.

        var request = HTTPClientRequest(url: presignedURL.absoluteString)
        request.method = .PUT
        request.body = .bytes(fileData)

        _ = try await HTTPClient.shared.execute(request, timeout: .seconds(5*60))
```

This creates a new `S3ClientConfiguration` struct with the value of `maxAttempts` set to 6, then creates a `PutObjectInput` struct which is used to generate an URL that's presigned using the custom configuration. The presigned URL is a standard Foundation `URL` object.

To use the `AsyncHTTPClient` package to send the data to Amazon S3, an `HTTPClientRequest` is created using the presigned URL as the destination URL. The request's method is set to `.PUT`, indicating that it's an upload request. Then the request body is set to the contents of `fileData`, which contains the `Data` to be sent to Amazon S3.

Finally, the request is executed using the shared `HTTPClient` managed by `AsyncHTTPClient`.

Multipart Amazon S3 uploads using AWS SDK for Swift

Overview

Multipart upload provides a way to upload a single large object in multiple parts, which are combined by Amazon S3 when the upload or copy is complete. This can improve performance because it lets multiple parts be uploaded in parallel, allows pause and resume of uploads, and provides better performance when handling errors since only the part or parts that failed need to be re-uploaded. See Uploading and copying objects using multipart upload in *Amazon Simple Storage Service User Guide* for more information.

The multipart upload process

> **Note**
>
> This section is only a brief review of the multipart upload process. For a more detailed look at the process, including more information about limitations and additional capabilities, see Multipart upload process in *Amazon Simple Storage Service User Guide*.

Fundamentally, multipart upload involves three steps: initiating the upload, uploading the object's parts, and completing the upload once all of the parts have been uploaded. Parts can be uploaded in any order. When uploading each part, you assign it a part number corresponding to the order in which it belongs in the fully assembled file. When received by Amazon S3, each part is assigned an ETag, which is returned to you. The client and the server use the ETag and part number to match the data to the position it occupies in the object during the process of reassembling the content into the complete uploaded object.

This example can be found in its entirety on GitHub.

Start a multipart upload

The first step is to call the SDK for Swift function `S3Client.createMultipartUpload(input:)` with the name of the bucket to store the object into and the key (name) for the new object.

```
    /// Start a multi-part upload to Amazon S3.
    /// - Parameters:
    ///   - bucket: The name of the bucket to upload into.
```

```
    ///   - key: The name of the object to store in the bucket.
    ///
    /// - Returns: A string containing the `uploadId` of the multi-part
    ///   upload job.
    ///
    /// - Throws:
    func startMultipartUpload(client: S3Client, bucket: String, key: String) async
throws -> String {
        let multiPartUploadOutput: CreateMultipartUploadOutput

        // First, create the multi-part upload.

        do {
            multiPartUploadOutput = try await client.createMultipartUpload(
                input: CreateMultipartUploadInput(
                    bucket: bucket,
                    key: key
                )
            )
        } catch {
            throw TransferError.multipartStartError
        }

        // Get the upload ID. This needs to be included with each part sent.

        guard let uploadID = multiPartUploadOutput.uploadId else {
            throw TransferError.uploadError("Unable to get the upload ID")
        }

        return uploadID
    }
```

The output from `createMultipartUpload(input:)` is a string which uniquely identifies the upload that has been started. This ID is used for each call to `uploadPart(input:)` to match the uploaded part to a particular upload. Since each upload has a unique ID, you can have multiple multipart uploads in progress at the same time.

Upload the parts

After creating the multipart upload, upload the numbered parts by calling `S3Client.uploadPart(input:)` once for each part. Each part (except the last) must be at least 5 MB in size.

```
            for partNumber in 1...blockCount {
                let data: Data
                let startIndex = UInt64(partNumber - 1) * UInt64(blockSize)

                // Read the block from the file.

                data = try readFileBlock(file: fileHandle, startIndex: startIndex,
size: blockSize)

                // Upload the part to Amazon S3 and append the `CompletedPart` to
                // the array `completedParts` for use after all parts are uploaded.

                let completedPart = try await uploadPart(
                    client: s3Client, uploadID: uploadID,
                    bucket: bucket, key: fileName,
                    partNumber: partNumber, data: data
                )
                completedParts.append(completedPart)

                let percent = Double(partNumber) / Double(blockCount) * 100
                print(String(format: " %.1f%%", percent))
            }

            .
            .
            .

    /// Upload the specified data as part of an Amazon S3 multi-part upload.
    ///
    /// - Parameters:
    ///   - client: The S3Client to use to upload the part.
    ///   - uploadID: The upload ID of the multi-part upload to add the part to.
    ///   - bucket: The name of the bucket the data is being written to.
    ///   - key: A string giving the key which names the Amazon S3 object the file is
being added to.
    ///   - partNumber: The part number within the file that the specified data
represents.
    ///   - data: The data to send as the specified object part number in the object.
    ///
    /// - Throws: `TransferError.uploadError`
    ///
    /// - Returns: A `CompletedPart` object describing the part that was uploaded.
    ///   contains the part number as well as the ETag returned by Amazon S3.
```

```
    func uploadPart(client: S3Client, uploadID: String, bucket: String,
                    key: String, partNumber: Int, data: Data)
                    async throws -> S3ClientTypes.CompletedPart {
        let uploadPartInput = UploadPartInput(
            body: ByteStream.data(data),
            bucket: bucket,
            key: key,
            partNumber: partNumber,
            uploadId: uploadID
        )

        do {
        let uploadPartOutput = try await client.uploadPart(input: uploadPartInput)
            guard let eTag = uploadPartOutput.eTag else {
                throw TransferError.uploadError("Missing eTag")
            }

            return S3ClientTypes.CompletedPart(eTag: eTag, partNumber: partNumber)
        } catch {
            throw TransferError.uploadError(error.localizedDescription)
        }
    }
```

This example iterates over chunks of the file, reading the data from the source file then uploading it with the corresponding part number. The response includes the ETag assigned to the newly uploaded part. This and the part number are used to create an `S3ClientTypes.CompletedPart` record matching the ETag to the part number, and this record is added to the array of completed part records. This will be needed when all the parts have been uploaded and it's time to call `S3Client.completeMultipartUpload(input:)`.

Complete a multipart upload

Once all parts have been uploaded, call the `S3Client` function `completeMultipartUpload(input:)`. This takes as input the names of the bucket and key for the object, the upload ID string, and the array that matches each of the object's part numbers to the corresponding ETags, as created in the section above.

```
    /// Complete a multi-part upload using an array of `CompletedMultipartUpload`
    /// objects describing the completed parts.
    ///
    /// - Parameters:
```

```
    ///   - client: The S3Client to finish uploading with.
    ///   - uploadId: The multi-part upload's ID string.
    ///   - bucket: The name of the bucket the upload is targeting.
    ///   - key: The name of the object being written to the bucket.
    ///   - parts: An array of `CompletedPart` objects describing each part
    ///     of the upload.
    ///
    /// - Throws: `TransferError.multipartFinishError`
    func finishMultipartUpload(client: S3Client, uploadId: String, bucket: String, key:
String,
                               parts: [S3ClientTypes.CompletedPart]) async throws {
        do {
            let partInfo = S3ClientTypes.CompletedMultipartUpload(parts: parts)
            let multiPartCompleteInput = CompleteMultipartUploadInput(
                bucket: bucket,
                key: key,
                multipartUpload: partInfo,
                uploadId: uploadId
            )
            _ = try await client.completeMultipartUpload(input: multiPartCompleteInput)
        } catch {
            dump(error)
            throw TransferError.multipartFinishError(error.localizedDescription)
        }
    }
```

Additional information

- Uploading and copying objects using multipart upload

Use waiters with the AWS SDK for Swift

A waiter is a client-side abstraction that automatically polls a resource until a desired state is reached, or until it's determined that the resource won't enter that state. This kind of polling is commonly used when working with services that typically reach a consistent state, such as Amazon Simple Storage Service (Amazon S3), or services that create resources asynchronously, like Amazon Elastic Compute Cloud (Amazon EC2).

Instead of writing logic to continuously poll an AWS resource, which can be cumbersome and error-prone, you can use a waiter to poll the resource. When the waiter returns, your code knows whether or not the desired state was reached and can act accordingly.

How to use waiters

Many service client classes in the AWS SDK for Swift provide waiters for common "poll and wait" situations that occur while using AWS services. These situations can include both waiting until a resource is available and waiting until the resource becomes unavailable. For example:

`S3Client.waitUntilBucketExists(options:input:)`, **`S3Client.waitUntilBucketNotExists(options:input:)`**

Wait until an Amazon S3 bucket exists (or no longer exists).

`DynamoDBClient.waitUntilTableExists(options:input:)`, **`DynamoDBClient.waitUntilTableNotExists(options:input:)`**

Wait until a specific Amazon DynamoDB table exists (or no longer exists).

Waiter functions in the AWS SDK for Swift take two parameters, `options` and `input`.

Waiter options

The first parameter for any waiter function, `options`, is a structure of type `WaiterOptions` (part of the `ClientRuntime` package) that describes the waiter's polling behavior. These options specify the maximum number of seconds to wait before polling times out, and let you optionally set the minimum and maximum delays between retries.

The following example shows how to configure a waiter to wait between a tenth of a second and a half second between retries. The maximum polling time is two seconds.

```
let options = WaiterOptions(maxWaitTime: 2, minDelay: 0.1, maxDelay: 0.5)
```

Waiter parameters

A waiter function's inputs are specified using its `input` parameter. The input's data type corresponds to that of the SDK for Swift function used internally by the waiter to poll the AWS service. For example, the type is `HeadBucketInput` for Amazon S3 waiters like `S3Client.waitUntilBucketExists(options:input:)` because this waiter uses the `S3Client.headBucket(input:)` function to poll the bucket. The DynamoDB waiter `DynamoDBClient.waitUntilTableExists(options:input:)` takes as its input a structure of type `DescribeTableInput` because it calls `DynamoDBClient.describeTable(input:)` internally.

Example: Wait for an S3 bucket to exist

The AWS SDK for Swift offers several waiters for Amazon S3. One of them is `waitUntilBucketExists(options:input:)`, which polls the server until the specified bucket exists.

```
    /// Wait until a bucket with the specified name exists, then return
    /// to the caller. Times out after 60 seconds. Throws an error if the
    /// wait fails.
    ///
    /// - Parameter bucketName: A string giving the name of the bucket
    ///   to wait for.
    ///
    /// - Returns: `true` if the bucket was found or `false` if not.
    ///
    public func waitForBucket(name bucketName: String) async throws -> Bool {
        // Because `waitUntilBucketExists()` internally uses the Amazon S3
        // action `HeadBucket` to look for the bucket, the input is specified
        // with a `HeadBucketInput` structure.

        let output = try await client.waitUntilBucketExists(
            options: WaiterOptions(maxWaitTime: 60.0),
            input: HeadBucketInput(bucket: bucketName)
        )

        switch output.result {
            case .success:
                return true
            case .failure:
                return false
        }
    }
```

This example creates a function that waits until the specified bucket exists, then returns `true`. It returns `false` if polling fails, and might throw an exception if an error occurs. It works by calling the waiter function `S3Client.waitUntilBucketExists(options:input:)` to poll the server. The `options` specify that polling should time out after 60 seconds.

Because this polling is done using the Amazon S3 action `HeadBucket`, a `HeadBucketInput` object is created with the input parameters for that operation, including the name of the bucket to poll for. This is used as the `input` parameter's value.

`S3Client.waitUntilBucketExists(options:input:)` returns a `result` property whose value is either `.success` if the polling successfully found the bucket, or `.failure` if polling failed. The function returns the corresponding Boolean value to the caller.

Configure retry using the AWS SDK for Swift

Calls to AWS services occasionally encounter problems or unexpected situations. Certain types of errors, such as throttling or transient errors, might be successful if the call is retried.

This page describes how to configure automatic retries with the AWS SDK for Swift.

Default retry configuration

By default, every service client is automatically configured with a standard retry strategy. The default configuration tries each action up to three times (the initial attempt plus two retries). The intervening delay between each call is configured with exponential backoff and random jitter to avoid retry storms. This configuration works for the majority of use cases but may be unsuitable in some circumstances, such as high-throughput systems.

The SDK attempts retries only on retryable errors. Examples of retryable errors are socket timeouts, service-side throttling, concurrency or optimistic lock failures, and transient service errors. Missing or invalid parameters, authentication/security errors, and misconfiguration exceptions are not considered retryable.

Configure retry

You can customize the standard retry strategy by setting the maximum number of attempts and the rate limiting strategy to use. To change the retry configuration, create a structure of type `S3ClientConfiguration` with the properties you wish to customize. For details on how to configure a service client, see Customize AWS service client configurations.

Maximum number of attempts

You can customize the maximum number of attempts by specifying a value for the `S3ClientConfiguration` structure's `maxAttempts` property. The default value is 3.

Retry mode

The retry mode can be set by changing the `S3ClientConfiguration` structure's `retryMode`. The available values are provided by the `enum` `AWSRetryMode`.

Legacy

The `legacy` retry mode—the default—is the original standard retry mode. In this mode, requests may be sent immediately, and are not delayed for rate limiting when throttling is detected. Requests are only delayed according to the backoff strategy in use, which is exponentially by default.

Standard

In the AWS SDK for Swift, the `standard` mode is the same as legacy mode.

Adaptive

In the `adaptive` retry mode, initial and retry requests may be delayed by an additional amount when throttling is detected. This is intended to reduce congestion when throttling is in effect. This is sometimes called "client-side rate limiting" mode, and is available opt-in. You should only use adaptive mode when advised to do so by an AWS representative.

Example

First, import the modules needed to configure retry:

```
import AWSS3
import SmithyRetries
import SmithyRetriesAPI
```

Then create the custom configuration. This example's `S3ClientConfiguration` asks for the client to make up to three attempts for each action, using the `adaptive` retry mode.

```
        let config: S3Client.S3ClientConfiguration

        // Create an Amazon S3 client configuration object that specifies the
        // adaptive retry mode and sets the maximum number of attempts to 3.
        // If that fails, create a default configuration instead.

        do {
            config = try await S3Client.S3ClientConfiguration(
                awsRetryMode: .adaptive,
                maxAttempts: 3
            )
        } catch {
```

```
        do {
            config = try await S3Client.S3ClientConfiguration()
        } catch {
            print("Error: Unable to configure Amazon S3.")
            dump(error)
            return
        }
    }

    // Create an Amazon S3 client using the configuration created above.

    let client = S3Client(config: config)
```

It first attempts to create the configuration using the application's preferred settings. If that fails, a default configuration is created instead. If that also fails, the example outputs an error message. A real-world application might instead present an error message and let the user decide what to do.

Handle AWS SDK for Swift errors

Overview

The AWS SDK for Swift uses Swift's standard error handling mechanism to report errors that occur while using AWS services. Errors are reported using Swift's `throw` statement.

To catch the errors that an AWS SDK for Swift function might return, use the Swift `do`/`catch` statement. Encapsulate the code that calls the SDK inside the `do` block, then use one or more `catch` blocks to capture and handle the errors. Each `catch` block can capture a specific error, all errors of a specific category, or all uncaught errors. This lets an application recover from errors it knows how to handle, notify the user of transient errors or errors that can't be recovered from but are non-fatal, and safely exit the program if the error is fatal.

> **Note**
>
> The APs for each service client are generated using models specified using the Smithy interface definition language (IDL). The Smithy models describe the underlying types and APIs of each AWS service. These models are used to generate the Swift types and classes that comprise the SDK. This is useful to understand both while reading this guide and while writing error handling code.

AWS SDK for Swift error protocols

SDK for Swift errors conform to one or more error protocols. The protocols implemented by the error depend on the type of error that occurred and the context in which it occurred.

The `Error` protocol

Every error thrown by the AWS SDK for Swift conforms to the standard Swift `Error` protocol. As such, every error has a `localizedDescription` property that returns a string containing a useful description of the error.

When the underlying AWS service provides an error message, that string is used as the `localizedDescription`. These are usually in English. Otherwise, the SDK generates an appropriate message, which may or may not be localized.

The `AWSServiceError` protocol

When the AWS service responds with a service error, the error object conforms to the `AWSClientRuntime.AWSServiceError` protocol.

> **Note**
>
> If an `AWSServiceError` occurs while performing a service action over an HTTP connection, the error also implements the `HTTPError` protocol. Currently, all AWS protocols use HTTP, but if this were to change, an appropriate error protocol would be added.

Errors that conform to `AWSServiceError` include these additional properties:

`errorCode`

An `optional` string identifying the error type.

`requestID`

An `optional` string that gives the request ID of the request that resulted in the error.

The `ModeledError` protocol

When an error occurs that matches a defined, modeled error type, the error object conforms to the protocol `ClientRuntime.ModeledError`, in addition to any other appropriate protocols such as `HTTPError`. This includes most of the errors defined by an AWS service.

`ModeledError` adds several useful properties to the error:

`fault`

A value from the `ClientRuntime.ErrorFault` enum. The value is `.client` if the source of the error is the client, or `.server` if the server is the source of the error.

`isRetryable`

A Boolean value indicating whether or not the model indicates that the failed operation can be retried.

`isThrottling`

A Boolean value indicating whether or not the model indicates that the error is due to throttling.

The `HTTPError` protocol

Errors that occur during an action that uses an HTTP connection conform to the `ClientRuntime.HTTPError` protocol. An error conforming to `HTTPError` contains an HTTP response whose status code is in either the 4xx range or the 5xx range.

`HTTPError` adds one property to the error:

`httpResponse`

An object of type `HttpResponse`, which describes the entire HTTP response from the AWS service. It has properties that include the response's headers, body, and the HTTP status code.

Handling errors

All errors returned by the SDK for Swift implement the standard Swift `Error` protocol. The error's type depends on the service and the error being reported, so it could be any Swift type including but not limited to `enum`, `struct`, or `class`, depending on what kind of error occurred.

For example, an error reporting that an Amazon S3 bucket is missing may conform to `Error`, `AWSServiceError`, and `HTTPError`. This lets you know it's a service error that occurred while communicating using the HTTP protocol. In this case, the HTTP status code is 404 (Not Found), because of the missing bucket.

Even if no other information is provided, the error's `localizedDescription` property is always a string describing the error.

When catching errors thrown by the AWS SDK for Swift, follow these guidelines:

- If the error is modeled, the error is a `struct` describing the error. Catch these errors using that `struct`'s name. In many cases, you can find these modeled errors listed in the documentation of an action in the AWS SDK for Swift API Reference.
- If the error isn't modeled, but still originates from an AWS service, it will conform to the protocol `AWSServiceError`. Use `catch let error as AWSServiceError`, then look at the error's `errorCode` property to determine what error occurred.
- **Don't catch any concrete types that represent unknown errors**, such as `UnknownAWSHTTPServiceError`. These are reserved for internal use.

Service errors

An error thrown because of an AWS service response, whether it could be parsed or not, conforms to the `AWSServiceError` protocol. An error defined by the underlying Smithy model for service also conforms to `ModeledError` and has a concrete type. One example is the Amazon S3 error `CreateBucketOutputError`, which is thrown by the `S3Client.CreateBucket()` method.

Any `AWSServiceError` received over an HTTP connection also conforms to `HTTPError`. This is currently all service errors, but that could change in the future if a service adds support for other network protocols.

The following code tries to create an object on Amazon S3, with code to handle service errors. It features a `catch` clause that specifically handles the error code `NoSuchBucket`, which indicates that the bucket doesn't exist. This snippet assumes that the given bucket name doesn't exist.

```
        do {
            let client = try S3Client(region: "us-east-1")

            _ = try await client.putObject(input: PutObjectInput(
```

```
                body: ByteStream.data(Data(body.utf8)),
                bucket: bucketName,
                key: objectKey
            ))
            print("Done.")
        } catch let error as AWSServiceError {
            let errorCode = error.errorCode ?? "<none>"
            let message = error.message ?? "<no message>"

            switch errorCode {
            case "NoSuchBucket":
                print("   | The bucket \"\(bucketName)\" doesn't exist. This is the
expected result.")
                print("   | In a real app, you might ask the user whether to use a
different name or")
                print("   | create the bucket here.")
            default:
                print("   | Service error of type \(error.errorCode ?? "<unknown>"):
\(message)")
            }
        } catch {
            print("Some other error occurred.")
        }
```

HTTP errors

When the SDK encounters an error while communicating with an AWS service over HTTP, it throws an error that conforms to the protocol `ClientRuntime.HTTPError`. This kind of error represents an HTTP response whose status codes are in the 4xx and 5xx ranges.

> **Note**
>
> Currently, HTTP is the only network protocol used by AWS. If a future AWS product uses a non-HTTP network protocol, a corresponding error protocol would be added to the SDK. Errors that occur while using the new wire protocol would conform to that Swift protocol instead of `HTTPError`.

`HTTPError` includes an `httpResponse` property that contains an object of the class `HttpResponse`. The `httpResponse` provides information received in the response to the failed HTTP request. This provides access to the response headers, including the HTTP status code.

```
        do {
            let client = try S3Client(region: "us-east-1")

            _ = try await client.getObject(input: GetObjectInput(
                bucket: "not-a-real-bucket",
                key: "not-a-real-key"
            ))
            print("   | Found a matching bucket but shouldn't have!")
        } catch let error as HTTPError {
            print("   | HTTP error; status code:
 \(error.httpResponse.statusCode.rawValue). This is the")
            print("   | expected result.")
        } catch {
            dump(error, name: "   | An unexpected error occurred.")
        }
```

This example creates an Amazon S3 client, then calls its `getObject(input:)` function to fetch an object using a bucket name and key that don't exist. Two `catch` blocks are used. The first matches errors of type `HTTPError`. It retrieves the HTTP status code from the response. The status code can then be used to handle specific scenarios, recover from recoverable errors, or whatever the project requires.

The second `catch` block is a catch-all that just dumps the error to the console. In a full application, this block would ideally either clean up after the failed access attempt and return the application to a safe state, or perform as clean an application exit as possible.

Handling other errors

To catch any errors not already caught for a given do block, use the `catch` keyword with no qualifiers. The following snippet simply catches all errors.

```
do {
    let s3 = try await S3Client()

    // ...
} catch {
    // Handle the error here.
}
```

Within the context of the `catch` block, the caught error, reported in the constant with the default name `error`, conforms to at least the standard Swift `Error` type. It may also conform to a combination of the other AWS SDK for Swift error protocols.

If you use a catch-all like this in your code, it needs to safely stop whatever task it was trying to perform and clean up after itself. In extreme cases, it may be necessary to safely terminate the application and ideally provide diagnostic output to be relayed to the developer.

While developing a project, it can be helpful to temporarily output error details to the console. This can be useful when debugging, or to help determine which errors that occur may need special handling. The Swift `dump()` function can be used to do this.

```
do {
    let output = try await client.listBuckets(input: ListBucketsInput())

    // ...
} catch {
    dump(error, name: "Getting the bucket list")
}
```

The `dump()` function outputs the entire contents of the `error` to the console. The `name` argument is an `optional` string used as a label for the output, to help identify the source of the error in the program's output.

Testing and debugging

Logging

The AWS SDK for Swift and the underlying Common RunTime (CRT) library use Apple's SwiftLog mechanism to log informational and debugging messages. The output from the logging system appears in the console provided by most IDEs, and also in the standard system console. This section covers how to control the output level for both the SDK for Swift and the CRT library.

Configure SDK debugging

By default, the SDK's logging system emits logs containing trace and debug level information. The default log level is `info`. To see more informational text as the SDK works, you can change the log level to `error` as shown in the following example:

```
import ClientRuntime
```

```
await SDKLoggingSystem().initialize(logLevel: .error)
```

Call `SDKLoggingSystem.initialize(logLevel:)` no more than one time in your application, to set the log level cutoff.

The log levels supported by the AWS SDK for Swift are defined in the `SDKLogLevel enum`. These correspond to similarly-named log levels defined in SwiftLog. Each log level is inclusive of the messages at and more severe that level. For example, setting the log level to `warning` also causes log messages to be output for levels `error` and `critical`.

The SDK for Swift log levels (from least severe to most severe) are:

- `.trace`
- `.debug`
- `.info` (the default)
- `.notice`
- `.warning`
- `.error`
- `.critical`

Configure Common RunTime logging

If the level of detail generated by the SDK logs isn't providing what you need, you can try configuring the Common RunTime (CRT) log level. CRT is responsible for the lower-level networking and other system interactions performed by the SDK. Its log output includes details about HTTP traffic, for example.

To set CRT's log level to debug:

```
import ClientRuntime

SDKDefaultIO.shared.setLogLevel(level: .error)
```

The CRT log levels (from least severe to most severe) are:

- `.none`

- `.trace`
- `.debug`
- `.info`
- `.warn`
- `.error`
- `.fatal`

Setting the CRT log level to `trace` provides an enormous level of detail that can take some effort to read, but it can be useful in demanding debugging situations.

Mock the AWS SDK for Swift

When writing unit tests for your AWS SDK for Swift project, it's useful to be able to mock the SDK. Mocking is a technique for unit testing in which external dependencies — such as the SDK for Swift — are replaced with code that simulates those dependencies in a controlled and predictable way. Mocking the SDK removes network requests, which eliminates the chance that tests can be unreliable due to intermittent network issues.

In addition, well-written mocks are almost always faster than the operations they simulate, letting you test more thoroughly in less time.

The Swift language doesn't provide the read/write reflection needed for direct mocking. Instead, you adapt your code to allow an indirect form of mocking. This section describes how to do so with minimal changes to the main body of your code.

To mock the AWS SDK for Swift implementation of a service class, create a protocol. In the protocol, define each of that class's functions that you need to use. This serves as the abstraction layer that you need to implement mocking. It's up to you whether to use a separate protocol for each AWS service class used in your project. Alternatively, you can use a single protocol that encapsulates every SDK function that you call. The examples in this guide use the latter approach, but this is often the same as using a protocol for each service.

After you define the protocol, you need two classes that conform to the protocol: one class in which each function calls through to the corresponding SDK function, and one that mocks the results as if the SDK function was called. Because these two classes both conform to the same protocol, you can create functions that perform AWS actions by calling functions on an object conforming to the protocol.

Example: Mock an Amazon S3 function

Consider a program that needs to use the `S3Client` function `listBuckets(input:)`. To support mocking, this project implements the following:

- `S3SessionProtocol`, a Swift protocol which declares the Amazon S3 functions used by the project. This example uses just one Amazon S3 function: `listBuckets(input:)`.
- `S3Session`, a class conforming to `S3SessionProtocol`, whose implementation of `listBuckets(input:)` calls `S3Client.listBuckets(input:)`. This is used when running the program normally.
- `MockS3Session`, a class conforming to `S3SessionProtocol`, whose implementation of `listBuckets(input:)` returns mocked results based on the input parameters. This is used when running tests.
- `BucketManager`, a class that implements access to Amazon S3. This class should accept an object conforming to the session protocol `S3SessionProtocol` during initialization, then perform all AWS requests by making calls through that object. This makes the code testable: the *application* initializes the class by using an `S3Session` object for AWS access, while *tests* use a `MockS3Session` object.

The rest of this section takes an in-depth look at this implementation of mocking. The complete example is available on GitHub.

Protocol

In this example, the `S3SessionProtocol` protocol declares the one `S3Client` function that it needs:

```
/// The S3SessionProtocol protocol describes the Amazon S3 functions this
/// program uses during an S3 session. It needs to be implemented once to call
/// through to the corresponding SDK for Swift functions, and a second time to
/// instead return mock results.
public protocol S3SessionProtocol {
    func listBuckets(input: ListBucketsInput) async throws
            -> ListBucketsOutput
}
```

This protocol describes the interface by which the pair of classes perform Amazon S3 actions.

Main program implementation

To let the main program make Amazon S3 requests using the session protocol, you need an implementation of the protocol in which each function calls the corresponding SDK function. In this example, you create a class named `S3Session` with an implementation of `listBuckets(input:)` that calls `S3Client.listBuckets(input:)`:

```
public class S3Session: S3SessionProtocol {
    let client: S3Client
    let region: String

    /// Initialize the session to use the specified AWS Region.
    ///
    /// - Parameter region: The AWS Region to use. Default is `us-east-1`.
    init(region: String = "us-east-1") throws {
        self.region = region

        // Create an ``S3Client`` to use for AWS SDK for Swift calls.
        self.client = try S3Client(region: self.region)
    }

    /// Call through to the ``S3Client`` function `listBuckets()`.
    ///
    /// - Parameter input: The input to pass through to the SDK function
    ///   `listBuckets()`.
    ///
    /// - Returns: A ``ListBucketsOutput`` with the returned data.
    ///
    public func listBuckets(input: ListBucketsInput) async throws
            -> ListBucketsOutput {
        return try await self.client.listBuckets(input: input)
    }
}
```

The initializer creates the underlying `S3Client` through which the SDK for Swift is called. The only other function is `listBuckets(input:)`, which returns the result of calling the `S3Client` function of the same name. Calls to AWS services work the same way they do when calling the SDK directly.

Mock implementation

In this example, add support for mocking calls to Amazon S3 by using a second implementation of `S3SessionProtocol` called `MockS3Session`. In this class, the `listBuckets(input:)` function generates and returns mock results:

```
    /// An implementation of the Amazon S3 function `listBuckets()` that
    /// returns the mock data instead of accessing AWS.
    ///
    /// - Parameter input: The input to the `listBuckets()` function.
    ///
    /// - Returns: A `ListBucketsOutput` object containing the list of
    ///   buckets.
    public func listBuckets(input: ListBucketsInput) async throws
            -> ListBucketsOutput {
        let response = ListBucketsOutput(
            buckets: self.mockBuckets,
            owner: nil
        )
        return response
    }
```

This works by creating and returning a `ListBucketsOutput` object, like the actual `S3Client` function does. Unlike the SDK function, this makes no actual AWS service requests. Instead, it fills out the response object with data that simulates actual results. In this case, an array of `S3ClientTypes.Bucket` objects describing a number of mock buckets is returned in the `buckets` property.

Not every property of the returned response object is filled out in this example. The only properties that get values are those that always contain a value and those actually used by the application. Your project might require more detailed results in its mock implementations of functions.

Encapsulate access to AWS services

A convenient way to use this approach in your application design is to create an access manager class that encapsulates all your SDK calls. For example, when using Amazon DynamoDB (DynamoDB) to manage a product database, create a `ProductDatabase` class that has functions to perform needed activities. This might include adding products and searching for products. This Amazon S3 example has a class that handles bucket interactions, called `BucketManager`.

The `BucketManager` class initializer needs to accept an object conforming to `S3SessionProtocol` as an input. This lets the caller specify whether to interact with AWS by using actual SDK for Swift calls or by using a mock. Then, every other function in the class that uses AWS actions should use that session object to do so. This lets `BucketManager` use actual SDK calls or mocked ones based on whether testing is underway.

With this in mind, the `BucketManager` class can now be implemented. It needs an `init(session:)` initializer and a `getBucketNames(input:)` function:

```
public class BucketManager {
    /// The object based on the ``S3SessionProtocol`` protocol through which to
    /// call SDK for swift functions. This may be either ``S3Session`` or
    /// ``MockS3Session``.
    var session: S3SessionProtocol

    /// Initialize the ``BucketManager`` to call Amazon S3 functions using the
    /// specified object that implements ``S3SessionProtocol``.
    ///
    /// - Parameter session: The session object to use when calling Amazon S3.
    init(session: S3SessionProtocol) {
        self.session = session
    }

    /// Return an array listing all of the user's buckets by calling the
    /// ``S3SessionProtocol`` function `listBuckets()`.
    ///
    /// - Returns: An array of bucket name strings.
    ///
    public func getBucketNames() async throws -> [String] {
        let output = try await session.listBuckets(input: ListBucketsInput())

        guard let buckets = output.buckets else {
            return []
        }

        return buckets.map { $0.name ?? "<unknown>" }
    }
}
```

The `BucketManager` class in this example has an initializer that takes an object that conforms to `S3SessionProtocol` as an input. That session is used to access or simulate access to AWS actions instead of calling the SDK directly, as shown by the `getBucketNames()` function.

Use the access manager in the main program

The main program can now specify an `S3Session` when creating a `BucketManager` object, which directs its requests to AWS:

```
        /// An ``S3Session`` object that passes calls through to the SDK for
        /// Swift.
        let session: S3Session
        /// A ``BucketManager`` object that will be initialized to call the
        /// SDK using the session.
        let bucketMgr: BucketManager

        // Create the ``S3Session`` and a ``BucketManager`` that calls the SDK
        // using it.
        do {
            session = try S3Session(region: "us-east-1")
            bucketMgr = BucketManager(session: session)
        } catch {
            print("Unable to initialize access to Amazon S3.")
            return
        }
```

Write tests using the protocol

Whether you write tests using Apple's XCTest framework or another framework, you must design the tests to use the mock implementation of the functions that access AWS services. In this example, tests use a class of type `XCTestCase` to implement a standard Swift test case:

```
final class MockingTests: XCTestCase {
    /// The session to use for Amazon S3 calls. In this case, it's a mock
    /// implementation.
    var session: MockS3Session? = nil
    /// The ``BucketManager`` that uses the session to perform Amazon S3
    /// operations.
    var bucketMgr: BucketManager? = nil

    /// Perform one-time initialization before executing any tests.
    override class func setUp() {
        super.setUp()
    }

    /// Set up things that need to be done just before each
```

```
    /// individual test function is called.
    override func setUp() {
        super.setUp()

        self.session = MockS3Session()
        self.bucketMgr = BucketManager(session: self.session!)
    }

    /// Test that `getBucketNames()` returns the expected results.
    func testGetBucketNames() async throws {
        let returnedNames = try await self.bucketMgr!.getBucketNames()
        XCTAssertTrue(self.session!.checkBucketNames(names: returnedNames),
                "Bucket names don't match")
    }
}
```

This `XCTestCase` example's per-test `setUp()` function creates a new `MockS3Session`. Then it uses the mock session to create a `BucketManager` that will return mock results. The `testGetBucketNames()` test function tests the `getBucketNames()` function in the bucket manager object. This way, the tests operate using known data, without needing to access the network, and without accessing AWS services at all.

AWS SDK for Swift integration on Apple platforms

Software running on any of Apple's platforms (macOS, iOS, iPadOS, tvOS, visionOS, or watchOS) may wish to integrate into the Apple ecosystem. This may include, for example, letting the user authenticate to access AWS services using Sign In With Apple.

Authenticate using Sign In With Apple

One convenient way for users to sign into AWS while using your application by adding support for Sign In With Apple. This allows your users to access AWS services using their Apple ID and either TouchID or FaceID.

Adding Sign In With Apple support to your app requires planning and configuration of services:

1. Set up your application in Apple's "Certificates, Identifiers & Profiles" dashboard, or configure Sign In With Apple using Xcode.
2. Set up the application's security and authentication configuration in the AWS Management Console.
3. Add the Sign In With Apple authentication method to your application.

> **Note**
>
> When setting up Sign In With Apple as a way to authenticate to use AWS services, the JWT audience should always be the same as your application's bundle ID as configured in Xcode (or the `Info.plist` file) and the Apple developer portal.

The rest of this section discusses the process of setting up and using Sign In With Apple for AWS authentication. The example authenticates to AWS using Sign In With Apple, then lists the user's Amazon Simple Storage Service (Amazon S3) buckets in a SwiftUI view. The example works on macOS, iOS, and iPadOS and is shown in part during the walkthrough below. The complete example is on GitHub.

Configure the app for Sign in With Apple

If you use Xcode, you can enable Sign In With Apple in the options for your application's main target.

1. Verify that your team name and bundle identifier are correct for the application. Be sure to use Apple's standard reverse-URI notation, such as `com.example.buckets`.
2. Click the **+ Capability** button to open a capability picker. Add Sign In With Apple to your target's capabilities.
3. Click the **Automatically manage signing** checkbox to enable automatic signing of your application. This also configures your application on the Apple developer portal.
4. Under **App Sandbox**, enable **Outgoing Connections (Client)** to have your signed application request network access permission.

If you don't use Xcode, or you prefer to configure the app by hand, you can do so by following the instructions on Apple's developer website.

Configure AWS services for your application

Next, open the IAM Management Console to configure IAM to support authentication using the JSON Web Token (JWT) returned by Sign In With Apple after a sign in request.

1. In the sidebar, click **Identity providers**, then look to see if there's already a provider for `appleid.apple.com`. If there is, click on it to add your new application's bundle ID as a new audience. Otherwise, click **Add provider** to create a new identity provider with the following configuration:
   - Set the provider type to **OpenID Connect**.
   - Set the **Provider URL** to `https://appleid.apple.com`.
   - Set the audience to match your application's bundle ID, such as `com.example.buckets`.
2. If you don't already have a permissions policy set up to grant the permissions needed by your application (and no permissions it doesn't need):
   a. Click **Policies** in the IAM Management Console sidebar, then click **Create policy**.
   b. Add the permissions your application needs, or directly edit the JSON. The JSON for this example application looks like this:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "s3:ListAllMyBuckets",
```

```
            "Resource": ["arn:aws:s3:::*"]
        }
    ]
}
```

This policy allows your application to get a list of your Amazon S3 buckets.

c. On the **Review and create** page, set the **Policy name** to be a unique name to identify your policy, such as `example-buckets-app-policy`.

d. Enter a helpful description for your policy, such as "Application permissions for the Sign In With Apple buckets example".

e. Click **Create policy**.

3. Create a new role for your application:

a. Click **Roles** in the IAM Management Console sidebar, then click **Create role**.

b. Set the trusted entity type to **Web identity**.

c. Under **Web identity**, select the Apple ID identity provider you created above.

d. Select your application's bundle ID as the **Audience**.

e. Under **Permissions policies**, choose the policy you created above.

f. On the **Name review, and create** page, set a unique name for your new role, such as `example-buckets-app-role`, and enter a description for the role, such as "Permissions for the Buckets example."

g. Review the **Trust policy** generated by the console. The JSON should resemble the following:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "sts:AssumeRoleWithWebIdentity",
            "Principal": {
                "Federated": "arn:aws:iam::111122223333:oidc-provider/
appleid.apple.com"
            },
            "Condition": {
                "StringEquals": {
                    "appleid.apple.com:aud": [
                        "com.example.buckets"
                    ]
```

```
                }
            }
        }
    ]
}
```

This indicates that the Apple ID identity provider used to process Sign In With Apple requests should be allowed to use the role if the web token's aud property (the audience) matches the application's bundle ID.

Add Sign In With Apple support to your application

Once both Apple and AWS know about your application and that Sign In With Apple should be allowed to authenticate the user for the desired services, the next step is to add a Sign In With Apple button and its supporting code to the application.

This section explains how an application can provide a Sign In With Apple option for authentication, using Apple's SwiftUI and Authentication Services libraries.

Add a "Sign In With Apple" button

To include a Sign In With Apple button, the `AuthenticationServices` module needs to be imported along with SwiftUI:

```
import SwiftUI
import AuthenticationServices
```

In your SwiftUI sign-in view, embed a Sign In With Apple button. The `SignInWithAppleButton` is used to trigger and manage the Sign In With Apple process. This button calls a completion handler with a `Result` object, which indicates whether or not a valid Apple ID was authenticated:

```
                    // Show the "Sign In With Apple" button, using the
                    // `.continue` mode, which allows the user to create
                    // a new ID if they don't already have one. When SIWA
                    // is complete, the view model's `handleSignInResult()`
                    // function is called to turn the JWT token into AWS
                    // credentials.
                    SignInWithAppleButton(.continue) { request in
                        request.requestedScopes = [.email, .fullName ]
                    } onCompletion: { result in
```

```
                    Task {
                        do {
                            try await viewModel.handleSignInResult(result)
                        } catch BucketsAppError.signInWithAppleCanceled {
                            // The "error" is actually Sign In With Apple being
                            // canceled by the user, so end the sign in
                            // attempt.
                            return
                        } catch let error as BucketsAppError {
                            // Handle AWS errors.
                            viewModel.error = error
                            return
                        }
                    }
                }
```

This example `SignInWithAppleButton` uses a function named `handleSignInResult()` as its completion handler. This function is passed a `Result` that contains an `ASAuthorization` object if Sign In With Apple succeeded. If sign in failed or was canceled, the `Result` contains an `Error` instead. If the error actually indicates that sign in was canceled by the user, the sign in attempt is ended. Actual errors are stored for display by a SwiftUI alert sheet.

> **Note**
>
> By default, the token returned by Sign In With Apple expires one day from its creation time.

Process the JSON Web Token

A number of variables are used by the example application to store information related to the user and their sign-in session:

```
    /// The unique string assigned by Sign In With Apple for this login
    /// session. This ID is valid across application launches until it
    /// is signed out from Sign In With Apple.
    var userID = ""

    /// The user's email address.
    ///
    /// This is only returned by SIWA if the user has just created
    /// the app's SIWA account link. Otherwise, it's returned as `nil`
    /// by SIWA and must be retrieved from local storage if needed.
```

```
    var email = ""

    /// The user's family (last) name.
    ///
    /// This is only returned by SIWA if the user has just created
    /// the app's SIWA account link. Otherwise, it's returned as `nil`
    /// by SIWA and must be retrieved from local storage if needed.
    var familyName = ""

    /// The user's given (first) name.
    ///
    /// This is only returned by SIWA if the user has just created
    /// the app's SIWA account link. Otherwise, it's returned as `nil` by SIWA
    /// and must be retrieved from local storage if needed.
    var givenName = ""

    /// The AWS account number provided by the user.
    var awsAccountNumber = ""

    /// The AWS IAM role name given by the user.
    var awsIAMRoleName = ""

    /// The credential identity resolver created by the AWS SDK for
    /// Swift. This resolves temporary credentials using
    /// `AssumeRoleWithWebIdentity`.
    var identityResolver: STSWebIdentityAWSCredentialIdentityResolver? = nil
```

These are used to record the AWS account information (the AWS account number and the name of the IAM role to use, as entered by the user), as well as the user's information returned by Sign In With Apple. The `STSWebIdentityAWSCredentialIdentityResolver` is used to convert the JWT token into valid, temporary AWS credentials when creating a service client object.

The `handleSignInResult()` function looks like this:

```
    /// Called by the Sign In With Apple button when a JWT token has
    /// been returned by the Sign In With Apple service. This function
    /// in turn handles fetching AWS credentials using that token.
    ///
    /// - Parameters:
    ///   - result: The Swift `Result` object passed to the Sign In
    ///     With Apple button's `onCompletion` handler. If the sign
    ///     in request succeeded, this contains an `ASAuthorization`
    ///     object that contains the Apple ID sign in information.
```

```
    func handleSignInResult(_ result: Result<ASAuthorization, Error>) async throws {
        switch result {
        case .success(let auth):
            // Sign In With Apple returned a JWT identity token. Gather
            // the information it contains and prepare to convert the
            // token into AWS credentials.

            guard let credential = auth.credential as?
ASAuthorizationAppleIDCredential,
                  let webToken = credential.identityToken,
                  let tokenString = String(data: webToken, encoding: .utf8)
            else {
                throw BucketsAppError.credentialsIncomplete
            }

            userID = credential.user

            // If the email field has a value, set the user's recorded email
            // address. Otherwise, keep the existing one.
            email = credential.email ?? self.email

            // Similarly, if the name is present in the credentials, use it.
            // Otherwise, the last known name is retained.
            if let name = credential.fullName {
                self.familyName = name.familyName ?? self.familyName
                self.givenName = name.givenName ?? self.givenName
            }

            // Use the JWT token to request a set of temporary AWS
            // credentials. Upon successful return, the
            // `credentialsProvider` can be used when configuring
            // any AWS service.

            try await authenticate(withWebIdentity: tokenString)
        case .failure(let error as ASAuthorizationError):
            if error.code == .canceled {
                throw BucketsAppError.signInWithAppleCanceled
            } else {
                throw BucketsAppError.signInWithAppleFailed
            }
        case .failure:
            throw BucketsAppError.signInWithAppleFailed
        }
```

```
        // Successfully signed in. Fetch the bucket list.
        do {
            try await self.getBucketList()
        } catch {
            throw BucketsAppError.bucketListMissing
        }
    }
```

If sign in is successful, the `Result` provides details about the authentication returned by Sign In With Apple. The authentication credential contains a JSON Web Token, as well as a user ID which is unique for the current session.

If the authentication represents a new link between Sign In With Apple and this application, it may contain the user's email address and full name. If it does, this function retrieves and stores that information. The email address and user name *will never be provided again* by Sign In With Apple.

> **Note**
>
> Sign In With Apple only includes personally identifiable information (PII) the when the user first associates their Apple ID with your application. For all subsequent connections, Sign In With Apple only provides a unique user ID. If the application needs any of this PII, it's the app's responsibility to securely save it *locally* and *securely*. The example application stores the information in the Keychain.

The JWT is converted to a string, which is passed to a function called `authenticate(withWebIdentity: region:)` to actually create the web identity resolver.

Create the web identity credential identity resolver

The `authenticate(withWebIdentity: region:)` function performs credential identity resolution by first writing the token to a local file, then creating an object of type `STSWebIdentityAWSCredentialIdentityResolver`, specifying the stored web identity token file when doing so. This AWS Security Token Service (AWS STS) credential identity resolver is used when creating service clients.

After authenticating with Sign In With Apple, a function named `saveUserData()` is called to securely store the user's information in the Keychain. This lets the sign in screen automatically fill in the form fields, and lets the application remember the user's email address and name if available.

```
    /// Convert the given JWT identity token string into the temporary
    /// AWS credentials needed to allow this application to operate, as
    /// specified using the Apple Developer portal and the AWS Identity
    /// and Access Management (IAM) service.
    ///
    /// - Parameters:
    ///   - tokenString: The string version of the JWT identity token
    ///     returned by Sign In With Apple.
    ///   - region: An optional string specifying the AWS Region to
    ///     access. If not specified, "us-east-1" is assumed.
    func authenticate(withWebIdentity tokenString: String,
                      region: String = "us-east-1") async throws {
        // If the role is empty, pass `nil` to use the default role for
        // the user.

        let roleARN = "arn:aws:iam::\(awsAccountNumber):role/\(awsIAMRoleName)"

        // Use the AWS Security Token Service (STS) action
        // `AssumeRoleWithWebIdentity` to convert the JWT token into a
        // set of temporary AWS credentials. The first step: write the token
        // to disk so it can be used by the
        // `STSWebIdentityAWSCredentialIdentityResolver`.

        let tokenFileURL = createTokenFileURL()
        let tokenFilePath = tokenFileURL.path
        do {
            try tokenString.write(to: tokenFileURL, atomically: true, encoding: .utf8)
        } catch {
            throw BucketsAppError.tokenFileError()
        }

        // Create an identity resolver that uses the JWT token received
        // from Apple to create AWS credentials.

        do {
            identityResolver = try STSWebIdentityAWSCredentialIdentityResolver(
                region: region,
                roleArn: roleARN,
                roleSessionName: "BucketsExample",
                tokenFilePath: tokenFilePath
            )
        } catch {
            throw BucketsAppError.assumeRoleFailed
```

```
        }

        // Save the user's data securely to local storage so it's available
        // in the future.
        //
        // IMPORTANT: Any potential Personally Identifiable Information _must_
        // be saved securely, such as by using the Keychain or an appropriate
        // encrypting technique.

        saveUserData()
    }
```

> **⚠ Important**
>
> The token is written to disk since the SDK expects to load the token from a file. This token file *must remain in place* until you have finished using the service client. When you're done using the client, delete the token file.

Create a service client using the web identity credential identity resolver

To access an AWS service, create a client configuration object that includes the `awsCredentialIdentityResolver` property. This property's value should be the web identity credential identity resolver created by the `authorize(withWebIdentity: region:)` function:

```
    /// Fetches a list of the user's Amazon S3 buckets.
    ///
    /// The bucket names are stored in the view model's `bucketList`
    /// property.
    func getBucketList() async throws {
        // If there's no identity resolver yet, return without doing anything.
        guard let identityResolver = identityResolver else {
            return
        }

        // Create an Amazon S3 client configuration that uses the
        // credential identity resolver created from the JWT token
        // returned by Sign In With Apple.
        let config = try await S3Client.S3ClientConfiguration(
            awsCredentialIdentityResolver: identityResolver,
            region: "us-east-1"
        )
```

```
        let s3 = S3Client(config: config)

        let output = try await s3.listBuckets(
            input: ListBucketsInput()
        )

        guard let buckets = output.buckets else {
            throw BucketsAppError.bucketListMissing
        }

        // Add the names of all the buckets to `bucketList`. Each
        // name is stored as a new `IDString` for use with the SwiftUI
        // `List`.
        for bucket in buckets {
            self.bucketList.append(IDString(bucket.name ?? "<unknown>"))
        }
    }
```

This function creates an `S3Client` configured to use our web identity credential identity resolver. That client is then used to fetch a list of Amazon S3 buckets. The buckets' names are then added to the bucket list. The SwiftUI `List` view automatically refreshes to display the newly-added bucket names.

SDK for Swift code examples

The code examples in this topic show you how to use the AWS SDK for Swift with AWS.

Basics are code examples that show you how to perform the essential operations within a service.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Some services contain additional example categories that show how to leverage libraries or functions specific to the service.

Services

- Amazon Cognito Identity examples using SDK for Swift
- DynamoDB examples using SDK for Swift
- IAM examples using SDK for Swift
- Amazon S3 examples using SDK for Swift
- AWS STS examples using SDK for Swift
- Amazon Transcribe Streaming examples using SDK for Swift

Amazon Cognito Identity examples using SDK for Swift

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for Swift with Amazon Cognito Identity.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- Actions

Actions

`CreateIdentityPool`

The following code example shows how to use `CreateIdentityPool`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSCognitoIdentity


    /// Create a new identity pool and return its ID.
    ///
    /// - Parameters:
    ///     - name: The name to give the new identity pool.
    ///
    /// - Returns: A string containing the newly created pool's ID, or `nil`
    ///   if an error occurred.
    ///
    func createIdentityPool(name: String) async throws -> String? {
        do {
            let cognitoInputCall = CreateIdentityPoolInput(developerProviderName:
 "com.exampleco.CognitoIdentityDemo",
                                                           identityPoolName: name)

            let result = try await cognitoIdentityClient.createIdentityPool(input:
 cognitoInputCall)
            guard let poolId = result.identityPoolId else {
                return nil
            }

            return poolId
        } catch {
            print("ERROR: createIdentityPool:", dump(error))
            throw error
        }
```

```
    }
```

- For more information, see AWS SDK for Swift developer guide.
- For API details, see CreateIdentityPool in *AWS SDK for Swift API reference*.

`DeleteIdentityPool`

The following code example shows how to use `DeleteIdentityPool`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSCognitoIdentity


    /// Delete the specified identity pool.
    ///
    /// - Parameters:
    ///   - id: The ID of the identity pool to delete.
    ///
    func deleteIdentityPool(id: String) async throws {
        do {
            let input = DeleteIdentityPoolInput(
                identityPoolId: id
            )

            _ = try await cognitoIdentityClient.deleteIdentityPool(input: input)
        } catch {
            print("ERROR: deleteIdentityPool:", dump(error))
            throw error
        }
    }
```

- For more information, see AWS SDK for Swift developer guide.
- For API details, see DeleteIdentityPool in *AWS SDK for Swift API reference*.

`ListIdentityPools`

The following code example shows how to use `ListIdentityPools`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSCognitoIdentity


    /// Return the ID of the identity pool with the specified name.
    ///
    /// - Parameters:
    ///   - name: The name of the identity pool whose ID should be returned.
    ///
    /// - Returns: A string containing the ID of the specified identity pool
    ///   or `nil` on error or if not found.
    ///
    func getIdentityPoolID(name: String) async throws -> String? {
        let listPoolsInput = ListIdentityPoolsInput(maxResults: 25)
        // Use "Paginated" to get all the objects.
        // This lets the SDK handle the 'nextToken' field in
 "ListIdentityPoolsOutput".
        let pages = cognitoIdentityClient.listIdentityPoolsPaginated(input:
 listPoolsInput)

        do {
            for try await page in pages {
                guard let identityPools = page.identityPools else {
                    print("ERROR: listIdentityPoolsPaginated returned nil
 contents.")
                    continue
                }
```

```
                /// Read pages of identity pools from Cognito until one is found
                /// whose name matches the one specified in the `name` parameter.
                /// Return the matching pool's ID.

                for pool in identityPools {
                    if pool.identityPoolName == name {
                        return pool.identityPoolId!
                    }
                }
            }
        } catch {
            print("ERROR: getIdentityPoolID:", dump(error))
            throw error
        }

        return nil
    }
```

Get the ID of an existing identity pool or create it if it doesn't already exist.

```
import AWSCognitoIdentity


    /// Return the ID of the identity pool with the specified name.
    ///
    /// - Parameters:
    ///   - name: The name of the identity pool whose ID should be returned
    ///
    /// - Returns: A string containing the ID of the specified identity pool.
    ///   Returns `nil` if there's an error or if the pool isn't found.
    ///
    public func getOrCreateIdentityPoolID(name: String) async throws -> String? {
        // See if the pool already exists. If it doesn't, create it.

        do {
            guard let poolId = try await getIdentityPoolID(name: name) else {
                return try await createIdentityPool(name: name)
            }

            return poolId
```

```
        } catch {
            print("ERROR: getOrCreateIdentityPoolID:", dump(error))
            throw error
        }
    }
```

- For more information, see AWS SDK for Swift developer guide.
- For API details, see ListIdentityPools in *AWS SDK for Swift API reference*.

DynamoDB examples using SDK for Swift

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for Swift with DynamoDB.

Basics are code examples that show you how to perform the essential operations within a service.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

Basics

Learn the basics

The following code example shows how to:

- Create a table that can hold movie data.
- Put, get, and update a single movie in the table.
- Write movie data to the table from a sample JSON file.
- Query for movies that were released in a given year.

- Scan for movies that were released in a range of years.
- Delete a movie from the table, then delete the table.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

A Swift class that handles DynamoDB calls to the SDK for Swift.

```
import AWSDynamoDB
import Foundation

/// An enumeration of error codes representing issues that can arise when using
/// the `MovieTable` class.
enum MoviesError: Error {
    /// The specified table wasn't found or couldn't be created.
    case TableNotFound
    /// The specified item wasn't found or couldn't be created.
    case ItemNotFound
    /// The Amazon DynamoDB client is not properly initialized.
    case UninitializedClient
    /// The table status reported by Amazon DynamoDB is not recognized.
    case StatusUnknown
    /// One or more specified attribute values are invalid or missing.
    case InvalidAttributes
}

/// A class representing an Amazon DynamoDB table containing movie
/// information.
public class MovieTable {
    var ddbClient: DynamoDBClient?
    let tableName: String

    /// Create an object representing a movie table in an Amazon DynamoDB
    /// database.
    ///
    /// - Parameters:
```

```
    ///   - region: The optional Amazon Region to create the database in.
    ///   - tableName: The name to assign to the table. If not specified, a
    ///     random table name is generated automatically.
    ///
    /// > Note: The table is not necessarily available when this function
    /// returns. Use `tableExists()` to check for its availability, or
    /// `awaitTableActive()` to wait until the table's status is reported as
    /// ready to use by Amazon DynamoDB.
    ///
    init(region: String? = nil, tableName: String) async throws {
        do {
            let config = try await DynamoDBClient.DynamoDBClientConfiguration()
            if let region = region {
                config.region = region
            }

            self.ddbClient = DynamoDBClient(config: config)
            self.tableName = tableName

            try await self.createTable()
        } catch {
            print("ERROR: ", dump(error, name: "Initializing Amazon DynamoDBClient
client"))
            throw error
        }
    }

    ///
    /// Create a movie table in the Amazon DynamoDB data store.
    ///
    private func createTable() async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = CreateTableInput(
                attributeDefinitions: [
                    DynamoDBClientTypes.AttributeDefinition(attributeName: "year",
attributeType: .n),
                    DynamoDBClientTypes.AttributeDefinition(attributeName: "title",
attributeType: .s)
                ],
                keySchema: [
```

```
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "year",
keyType: .hash),
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "title",
keyType: .range)
                ],
                provisionedThroughput: DynamoDBClientTypes.ProvisionedThroughput(
                    readCapacityUnits: 10,
                    writeCapacityUnits: 10
                ),
                tableName: self.tableName
            )
            let output = try await client.createTable(input: input)
            if output.tableDescription == nil {
                throw MoviesError.TableNotFound
            }
        } catch {
            print("ERROR: createTable:", dump(error))
            throw error
        }
    }


    /// Check to see if the table exists online yet.
    ///
    /// - Returns: `true` if the table exists, or `false` if not.
    ///
    func tableExists() async throws -> Bool {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DescribeTableInput(
                tableName: tableName
            )
            let output = try await client.describeTable(input: input)
            guard let description = output.table else {
                throw MoviesError.TableNotFound
            }

            return description.tableName == self.tableName
        } catch {
            print("ERROR: tableExists:", dump(error))
            throw error
```

```
        }
    }


    ///
    /// Waits for the table to exist and for its status to be active.
    ///
    func awaitTableActive() async throws {
        while try (await self.tableExists() == false) {
            do {
                let duration = UInt64(0.25 * 1_000_000_000) // Convert .25 seconds
to nanoseconds.
                try await Task.sleep(nanoseconds: duration)
            } catch {
                print("Sleep error:", dump(error))
            }
        }

        while try (await self.getTableStatus() != .active) {
            do {
                let duration = UInt64(0.25 * 1_000_000_000) // Convert .25 seconds
to nanoseconds.
                try await Task.sleep(nanoseconds: duration)
            } catch {
                print("Sleep error:", dump(error))
            }
        }
    }


    ///
    /// Deletes the table from Amazon DynamoDB.
    ///
    func deleteTable() async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DeleteTableInput(
                tableName: self.tableName
            )
            _ = try await client.deleteTable(input: input)
        } catch {
```

```
            print("ERROR: deleteTable:", dump(error))
            throw error
        }
    }


    /// Get the table's status.
    ///
    /// - Returns: The table status, as defined by the
    ///   `DynamoDBClientTypes.TableStatus` enum.
    ///
    func getTableStatus() async throws -> DynamoDBClientTypes.TableStatus {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DescribeTableInput(
                tableName: self.tableName
            )
            let output = try await client.describeTable(input: input)
            guard let description = output.table else {
                throw MoviesError.TableNotFound
            }
            guard let status = description.tableStatus else {
                throw MoviesError.StatusUnknown
            }
            return status
        } catch {
            print("ERROR: getTableStatus:", dump(error))
            throw error
        }
    }


    /// Populate the movie database from the specified JSON file.
    ///
    /// - Parameter jsonPath: Path to a JSON file containing movie data.
    ///
    func populate(jsonPath: String) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }
```

```
            // Create a Swift `URL` and use it to load the file into a `Data`
            // object. Then decode the JSON into an array of `Movie` objects.

            let fileUrl = URL(fileURLWithPath: jsonPath)
            let jsonData = try Data(contentsOf: fileUrl)

            var movieList = try JSONDecoder().decode([Movie].self, from: jsonData)

            // Truncate the list to the first 200 entries or so for this example.

            if movieList.count > 200 {
                movieList = Array(movieList[...199])
            }

            // Before sending records to the database, break the movie list into
            // 25-entry chunks, which is the maximum size of a batch item request.

            let count = movieList.count
            let chunks = stride(from: 0, to: count, by: 25).map {
                Array(movieList[$0 ..< Swift.min($0 + 25, count)])
            }

            // For each chunk, create a list of write request records and populate
            // them with `PutRequest` requests, each specifying one movie from the
            // chunk. Once the chunk's items are all in the `PutRequest` list,
            // send them to Amazon DynamoDB using the
            // `DynamoDBClient.batchWriteItem()` function.

            for chunk in chunks {
                var requestList: [DynamoDBClientTypes.WriteRequest] = []

                for movie in chunk {
                    let item = try await movie.getAsItem()
                    let request = DynamoDBClientTypes.WriteRequest(
                        putRequest: .init(
                            item: item
                        )
                    )
                    requestList.append(request)
                }

                let input = BatchWriteItemInput(requestItems: [tableName:
requestList])
```

```
                _ = try await client.batchWriteItem(input: input)
            }
        } catch {
            print("ERROR: populate:", dump(error))
            throw error
        }
    }


    /// Add a movie specified as a `Movie` structure to the Amazon DynamoDB
    /// table.
    ///
    /// - Parameter movie: The `Movie` to add to the table.
    ///
    func add(movie: Movie) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Get a DynamoDB item containing the movie data.
            let item = try await movie.getAsItem()

            // Send the `PutItem` request to Amazon DynamoDB.

            let input = PutItemInput(
                item: item,
                tableName: self.tableName
            )
            _ = try await client.putItem(input: input)
        } catch {
            print("ERROR: add movie:", dump(error))
            throw error
        }
    }


    /// Given a movie's details, add a movie to the Amazon DynamoDB table.
    ///
    /// - Parameters:
    ///   - title: The movie's title as a `String`.
    ///   - year: The release year of the movie (`Int`).
    ///   - rating: The movie's rating if available (`Double`; default is
    ///     `nil`).
```

```
    ///   - plot: A summary of the movie's plot (`String`; default is `nil`,
    ///     indicating no plot summary is available).
    ///
    func add(title: String, year: Int, rating: Double? = nil,
             plot: String? = nil) async throws
    {
        do {
            let movie = Movie(title: title, year: year, rating: rating, plot: plot)
            try await self.add(movie: movie)
        } catch {
            print("ERROR: add with fields:", dump(error))
            throw error
        }
    }


    /// Return a `Movie` record describing the specified movie from the Amazon
    /// DynamoDB table.
    ///
    /// - Parameters:
    ///   - title: The movie's title (`String`).
    ///   - year: The movie's release year (`Int`).
    ///
    /// - Throws: `MoviesError.ItemNotFound` if the movie isn't in the table.
    ///
    /// - Returns: A `Movie` record with the movie's details.
    func get(title: String, year: Int) async throws -> Movie {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = GetItemInput(
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                tableName: self.tableName
            )
            let output = try await client.getItem(input: input)
            guard let item = output.item else {
                throw MoviesError.ItemNotFound
            }
```

```
            let movie = try Movie(withItem: item)
            return movie
        } catch {
            print("ERROR: get:", dump(error))
            throw error
        }
    }


    /// Get all the movies released in the specified year.
    ///
    /// - Parameter year: The release year of the movies to return.
    ///
    /// - Returns: An array of `Movie` objects describing each matching movie.
    ///
    func getMovies(fromYear year: Int) async throws -> [Movie] {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = QueryInput(
                expressionAttributeNames: [
                    "#y": "year"
                ],
                expressionAttributeValues: [
                    ":y": .n(String(year))
                ],
                keyConditionExpression: "#y = :y",
                tableName: self.tableName
            )
            // Use "Paginated" to get all the movies.
            // This lets the SDK handle the 'lastEvaluatedKey' property in
"QueryOutput".

            let pages = client.queryPaginated(input: input)

            var movieList: [Movie] = []
            for try await page in pages {
                guard let items = page.items else {
                    print("Error: no items returned.")
                    continue
                }
```

```
                // Convert the found movies into `Movie` objects and return an array
                // of them.

                for item in items {
                    let movie = try Movie(withItem: item)
                    movieList.append(movie)
                }
            }
            return movieList
        } catch {
            print("ERROR: getMovies:", dump(error))
            throw error
        }
    }


    /// Return an array of `Movie` objects released in the specified range of
    /// years.
    ///
    /// - Parameters:
    ///   - firstYear: The first year of movies to return.
    ///   - lastYear: The last year of movies to return.
    ///   - startKey: A starting point to resume processing; always use `nil`.
    ///
    /// - Returns: An array of `Movie` objects describing the matching movies.
    ///
    /// > Note: The `startKey` parameter is used by this function when
    ///   recursively calling itself, and should always be `nil` when calling
    ///   directly.
    ///
    func getMovies(firstYear: Int, lastYear: Int,
                   startKey: [Swift.String: DynamoDBClientTypes.AttributeValue]? =
nil)
        async throws -> [Movie]
    {
        do {
            var movieList: [Movie] = []

            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = ScanInput(
                consistentRead: true,
```

```
                exclusiveStartKey: startKey,
                expressionAttributeNames: [
                    "#y": "year" // `year` is a reserved word, so use `#y` instead.
                ],
                expressionAttributeValues: [
                    ":y1": .n(String(firstYear)),
                    ":y2": .n(String(lastYear))
                ],
                filterExpression: "#y BETWEEN :y1 AND :y2",
                tableName: self.tableName
            )

            let pages = client.scanPaginated(input: input)

            for try await page in pages {
                guard let items = page.items else {
                    print("Error: no items returned.")
                    continue
                }

                // Build an array of `Movie` objects for the returned items.

                for item in items {
                    let movie = try Movie(withItem: item)
                    movieList.append(movie)
                }
            }
            return movieList

        } catch {
            print("ERROR: getMovies with scan:", dump(error))
            throw error
        }
    }


    /// Update the specified movie with new `rating` and `plot` information.
    ///
    /// - Parameters:
    ///   - title: The title of the movie to update.
    ///   - year: The release year of the movie to update.
    ///   - rating: The new rating for the movie.
    ///   - plot: The new plot summary string for the movie.
    ///
```

```
    /// - Returns: An array of mappings of attribute names to their new
    ///   listing each item actually changed. Items that didn't need to change
    ///   aren't included in this list. `nil` if no changes were made.
    ///
    func update(title: String, year: Int, rating: Double? = nil, plot: String? =
nil) async throws
        -> [Swift.String: DynamoDBClientTypes.AttributeValue]?
    {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Build the update expression and the list of expression attribute
            // values. Include only the information that's changed.

            var expressionParts: [String] = []
            var attrValues: [Swift.String: DynamoDBClientTypes.AttributeValue] = [:]

            if rating != nil {
                expressionParts.append("info.rating=:r")
                attrValues[":r"] = .n(String(rating!))
            }
            if plot != nil {
                expressionParts.append("info.plot=:p")
                attrValues[":p"] = .s(plot!)
            }
            let expression = "set \(expressionParts.joined(separator: ", "))"

            let input = UpdateItemInput(
                // Create substitution tokens for the attribute values, to ensure
                // no conflicts in expression syntax.
                expressionAttributeValues: attrValues,
                // The key identifying the movie to update consists of the release
                // year and title.
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                returnValues: .updatedNew,
                tableName: self.tableName,
                updateExpression: expression
            )
            let output = try await client.updateItem(input: input)
```

```
            guard let attributes: [Swift.String: DynamoDBClientTypes.AttributeValue]
 = output.attributes else {
                throw MoviesError.InvalidAttributes
            }
            return attributes
        } catch {
            print("ERROR: update:", dump(error))
            throw error
        }
    }


    /// Delete a movie, given its title and release year.
    ///
    /// - Parameters:
    ///   - title: The movie's title.
    ///   - year: The movie's release year.
    ///
    func delete(title: String, year: Int) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DeleteItemInput(
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                tableName: self.tableName
            )
            _ = try await client.deleteItem(input: input)
        } catch {
            print("ERROR: delete:", dump(error))
            throw error
        }
    }
}
```

The structures used by the MovieTable class to represent movies.

```
import Foundation
import AWSDynamoDB

/// The optional details about a movie.
public struct Details: Codable {
    /// The movie's rating, if available.
    var rating: Double?
    /// The movie's plot, if available.
    var plot: String?
}

/// A structure describing a movie. The `year` and `title` properties are
/// required and are used as the key for Amazon DynamoDB operations. The
/// `info` sub-structure's two properties, `rating` and `plot`, are optional.
public struct Movie: Codable {
    /// The year in which the movie was released.
    var year: Int
    /// The movie's title.
    var title: String
    /// A `Details` object providing the optional movie rating and plot
    /// information.
    var info: Details

    /// Create a `Movie` object representing a movie, given the movie's
    /// details.
    ///
    /// - Parameters:
    ///   - title: The movie's title (`String`).
    ///   - year: The year in which the movie was released (`Int`).
    ///   - rating: The movie's rating (optional `Double`).
    ///   - plot: The movie's plot (optional `String`)
    init(title: String, year: Int, rating: Double? = nil, plot: String? = nil) {
        self.title = title
        self.year = year

        self.info = Details(rating: rating, plot: plot)
    }

    /// Create a `Movie` object representing a movie, given the movie's
    /// details.
    ///
    /// - Parameters:
    ///   - title: The movie's title (`String`).
```

```
    ///   - year: The year in which the movie was released (`Int`).
    ///   - info: The optional rating and plot information for the movie in a
    ///     `Details` object.
    init(title: String, year: Int, info: Details?){
        self.title = title
        self.year = year

        if info != nil {
            self.info = info!
        } else {
            self.info = Details(rating: nil, plot: nil)
        }
    }

    ///
    /// Return a new `MovieTable` object, given an array mapping string to Amazon
    /// DynamoDB attribute values.
    ///
    /// - Parameter item: The item information provided to the form used by
    ///   DynamoDB. This is an array of strings mapped to
    ///   `DynamoDBClientTypes.AttributeValue` values.
    init(withItem item: [Swift.String:DynamoDBClientTypes.AttributeValue]) throws  {
        // Read the attributes.

        guard let titleAttr = item["title"],
              let yearAttr = item["year"] else {
            throw MoviesError.ItemNotFound
        }
        let infoAttr = item["info"] ?? nil

        // Extract the values of the title and year attributes.

        if case .s(let titleVal) = titleAttr {
            self.title = titleVal
        } else {
            throw MoviesError.InvalidAttributes
        }

        if case .n(let yearVal) = yearAttr {
            self.year = Int(yearVal)!
        } else {
            throw MoviesError.InvalidAttributes
        }
```

```
        // Extract the rating and/or plot from the `info` attribute, if
        // they're present.

        var rating: Double? = nil
        var plot: String? = nil

        if infoAttr != nil, case .m(let infoVal) = infoAttr {
            let ratingAttr = infoVal["rating"] ?? nil
            let plotAttr = infoVal["plot"] ?? nil

            if ratingAttr != nil, case .n(let ratingVal) = ratingAttr {
                rating = Double(ratingVal) ?? nil
            }
            if plotAttr != nil, case .s(let plotVal) = plotAttr {
                plot = plotVal
            }
        }

        self.info = Details(rating: rating, plot: plot)
    }

    ///
    /// Return an array mapping attribute names to Amazon DynamoDB attribute
    /// values, representing the contents of the `Movie` record as a DynamoDB
    /// item.
    ///
    /// - Returns: The movie item as an array of type
    ///   `[Swift.String:DynamoDBClientTypes.AttributeValue]`.
    ///
    func getAsItem() async throws ->
[Swift.String:DynamoDBClientTypes.AttributeValue]  {
        // Build the item record, starting with the year and title, which are
        // always present.

        var item: [Swift.String:DynamoDBClientTypes.AttributeValue] = [
            "year": .n(String(self.year)),
            "title": .s(self.title)
        ]

        // Add the `info` field with the rating and/or plot if they're
        // available.

        var details: [Swift.String:DynamoDBClientTypes.AttributeValue] = [:]
        if (self.info.rating != nil || self.info.plot != nil) {
```

```
            if self.info.rating != nil {
                details["rating"] = .n(String(self.info.rating!))
            }
            if self.info.plot != nil {
                details["plot"] = .s(self.info.plot!)
            }
        }
        item["info"] = .m(details)

        return item
    }
}
```

A program that uses the MovieTable class to access a DynamoDB database.

```
import ArgumentParser
import ClientRuntime
import Foundation

import AWSDynamoDB


@testable import MovieList

extension String {
    // Get the directory if the string is a file path.
    func directory() -> String {
        guard let lastIndex = lastIndex(of: "/") else {
            print("Error: String directory separator not found.")
            return ""
        }
        return String(self[...lastIndex])
    }
}

struct ExampleCommand: ParsableCommand {
    @Argument(help: "The path of the sample movie data JSON file.")
    var jsonPath: String = #file.directory() + "../../../../../resources/
sample_files/movies.json"

    @Option(help: "The AWS Region to run AWS API calls in.")
    var awsRegion: String?
```

```
    @Option(
        help: ArgumentHelp("The level of logging for the Swift SDK to perform."),
        completion: .list([
            "critical",
            "debug",
            "error",
            "info",
            "notice",
            "trace",
            "warning"
        ])
    )
    var logLevel: String = "error"

    /// Configuration details for the command.
    static var configuration = CommandConfiguration(
        commandName: "basics",
        abstract: "A basic scenario demonstrating the usage of Amazon DynamoDB.",
        discussion: """
        An example showing how to use Amazon DynamoDB to perform a series of
        common database activities on a simple movie database.
        """
    )

    /// Called by ``main()`` to asynchronously run the AWS example.
    func runAsync() async throws {
        print("Welcome to the AWS SDK for Swift basic scenario for Amazon
DynamoDB!")

        //========================================================================
        // 1. Create the table. The Amazon DynamoDB table is represented by
        //    the `MovieTable` class.
        //========================================================================

        let tableName = "ddb-movies-sample-\(Int.random(in: 1 ... Int.max))"

        print("Creating table \"\(tableName)\"...")

        let movieDatabase = try await MovieTable(region: awsRegion,
                                                 tableName: tableName)

        print("\nWaiting for table to be ready to use...")
        try await movieDatabase.awaitTableActive()
```

```
        //========================================================================
        // 2. Add a movie to the table.
        //========================================================================

        print("\nAdding a movie...")
        try await movieDatabase.add(title: "Avatar: The Way of Water", year: 2022)
        try await movieDatabase.add(title: "Not a Real Movie", year: 2023)

        //========================================================================
        // 3. Update the plot and rating of the movie using an update
        //    expression.
        //========================================================================

        print("\nAdding details to the added movie...")
        _ = try await movieDatabase.update(title: "Avatar: The Way of Water", year:
2022,
                                           rating: 9.2, plot: "It's a sequel.")

        //========================================================================
        // 4. Populate the table from the JSON file.
        //========================================================================

        print("\nPopulating the movie database from JSON...")
        try await movieDatabase.populate(jsonPath: jsonPath)

        //========================================================================
        // 5. Get a specific movie by key. In this example, the key is a
        //    combination of `title` and `year`.
        //========================================================================

        print("\nLooking for a movie in the table...")
        let gotMovie = try await movieDatabase.get(title: "This Is the End", year:
2013)

        print("Found the movie \"\(gotMovie.title)\", released in
\(gotMovie.year).")
        print("Rating: \(gotMovie.info.rating ?? 0.0).")
        print("Plot summary: \(gotMovie.info.plot ?? "None.")")

        //========================================================================
        // 6. Delete a movie.
        //========================================================================
```

```
        print("\nDeleting the added movie...")
        try await movieDatabase.delete(title: "Avatar: The Way of Water", year:
 2022)

        //========================================================================
        // 7. Use a query with a key condition expression to return all movies
        //    released in a given year.
        //========================================================================

        print("\nGetting movies released in 1994...")
        let movieList = try await movieDatabase.getMovies(fromYear: 1994)
        for movie in movieList {
            print("    \(movie.title)")
        }

        //========================================================================
        // 8. Use `scan()` to return movies released in a range of years.
        //========================================================================

        print("\nGetting movies released between 1993 and 1997...")
        let scannedMovies = try await movieDatabase.getMovies(firstYear: 1993,
 lastYear: 1997)
        for movie in scannedMovies {
            print("    \(movie.title) (\(movie.year))")
        }

        //========================================================================
        // 9. Delete the table.
        //========================================================================

        print("\nDeleting the table...")
        try await movieDatabase.deleteTable()
    }
}

@main
struct Main {
    static func main() async {
        let args = Array(CommandLine.arguments.dropFirst())

        do {
            let command = try ExampleCommand.parse(args)
            try await command.runAsync()
        } catch {
```

```
            ExampleCommand.exit(withError: error)
        }
    }
}
```

- For API details, see the following topics in *AWS SDK for Swift API reference*.
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query
  - Scan
  - UpdateItem

Actions

`BatchGetItem`

The following code example shows how to use `BatchGetItem`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Gets an array of `Movie` objects describing all the movies in the
```

```
    /// specified list. Any movies that aren't found in the list have no
    /// corresponding entry in the resulting array.
    ///
    /// - Parameters
    ///     - keys: An array of tuples, each of which specifies the title and
    ///       release year of a movie to fetch from the table.
    ///
    /// - Returns:
    ///     - An array of `Movie` objects describing each match found in the
    ///     table.
    ///
    /// - Throws:
    ///     - `MovieError.ClientUninitialized` if the DynamoDB client has not
    ///     been initialized.
    ///     - DynamoDB errors are thrown without change.
    func batchGet(keys: [(title: String, year: Int)]) async throws -> [Movie] {
        do {
            guard let client = self.ddbClient else {
                throw MovieError.ClientUninitialized
            }

            var movieList: [Movie] = []
            var keyItems: [[Swift.String: DynamoDBClientTypes.AttributeValue]] = []

            // Convert the list of keys into the form used by DynamoDB.

            for key in keys {
                let item: [Swift.String: DynamoDBClientTypes.AttributeValue] = [
                    "title": .s(key.title),
                    "year": .n(String(key.year))
                ]
                keyItems.append(item)
            }

            // Create the input record for `batchGetItem()`. The list of requested
            // items is in the `requestItems` property. This array contains one
            // entry for each table from which items are to be fetched. In this
            // example, there's only one table containing the movie data.
            //
            // If we wanted this program to also support searching for matches
            // in a table of book data, we could add a second `requestItem`
            // mapping the name of the book table to the list of items we want to
            // find in it.
            let input = BatchGetItemInput(
```

```
                requestItems: [
                    self.tableName: .init(
                        consistentRead: true,
                        keys: keyItems
                    )
                ]
            )

            // Fetch the matching movies from the table.

            let output = try await client.batchGetItem(input: input)

            // Get the set of responses. If there aren't any, return the empty
            // movie list.

            guard let responses = output.responses else {
                return movieList
            }

            // Get the list of matching items for the table with the name
            // `tableName`.

            guard let responseList = responses[self.tableName] else {
                return movieList
            }

            // Create `Movie` items for each of the matching movies in the table
            // and add them to the `MovieList` array.

            for response in responseList {
                try movieList.append(Movie(withItem: response))
            }

            return movieList
        } catch {
            print("ERROR: batchGet", dump(error))
            throw error
        }
    }
```

- For API details, see BatchGetItem in *AWS SDK for Swift API reference*.

`BatchWriteItem`

The following code example shows how to use `BatchWriteItem`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Populate the movie database from the specified JSON file.
    ///
    /// - Parameter jsonPath: Path to a JSON file containing movie data.
    ///
    func populate(jsonPath: String) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Create a Swift `URL` and use it to load the file into a `Data`
            // object. Then decode the JSON into an array of `Movie` objects.

            let fileUrl = URL(fileURLWithPath: jsonPath)
            let jsonData = try Data(contentsOf: fileUrl)

            var movieList = try JSONDecoder().decode([Movie].self, from: jsonData)

            // Truncate the list to the first 200 entries or so for this example.

            if movieList.count > 200 {
                movieList = Array(movieList[...199])
            }

            // Before sending records to the database, break the movie list into
            // 25-entry chunks, which is the maximum size of a batch item request.

            let count = movieList.count
```

```
            let chunks = stride(from: 0, to: count, by: 25).map {
                Array(movieList[$0 ..< Swift.min($0 + 25, count)])
            }

            // For each chunk, create a list of write request records and populate
            // them with `PutRequest` requests, each specifying one movie from the
            // chunk. Once the chunk's items are all in the `PutRequest` list,
            // send them to Amazon DynamoDB using the
            // `DynamoDBClient.batchWriteItem()` function.

            for chunk in chunks {
                var requestList: [DynamoDBClientTypes.WriteRequest] = []

                for movie in chunk {
                    let item = try await movie.getAsItem()
                    let request = DynamoDBClientTypes.WriteRequest(
                        putRequest: .init(
                            item: item
                        )
                    )
                    requestList.append(request)
                }

                let input = BatchWriteItemInput(requestItems: [tableName:
requestList])
                _ = try await client.batchWriteItem(input: input)
            }
        } catch {
            print("ERROR: populate:", dump(error))
            throw error
        }
    }
```

- For API details, see [BatchWriteItem](#) in *AWS SDK for Swift API reference*.

`CreateTable`

The following code example shows how to use `CreateTable`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    ///
    /// Create a movie table in the Amazon DynamoDB data store.
    ///
    private func createTable() async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = CreateTableInput(
                attributeDefinitions: [
                    DynamoDBClientTypes.AttributeDefinition(attributeName: "year",
 attributeType: .n),
                    DynamoDBClientTypes.AttributeDefinition(attributeName: "title",
 attributeType: .s)
                ],
                keySchema: [
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "year",
 keyType: .hash),
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "title",
 keyType: .range)
                ],
                provisionedThroughput: DynamoDBClientTypes.ProvisionedThroughput(
                    readCapacityUnits: 10,
                    writeCapacityUnits: 10
                ),
                tableName: self.tableName
            )
            let output = try await client.createTable(input: input)
            if output.tableDescription == nil {
                throw MoviesError.TableNotFound
```

```
            }
        } catch {
            print("ERROR: createTable:", dump(error))
            throw error
        }
    }
```

- For API details, see CreateTable in *AWS SDK for Swift API reference*.

`DeleteItem`

The following code example shows how to use `DeleteItem`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Delete a movie, given its title and release year.
    ///
    /// - Parameters:
    ///   - title: The movie's title.
    ///   - year: The movie's release year.
    ///
    func delete(title: String, year: Int) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DeleteItemInput(
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
```

```
                ],
                tableName: self.tableName
            )
            _ = try await client.deleteItem(input: input)
        } catch {
            print("ERROR: delete:", dump(error))
            throw error
        }
    }
```

- For API details, see DeleteItem in *AWS SDK for Swift API reference*.

`DeleteTable`

The following code example shows how to use `DeleteTable`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    ///
    /// Deletes the table from Amazon DynamoDB.
    ///
    func deleteTable() async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DeleteTableInput(
                tableName: self.tableName
            )
            _ = try await client.deleteTable(input: input)
        } catch {
```

```
            print("ERROR: deleteTable:", dump(error))
            throw error
        }
    }
```

- For API details, see DeleteTable in *AWS SDK for Swift API reference*.

`GetItem`

The following code example shows how to use `GetItem`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Return a `Movie` record describing the specified movie from the Amazon
    /// DynamoDB table.
    ///
    /// - Parameters:
    ///   - title: The movie's title (`String`).
    ///   - year: The movie's release year (`Int`).
    ///
    /// - Throws: `MoviesError.ItemNotFound` if the movie isn't in the table.
    ///
    /// - Returns: A `Movie` record with the movie's details.
    func get(title: String, year: Int) async throws -> Movie {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = GetItemInput(
                key: [
```

```
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                tableName: self.tableName
            )
            let output = try await client.getItem(input: input)
            guard let item = output.item else {
                throw MoviesError.ItemNotFound
            }

            let movie = try Movie(withItem: item)
            return movie
        } catch {
            print("ERROR: get:", dump(error))
            throw error
        }
    }
```

- For API details, see GetItem in *AWS SDK for Swift API reference*.

`ListTables`

The following code example shows how to use `ListTables`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Get a list of the DynamoDB tables available in the specified Region.
    ///
    /// - Returns: An array of strings listing all of the tables available
    ///   in the Region specified when the session was created.
    public func getTableList() async throws -> [String] {
```

```
        let input = ListTablesInput(
        )
        return try await session.listTables(input: input)
    }
```

- For API details, see ListTables in *AWS SDK for Swift API reference*.

`PutItem`

The following code example shows how to use `PutItem`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Add a movie specified as a `Movie` structure to the Amazon DynamoDB
    /// table.
    ///
    /// - Parameter movie: The `Movie` to add to the table.
    ///
    func add(movie: Movie) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Get a DynamoDB item containing the movie data.
            let item = try await movie.getAsItem()

            // Send the `PutItem` request to Amazon DynamoDB.

            let input = PutItemInput(
                item: item,
                tableName: self.tableName
```

```
            )
            _ = try await client.putItem(input: input)
        } catch {
            print("ERROR: add movie:", dump(error))
            throw error
        }
    }


    ///
    /// Return an array mapping attribute names to Amazon DynamoDB attribute
    /// values, representing the contents of the `Movie` record as a DynamoDB
    /// item.
    ///
    /// - Returns: The movie item as an array of type
    ///   `[Swift.String:DynamoDBClientTypes.AttributeValue]`.
    ///
    func getAsItem() async throws ->
[Swift.String:DynamoDBClientTypes.AttributeValue]  {
        // Build the item record, starting with the year and title, which are
        // always present.

        var item: [Swift.String:DynamoDBClientTypes.AttributeValue] = [
            "year": .n(String(self.year)),
            "title": .s(self.title)
        ]

        // Add the `info` field with the rating and/or plot if they're
        // available.

        var details: [Swift.String:DynamoDBClientTypes.AttributeValue] = [:]
        if (self.info.rating != nil || self.info.plot != nil) {
            if self.info.rating != nil {
                details["rating"] = .n(String(self.info.rating!))
            }
            if self.info.plot != nil {
                details["plot"] = .s(self.info.plot!)
            }
        }
        item["info"] = .m(details)

        return item
    }
```

- For API details, see PutItem in *AWS SDK for Swift API reference*.

`Query`

The following code example shows how to use `Query`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Get all the movies released in the specified year.
    ///
    /// - Parameter year: The release year of the movies to return.
    ///
    /// - Returns: An array of `Movie` objects describing each matching movie.
    ///
    func getMovies(fromYear year: Int) async throws -> [Movie] {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = QueryInput(
                expressionAttributeNames: [
                    "#y": "year"
                ],
                expressionAttributeValues: [
                    ":y": .n(String(year))
                ],
                keyConditionExpression: "#y = :y",
                tableName: self.tableName
            )
            // Use "Paginated" to get all the movies.
```

```
            // This lets the SDK handle the 'lastEvaluatedKey' property in
"QueryOutput".

            let pages = client.queryPaginated(input: input)

            var movieList: [Movie] = []
            for try await page in pages {
                guard let items = page.items else {
                    print("Error: no items returned.")
                    continue
                }

                // Convert the found movies into `Movie` objects and return an array
                // of them.

                for item in items {
                    let movie = try Movie(withItem: item)
                    movieList.append(movie)
                }
            }
            return movieList
        } catch {
            print("ERROR: getMovies:", dump(error))
            throw error
        }
    }
```

- For API details, see Query in *AWS SDK for Swift API reference*.

Scan

The following code example shows how to use `Scan`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Return an array of `Movie` objects released in the specified range of
    /// years.
    ///
    /// - Parameters:
    ///   - firstYear: The first year of movies to return.
    ///   - lastYear: The last year of movies to return.
    ///   - startKey: A starting point to resume processing; always use `nil`.
    ///
    /// - Returns: An array of `Movie` objects describing the matching movies.
    ///
    /// > Note: The `startKey` parameter is used by this function when
    ///   recursively calling itself, and should always be `nil` when calling
    ///   directly.
    ///
    func getMovies(firstYear: Int, lastYear: Int,
                   startKey: [Swift.String: DynamoDBClientTypes.AttributeValue]? =
 nil)
            async throws -> [Movie]
    {
        do {
            var movieList: [Movie] = []

            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = ScanInput(
                consistentRead: true,
                exclusiveStartKey: startKey,
                expressionAttributeNames: [
                    "#y": "year" // `year` is a reserved word, so use `#y` instead.
                ],
                expressionAttributeValues: [
                    ":y1": .n(String(firstYear)),
                    ":y2": .n(String(lastYear))
                ],
                filterExpression: "#y BETWEEN :y1 AND :y2",
                tableName: self.tableName
            )
```

```
            let pages = client.scanPaginated(input: input)

            for try await page in pages {
                guard let items = page.items else {
                    print("Error: no items returned.")
                    continue
                }

                // Build an array of `Movie` objects for the returned items.

                for item in items {
                    let movie = try Movie(withItem: item)
                    movieList.append(movie)
                }
            }
            return movieList

        } catch {
            print("ERROR: getMovies with scan:", dump(error))
            throw error
        }
    }
```

- For API details, see Scan in *AWS SDK for Swift API reference*.

`UpdateItem`

The following code example shows how to use `UpdateItem`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB
```

```
    /// Update the specified movie with new `rating` and `plot` information.
    ///
    /// - Parameters:
    ///   - title: The title of the movie to update.
    ///   - year: The release year of the movie to update.
    ///   - rating: The new rating for the movie.
    ///   - plot: The new plot summary string for the movie.
    ///
    /// - Returns: An array of mappings of attribute names to their new
    ///   listing each item actually changed. Items that didn't need to change
    ///   aren't included in this list. `nil` if no changes were made.
    ///
    func update(title: String, year: Int, rating: Double? = nil, plot: String? =
nil) async throws
        -> [Swift.String: DynamoDBClientTypes.AttributeValue]?
    {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Build the update expression and the list of expression attribute
            // values. Include only the information that's changed.

            var expressionParts: [String] = []
            var attrValues: [Swift.String: DynamoDBClientTypes.AttributeValue] = [:]

            if rating != nil {
                expressionParts.append("info.rating=:r")
                attrValues[":r"] = .n(String(rating!))
            }
            if plot != nil {
                expressionParts.append("info.plot=:p")
                attrValues[":p"] = .s(plot!)
            }
            let expression = "set \(expressionParts.joined(separator: ", "))"

            let input = UpdateItemInput(
                // Create substitution tokens for the attribute values, to ensure
                // no conflicts in expression syntax.
                expressionAttributeValues: attrValues,
                // The key identifying the movie to update consists of the release
                // year and title.
                key: [
```

```
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                returnValues: .updatedNew,
                tableName: self.tableName,
                updateExpression: expression
            )
            let output = try await client.updateItem(input: input)

            guard let attributes: [Swift.String: DynamoDBClientTypes.AttributeValue]
= output.attributes else {
                throw MoviesError.InvalidAttributes
            }
            return attributes
        } catch {
            print("ERROR: update:", dump(error))
            throw error
        }
    }
```

- For API details, see UpdateItem in *AWS SDK for Swift API reference*.

IAM examples using SDK for Swift

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for Swift with IAM.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- Actions

Actions

`AttachRolePolicy`

The following code example shows how to use `AttachRolePolicy`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func attachRolePolicy(role: String, policyArn: String) async throws {
        let input = AttachRolePolicyInput(
            policyArn: policyArn,
            roleName: role
        )
        do {
            _ = try await client.attachRolePolicy(input: input)
        } catch {
            print("ERROR: Attaching a role policy:", dump(error))
            throw error
        }
    }
```

- For API details, see AttachRolePolicy in *AWS SDK for Swift API reference*.

`CreateAccessKey`

The following code example shows how to use `CreateAccessKey`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func createAccessKey(userName: String) async throws ->
 IAMClientTypes.AccessKey {
        let input = CreateAccessKeyInput(
            userName: userName
        )
        do {
            let output = try await iamClient.createAccessKey(input: input)
            guard let accessKey = output.accessKey else {
                throw ServiceHandlerError.keyError
            }
            return accessKey
        } catch {
            print("ERROR: createAccessKey:", dump(error))
            throw error
        }
    }
```

- For API details, see CreateAccessKey in *AWS SDK for Swift API reference*.

`CreatePolicy`

The following code example shows how to use `CreatePolicy`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func createPolicy(name: String, policyDocument: String) async throws ->
 IAMClientTypes.Policy {
        let input = CreatePolicyInput(
            policyDocument: policyDocument,
            policyName: name
        )
        do {
            let output = try await iamClient.createPolicy(input: input)
            guard let policy = output.policy else {
                throw ServiceHandlerError.noSuchPolicy
            }
            return policy
        } catch {
            print("ERROR: createPolicy:", dump(error))
            throw error
        }
    }
```

- For API details, see CreatePolicy in *AWS SDK for Swift API reference*.

`CreateRole`

The following code example shows how to use `CreateRole`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func createRole(name: String, policyDocument: String) async throws ->
 String {
        let input = CreateRoleInput(
            assumeRolePolicyDocument: policyDocument,
            roleName: name
        )
        do {
            let output = try await client.createRole(input: input)
            guard let role = output.role else {
                throw ServiceHandlerError.noSuchRole
            }
            guard let id = role.roleId else {
                throw ServiceHandlerError.noSuchRole
            }
            return id
        } catch {
            print("ERROR: createRole:", dump(error))
            throw error
        }
    }
```

- For API details, see CreateRole in *AWS SDK for Swift API reference*.

`CreateServiceLinkedRole`

The following code example shows how to use `CreateServiceLinkedRole`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func createServiceLinkedRole(service: String, suffix: String? = nil,
 description: String?)
                    async throws -> IAMClientTypes.Role {
        let input = CreateServiceLinkedRoleInput(
            awsServiceName: service,
            customSuffix: suffix,
            description: description
        )
        do {
            let output = try await client.createServiceLinkedRole(input: input)
            guard let role = output.role else {
                throw ServiceHandlerError.noSuchRole
            }
            return role
        } catch {
            print("ERROR: createServiceLinkedRole:", dump(error))
            throw error
        }
    }
```

- For API details, see CreateServiceLinkedRole in *AWS SDK for Swift API reference*.

`CreateUser`

The following code example shows how to use `CreateUser`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func createUser(name: String) async throws -> String {
        let input = CreateUserInput(
            userName: name
        )
        do {
            let output = try await client.createUser(input: input)
            guard let user = output.user else {
                throw ServiceHandlerError.noSuchUser
            }
            guard let id = user.userId else {
                throw ServiceHandlerError.noSuchUser
            }
            return id
        } catch {
            print("ERROR: createUser:", dump(error))
            throw error
        }
    }
```

- For API details, see CreateUser in *AWS SDK for Swift API reference*.

`DeleteAccessKey`

The following code example shows how to use `DeleteAccessKey`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func deleteAccessKey(user: IAMClientTypes.User? = nil,
                                key: IAMClientTypes.AccessKey) async throws
    {
        let userName: String?

        if user != nil {
            userName = user!.userName
        } else {
            userName = nil
        }

        let input = DeleteAccessKeyInput(
            accessKeyId: key.accessKeyId,
            userName: userName
        )
        do {
            _ = try await iamClient.deleteAccessKey(input: input)
        } catch {
            print("ERROR: deleteAccessKey:", dump(error))
            throw error
        }
    }
```

- For API details, see DeleteAccessKey in *AWS SDK for Swift API reference*.

`DeletePolicy`

The following code example shows how to use `DeletePolicy`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func deletePolicy(policy: IAMClientTypes.Policy) async throws {
        let input = DeletePolicyInput(
            policyArn: policy.arn
        )
        do {
            _ = try await iamClient.deletePolicy(input: input)
        } catch {
            print("ERROR: deletePolicy:", dump(error))
            throw error
        }
    }
```

- For API details, see DeletePolicy in *AWS SDK for Swift API reference*.

`DeleteRole`

The following code example shows how to use `DeleteRole`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func deleteRole(role: IAMClientTypes.Role) async throws {
        let input = DeleteRoleInput(
            roleName: role.roleName
        )
        do {
            _ = try await iamClient.deleteRole(input: input)
        } catch {
            print("ERROR: deleteRole:", dump(error))
            throw error
        }
    }
```

- For API details, see DeleteRole in *AWS SDK for Swift API reference*.

`DeleteUser`

The following code example shows how to use `DeleteUser`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func deleteUser(user: IAMClientTypes.User) async throws {
        let input = DeleteUserInput(
            userName: user.userName
        )
        do {
            _ = try await iamClient.deleteUser(input: input)
        } catch {
            print("ERROR: deleteUser:", dump(error))
            throw error
        }
    }
```

- For API details, see DeleteUser in *AWS SDK for Swift API reference*.

`DeleteUserPolicy`

The following code example shows how to use `DeleteUserPolicy`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    func deleteUserPolicy(user: IAMClientTypes.User, policyName: String) async
 throws {
        let input = DeleteUserPolicyInput(
            policyName: policyName,
            userName: user.userName
```

```
        )
        do {
            _ = try await iamClient.deleteUserPolicy(input: input)
        } catch {
            print("ERROR: deleteUserPolicy:", dump(error))
            throw error
        }
    }
```

- For API details, see DeleteUserPolicy in *AWS SDK for Swift API reference*.

`DetachRolePolicy`

The following code example shows how to use `DetachRolePolicy`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func detachRolePolicy(policy: IAMClientTypes.Policy, role:
 IAMClientTypes.Role) async throws {
        let input = DetachRolePolicyInput(
            policyArn: policy.arn,
            roleName: role.roleName
        )

        do {
            _ = try await iamClient.detachRolePolicy(input: input)
        } catch {
            print("ERROR: detachRolePolicy:", dump(error))
            throw error
```

```
        }
    }
```

- For API details, see DetachRolePolicy in *AWS SDK for Swift API reference*.

`GetPolicy`

The following code example shows how to use `GetPolicy`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func getPolicy(arn: String) async throws -> IAMClientTypes.Policy {
        let input = GetPolicyInput(
            policyArn: arn
        )
        do {
            let output = try await client.getPolicy(input: input)
            guard let policy = output.policy else {
                throw ServiceHandlerError.noSuchPolicy
            }
            return policy
        } catch {
            print("ERROR: getPolicy:", dump(error))
            throw error
        }
    }
```

- For API details, see GetPolicy in *AWS SDK for Swift API reference*.

`GetRole`

The following code example shows how to use `GetRole`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func getRole(name: String) async throws -> IAMClientTypes.Role {
        let input = GetRoleInput(
            roleName: name
        )
        do {
            let output = try await client.getRole(input: input)
            guard let role = output.role else {
                throw ServiceHandlerError.noSuchRole
            }
            return role
        } catch {
            print("ERROR: getRole:", dump(error))
            throw error
        }
    }
```

- For API details, see GetRole in *AWS SDK for Swift API reference*.

`GetUser`

The following code example shows how to use `GetUser`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func getUser(name: String? = nil) async throws -> IAMClientTypes.User {
        let input = GetUserInput(
            userName: name
        )
        do {
            let output = try await iamClient.getUser(input: input)
            guard let user = output.user else {
                throw ServiceHandlerError.noSuchUser
            }
            return user
        } catch {
            print("ERROR: getUser:", dump(error))
            throw error
        }
    }
```

- For API details, see GetUser in *AWS SDK for Swift API reference*.

`ListAttachedRolePolicies`

The following code example shows how to use `ListAttachedRolePolicies`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3



    /// Returns a list of AWS Identity and Access Management (IAM) policies
    /// that are attached to the role.
    ///
    /// - Parameter role: The IAM role to return the policy list for.
    ///
    /// - Returns: An array of `IAMClientTypes.AttachedPolicy` objects
    ///   describing each managed policy that's attached to the role.
    public func listAttachedRolePolicies(role: String) async throws ->
 [IAMClientTypes.AttachedPolicy] {
        var policyList: [IAMClientTypes.AttachedPolicy] = []

        // Use "Paginated" to get all the attached role polices.
        // This lets the SDK handle the 'isTruncated' in
 "ListAttachedRolePoliciesOutput".
        let input = ListAttachedRolePoliciesInput(
            roleName: role
        )
        let output = client.listAttachedRolePoliciesPaginated(input: input)

        do {
            for try await page in output {
                guard let attachedPolicies = page.attachedPolicies else {
                    print("Error: no attached policies returned.")
                    continue
                }
                for attachedPolicy in attachedPolicies {
                    policyList.append(attachedPolicy)
                }
            }
```

```
        } catch {
            print("ERROR: listAttachedRolePolicies:", dump(error))
            throw error
        }

        return policyList
    }
```

- For API details, see ListAttachedRolePolicies in *AWS SDK for Swift API reference*.

`ListGroups`

The following code example shows how to use `ListGroups`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func listGroups() async throws -> [String] {
        var groupList: [String] = []

        // Use "Paginated" to get all the groups.
        // This lets the SDK handle the 'isTruncated' property in
 "ListGroupsOutput".
        let input = ListGroupsInput()

        let pages = client.listGroupsPaginated(input: input)
        do {
            for try await page in pages {
                guard let groups = page.groups else {
                    print("Error: no groups returned.")
                    continue
                }
```

```
                for group in groups {
                    if let name = group.groupName {
                        groupList.append(name)
                    }
                }
            }
        } catch {
            print("ERROR: listGroups:", dump(error))
            throw error
        }
        return groupList
    }
```

- For API details, see ListGroups in *AWS SDK for Swift API reference*.

`ListPolicies`

The following code example shows how to use `ListPolicies`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func listPolicies() async throws -> [MyPolicyRecord] {
        var policyList: [MyPolicyRecord] = []

        // Use "Paginated" to get all the policies.
        // This lets the SDK handle the 'isTruncated' in "ListPoliciesOutput".
        let input = ListPoliciesInput()
        let output = client.listPoliciesPaginated(input: input)

        do {
```

```
            for try await page in output {
                guard let policies = page.policies else {
                    print("Error: no policies returned.")
                    continue
                }

                for policy in policies {
                    guard let name = policy.policyName,
                          let id = policy.policyId,
                          let arn = policy.arn
                    else {
                        throw ServiceHandlerError.noSuchPolicy
                    }
                    policyList.append(MyPolicyRecord(name: name, id: id, arn: arn))
                }
            }
        } catch {
            print("ERROR: listPolicies:", dump(error))
            throw error
        }

        return policyList
    }
```

- For API details, see ListPolicies in *AWS SDK for Swift API reference*.

`ListRolePolicies`

The following code example shows how to use `ListRolePolicies`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3
```

```
public func listRolePolicies(role: String) async throws -> [String] {
    var policyList: [String] = []

    // Use "Paginated" to get all the role policies.
    // This lets the SDK handle the 'isTruncated' in "ListRolePoliciesOutput".
    let input = ListRolePoliciesInput(
        roleName: role
    )
    let pages = client.listRolePoliciesPaginated(input: input)

    do {
        for try await page in pages {
            guard let policies = page.policyNames else {
                print("Error: no role policies returned.")
                continue
            }

            for policy in policies {
                policyList.append(policy)
            }
        }
    } catch {
        print("ERROR: listRolePolicies:", dump(error))
        throw error
    }
    return policyList
}
```

- For API details, see ListRolePolicies in *AWS SDK for Swift API reference*.

`ListRoles`

The following code example shows how to use `ListRoles`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func listRoles() async throws -> [String] {
        var roleList: [String] = []

        // Use "Paginated" to get all the roles.
        // This lets the SDK handle the 'isTruncated' in "ListRolesOutput".
        let input = ListRolesInput()
        let pages = client.listRolesPaginated(input: input)

        do {
            for try await page in pages {
                guard let roles = page.roles else {
                    print("Error: no roles returned.")
                    continue
                }

                for role in roles {
                    if let name = role.roleName {
                        roleList.append(name)
                    }
                }
            }
        } catch {
            print("ERROR: listRoles:", dump(error))
            throw error
        }
        return roleList
    }
```

- For API details, see ListRoles in *AWS SDK for Swift API reference*.

`ListUsers`

The following code example shows how to use `ListUsers`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    public func listUsers() async throws -> [MyUserRecord] {
        var userList: [MyUserRecord] = []

        // Use "Paginated" to get all the users.
        // This lets the SDK handle the 'isTruncated' in "ListUsersOutput".
        let input = ListUsersInput()
        let output = client.listUsersPaginated(input: input)

        do {
            for try await page in output {
                guard let users = page.users else {
                    continue
                }
                for user in users {
                    if let id = user.userId, let name = user.userName {
                        userList.append(MyUserRecord(id: id, name: name))
                    }
                }
            }
        }
        catch {
            print("ERROR: listUsers:", dump(error))
            throw error
        }
       return userList
    }
```

- For API details, see ListUsers in *AWS SDK for Swift API reference*.

`PutUserPolicy`

The following code example shows how to use `PutUserPolicy`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSIAM
import AWSS3


    func putUserPolicy(policyDocument: String, policyName: String, user:
 IAMClientTypes.User) async throws {
        let input = PutUserPolicyInput(
            policyDocument: policyDocument,
            policyName: policyName,
            userName: user.userName
        )
        do {
            _ = try await iamClient.putUserPolicy(input: input)
        } catch {
            print("ERROR: putUserPolicy:", dump(error))
            throw error
        }
    }
```

- For API details, see PutUserPolicy in *AWS SDK for Swift API reference*.

Amazon S3 examples using SDK for Swift

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for Swift with Amazon S3.

Basics are code examples that show you how to perform the essential operations within a service.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

Basics

Learn the basics

The following code example shows how to:

- Create a bucket and upload a file to it.
- Download an object from a bucket.
- Copy an object to a subfolder in a bucket.
- List the objects in a bucket.
- Delete the bucket objects and the bucket.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSS3

import Foundation
```

```
import AWSS3
import Smithy
import ClientRuntime

/// A class containing all the code that interacts with the AWS SDK for Swift.
public class ServiceHandler {
    let configuration: S3Client.S3ClientConfiguration
    let client: S3Client

    enum HandlerError: Error {
        case getObjectBody(String)
        case readGetObjectBody(String)
        case missingContents(String)
    }

    /// Initialize and return a new ``ServiceHandler`` object, which is used to
 drive the AWS calls
    /// used for the example.
    ///
    /// - Returns: A new ``ServiceHandler`` object, ready to be called to
    ///            execute AWS operations.
    public init() async throws {
        do {
            configuration = try await S3Client.S3ClientConfiguration()
         //   configuration.region = "us-east-2" // Uncomment this to set the region
 programmatically.
            client = S3Client(config: configuration)
        }
        catch {
            print("ERROR: ", dump(error, name: "Initializing S3 client"))
            throw error
        }
    }


    /// Create a new user given the specified name.
    ///
    /// - Parameters:
    ///   - name: Name of the bucket to create.
    /// Throws an exception if an error occurs.
    public func createBucket(name: String) async throws {
        var input = CreateBucketInput(
            bucket: name
        )
```

```
        // For regions other than "us-east-1", you must set the locationConstraint
 in the createBucketConfiguration.
        // For more information, see LocationConstraint in the S3 API guide.
        // https://docs.aws.amazon.com/AmazonS3/latest/API/
API_CreateBucket.html#API_CreateBucket_RequestBody
        if let region = configuration.region {
            if region != "us-east-1" {
                input.createBucketConfiguration =
 S3ClientTypes.CreateBucketConfiguration(locationConstraint:
 S3ClientTypes.BucketLocationConstraint(rawValue: region))
            }
        }

        do {
            _ = try await client.createBucket(input: input)
        }
        catch let error as BucketAlreadyOwnedByYou {
            print("The bucket '\(name)' already exists and is owned by you. You may
 wish to ignore this exception.")
            throw error
        }
        catch {
            print("ERROR: ", dump(error, name: "Creating a bucket"))
            throw error
        }
    }

    /// Delete a bucket.
    /// - Parameter name: Name of the bucket to delete.
    public func deleteBucket(name: String) async throws {
        let input = DeleteBucketInput(
            bucket: name
        )
        do {
            _ = try await client.deleteBucket(input: input)
        }
        catch {
            print("ERROR: ", dump(error, name: "Deleting a bucket"))
            throw error
        }
    }

    /// Upload a file from local storage to the bucket.
```

```
    /// - Parameters:
    ///   - bucket: Name of the bucket to upload the file to.
    ///   - key: Name of the file to create.
    ///   - file: Path name of the file to upload.
    public func uploadFile(bucket: String, key: String, file: String) async throws {
        let fileUrl = URL(fileURLWithPath: file)
        do {
            let fileData = try Data(contentsOf: fileUrl)
            let dataStream = ByteStream.data(fileData)

            let input = PutObjectInput(
                body: dataStream,
                bucket: bucket,
                key: key
            )

            _ = try await client.putObject(input: input)
        }
        catch {
            print("ERROR: ", dump(error, name: "Putting an object."))
            throw error
        }
    }

    /// Create a file in the specified bucket with the given name. The new
    /// file's contents are uploaded from a `Data` object.
    ///
    /// - Parameters:
    ///   - bucket: Name of the bucket to create a file in.
    ///   - key: Name of the file to create.
    ///   - data: A `Data` object to write into the new file.
    public func createFile(bucket: String, key: String, withData data: Data) async
throws {
        let dataStream = ByteStream.data(data)

        let input = PutObjectInput(
            body: dataStream,
            bucket: bucket,
            key: key
        )

        do {
            _ = try await client.putObject(input: input)
        }
```

```
        catch {
            print("ERROR: ", dump(error, name: "Putting an object."))
            throw error
        }
    }

    /// Download the named file to the given directory on the local device.
    ///
    /// - Parameters:
    ///   - bucket: Name of the bucket that contains the file to be copied.
    ///   - key: The name of the file to copy from the bucket.
    ///   - to: The path of the directory on the local device where you want to
    ///     download the file.
    public func downloadFile(bucket: String, key: String, to: String) async throws {
        let fileUrl = URL(fileURLWithPath: to).appendingPathComponent(key)

        let input = GetObjectInput(
            bucket: bucket,
            key: key
        )
        do {
            let output = try await client.getObject(input: input)

            guard let body = output.body else {
                throw HandlerError.getObjectBody("GetObjectInput missing body.")
            }

            guard let data = try await body.readData() else {
                throw HandlerError.readGetObjectBody("GetObjectInput unable to read
data.")
            }

            try data.write(to: fileUrl)
        }
        catch {
            print("ERROR: ", dump(error, name: "Downloading a file."))
            throw error
        }
    }

    /// Read the specified file from the given S3 bucket into a Swift
    /// `Data` object.
    ///
    /// - Parameters:
```

```
    ///   - bucket: Name of the bucket containing the file to read.
    ///   - key: Name of the file within the bucket to read.
    ///
    /// - Returns: A `Data` object containing the complete file data.
    public func readFile(bucket: String, key: String) async throws -> Data {
        let input = GetObjectInput(
            bucket: bucket,
            key: key
        )
        do {
            let output = try await client.getObject(input: input)

            guard let body = output.body else {
                throw HandlerError.getObjectBody("GetObjectInput missing body.")
            }

            guard let data = try await body.readData() else {
                throw HandlerError.readGetObjectBody("GetObjectInput unable to read
 data.")
            }

            return data
        }
        catch {
            print("ERROR: ", dump(error, name: "Reading a file."))
            throw error
        }
    }


    /// Copy a file from one bucket to another.
    ///
    /// - Parameters:
    ///   - sourceBucket: Name of the bucket containing the source file.
    ///   - name: Name of the source file.
    ///   - destBucket: Name of the bucket to copy the file into.
    public func copyFile(from sourceBucket: String, name: String, to destBucket:
 String) async throws {
        let srcUrl = ("\(sourceBucket)/
\(name)").addingPercentEncoding(withAllowedCharacters: .urlPathAllowed)

        let input = CopyObjectInput(
            bucket: destBucket,
            copySource: srcUrl,
```

```
            key: name
        )
        do {
            _ = try await client.copyObject(input: input)
        }
        catch {
            print("ERROR: ", dump(error, name: "Copying an object."))
            throw error
        }
    }

    /// Deletes the specified file from Amazon S3.
    ///
    /// - Parameters:
    ///   - bucket: Name of the bucket containing the file to delete.
    ///   - key: Name of the file to delete.
    ///
    public func deleteFile(bucket: String, key: String) async throws {
        let input = DeleteObjectInput(
            bucket: bucket,
            key: key
        )

        do {
            _ = try await client.deleteObject(input: input)
        }
        catch {
            print("ERROR: ", dump(error, name: "Deleting a file."))
            throw error
        }
    }

    /// Returns an array of strings, each naming one file in the
    /// specified bucket.
    ///
    /// - Parameter bucket: Name of the bucket to get a file listing for.
    /// - Returns: An array of `String` objects, each giving the name of
    ///            one file contained in the bucket.
    public func listBucketFiles(bucket: String) async throws -> [String] {
        do {
            let input = ListObjectsV2Input(
                bucket: bucket
            )
```

```
            // Use "Paginated" to get all the objects.
            // This lets the SDK handle the 'continuationToken' in
 "ListObjectsV2Output".
            let output = client.listObjectsV2Paginated(input: input)
            var names: [String] = []

            for try await page in output {
                guard let objList = page.contents else {
                    print("ERROR: listObjectsV2Paginated returned nil contents.")
                    continue
                }

                for obj in objList {
                    if let objName = obj.key {
                        names.append(objName)
                    }
                }
            }


            return names
        }
        catch {
            print("ERROR: ", dump(error, name: "Listing objects."))
            throw error
        }
    }
}
```

```
import AWSS3

import Foundation
import ServiceHandler
import ArgumentParser

/// The command-line arguments and options available for this
/// example command.
struct ExampleCommand: ParsableCommand {
    @Argument(help: "Name of the S3 bucket to create")
    var bucketName: String

    @Argument(help: "Pathname of the file to upload to the S3 bucket")
```

```
    var uploadSource: String

    @Argument(help: "The name (key) to give the file in the S3 bucket")
    var objName: String

    @Argument(help: "S3 bucket to copy the object to")
    var destBucket: String

    @Argument(help: "Directory where you want to download the file from the S3
bucket")
    var downloadDir: String

    static var configuration = CommandConfiguration(
        commandName: "s3-basics",
        abstract: "Demonstrates a series of basic AWS S3 functions.",
        discussion: """
        Performs the following Amazon S3 commands:

        * `CreateBucket`
        * `PutObject`
        * `GetObject`
        * `CopyObject`
        * `ListObjects`
        * `DeleteObjects`
        * `DeleteBucket`
        """
    )

    /// Called by ``main()`` to do the actual running of the AWS
    /// example.
    func runAsync() async throws {
        let serviceHandler = try await ServiceHandler()

        // 1. Create the bucket.
        print("Creating the bucket \(bucketName)...")
        try await serviceHandler.createBucket(name: bucketName)

        // 2. Upload a file to the bucket.
        print("Uploading the file \(uploadSource)...")
        try await serviceHandler.uploadFile(bucket: bucketName, key: objName, file:
uploadSource)

        // 3. Download the file.
        print("Downloading the file \(objName) to \(downloadDir)...")
```

```
        try await serviceHandler.downloadFile(bucket: bucketName, key: objName, to:
 downloadDir)

        // 4. Copy the file to another bucket.
        print("Copying the file to the bucket \(destBucket)...")
        try await serviceHandler.copyFile(from: bucketName, name: objName, to:
 destBucket)

        // 5. List the contents of the bucket.

        print("Getting a list of the files in the bucket \(bucketName)")
        let fileList = try await serviceHandler.listBucketFiles(bucket: bucketName)
        let numFiles = fileList.count
        if numFiles != 0 {
            print("\(numFiles) file\((numFiles > 1) ? "s" : "") in bucket
 \(bucketName):")
            for name in fileList {
                print("  \(name)")
            }
        } else {
            print("No files found in bucket \(bucketName)")
        }

        // 6. Delete the objects from the bucket.

        print("Deleting the file \(objName) from the bucket \(bucketName)...")
        try await serviceHandler.deleteFile(bucket: bucketName, key: objName)
        print("Deleting the file \(objName) from the bucket \(destBucket)...")
        try await serviceHandler.deleteFile(bucket: destBucket, key: objName)

        // 7. Delete the bucket.
        print("Deleting the bucket \(bucketName)...")
        try await serviceHandler.deleteBucket(name: bucketName)

        print("Done.")
    }
}

//
// Main program entry point.
//
@main
struct Main {
    static func main() async {
```

```
        let args = Array(CommandLine.arguments.dropFirst())

        do {
            let command = try ExampleCommand.parse(args)
            try await command.runAsync()
        } catch {
            ExampleCommand.exit(withError: error)
        }
    }
}
```

- For API details, see the following topics in *AWS SDK for Swift API reference*.
  - CopyObject
  - CreateBucket
  - DeleteBucket
  - DeleteObjects
  - GetObject
  - ListObjectsV2
  - PutObject

Actions

`CopyObject`

The following code example shows how to use `CopyObject`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSS3
```

```
    public func copyFile(from sourceBucket: String, name: String, to destBucket:
 String) async throws {
        let srcUrl = ("\(sourceBucket)/
\(name)").addingPercentEncoding(withAllowedCharacters: .urlPathAllowed)

        let input = CopyObjectInput(
            bucket: destBucket,
            copySource: srcUrl,
            key: name
        )
        do {
            _ = try await client.copyObject(input: input)
        }
        catch {
            print("ERROR: ", dump(error, name: "Copying an object."))
            throw error
        }
    }
```

- For API details, see CopyObject in *AWS SDK for Swift API reference*.

`CreateBucket`

The following code example shows how to use `CreateBucket`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSS3

    public func createBucket(name: String) async throws {
        var input = CreateBucketInput(
            bucket: name
        )
```

```
        // For regions other than "us-east-1", you must set the locationConstraint
 in the createBucketConfiguration.
        // For more information, see LocationConstraint in the S3 API guide.
        // https://docs.aws.amazon.com/AmazonS3/latest/API/
API_CreateBucket.html#API_CreateBucket_RequestBody
        if let region = configuration.region {
            if region != "us-east-1" {
                input.createBucketConfiguration =
 S3ClientTypes.CreateBucketConfiguration(locationConstraint:
 S3ClientTypes.BucketLocationConstraint(rawValue: region))
            }
        }

        do {
            _ = try await client.createBucket(input: input)
        }
        catch let error as BucketAlreadyOwnedByYou {
            print("The bucket '\(name)' already exists and is owned by you. You may
 wish to ignore this exception.")
            throw error
        }
        catch {
            print("ERROR: ", dump(error, name: "Creating a bucket"))
            throw error
        }
    }
```

- For API details, see CreateBucket in *AWS SDK for Swift API reference*.

`DeleteBucket`

The following code example shows how to use `DeleteBucket`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSS3

    public func deleteBucket(name: String) async throws {
        let input = DeleteBucketInput(
            bucket: name
        )
        do {
            _ = try await client.deleteBucket(input: input)
        }
        catch {
            print("ERROR: ", dump(error, name: "Deleting a bucket"))
            throw error
        }
    }
```

- For API details, see DeleteBucket in *AWS SDK for Swift API reference*.

`DeleteObject`

The following code example shows how to use `DeleteObject`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSS3

    public func deleteFile(bucket: String, key: String) async throws {
        let input = DeleteObjectInput(
            bucket: bucket,
            key: key
        )

        do {
            _ = try await client.deleteObject(input: input)
        }
```

```
        catch {
            print("ERROR: ", dump(error, name: "Deleting a file."))
            throw error
        }
    }
```

- For API details, see DeleteObject in *AWS SDK for Swift API reference*.

`DeleteObjects`

The following code example shows how to use `DeleteObjects`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSS3

    public func deleteObjects(bucket: String, keys: [String]) async throws {
        let input = DeleteObjectsInput(
            bucket: bucket,
            delete: S3ClientTypes.Delete(
                objects: keys.map { S3ClientTypes.ObjectIdentifier(key: $0) },
                quiet: true
            )
        )

        do {
            _ = try await client.deleteObjects(input: input)
        } catch {
            print("ERROR: deleteObjects:", dump(error))
            throw error
        }
    }
```

- For API details, see DeleteObjects in *AWS SDK for Swift API reference*.

`GetObject`

The following code example shows how to use `GetObject`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSS3

    public func downloadFile(bucket: String, key: String, to: String) async throws {
        let fileUrl = URL(fileURLWithPath: to).appendingPathComponent(key)

        let input = GetObjectInput(
            bucket: bucket,
            key: key
        )
        do {
            let output = try await client.getObject(input: input)

            guard let body = output.body else {
                throw HandlerError.getObjectBody("GetObjectInput missing body.")
            }

            guard let data = try await body.readData() else {
                throw HandlerError.readGetObjectBody("GetObjectInput unable to read
 data.")
            }

            try data.write(to: fileUrl)
        }
        catch {
            print("ERROR: ", dump(error, name: "Downloading a file."))
            throw error
        }
    }
```

```
import AWSS3

    public func readFile(bucket: String, key: String) async throws -> Data {
        let input = GetObjectInput(
            bucket: bucket,
            key: key
        )
        do {
            let output = try await client.getObject(input: input)

            guard let body = output.body else {
                throw HandlerError.getObjectBody("GetObjectInput missing body.")
            }

            guard let data = try await body.readData() else {
                throw HandlerError.readGetObjectBody("GetObjectInput unable to read
 data.")
            }

            return data
        }
        catch {
            print("ERROR: ", dump(error, name: "Reading a file."))
            throw error
        }
  }
```

- For API details, see GetObject in *AWS SDK for Swift API reference*.

`ListBuckets`

The following code example shows how to use `ListBuckets`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSS3

    /// Return an array containing information about every available bucket.
    ///
    /// - Returns: An array of ``S3ClientTypes.Bucket`` objects describing
    ///   each bucket.
    public func getAllBuckets() async throws -> [S3ClientTypes.Bucket] {
        return try await client.listBuckets(input: ListBucketsInput())
    }
```

- For API details, see ListBuckets in *AWS SDK for Swift API reference*.

`ListObjectsV2`

The following code example shows how to use `ListObjectsV2`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSS3

    public func listBucketFiles(bucket: String) async throws -> [String] {
        do {
            let input = ListObjectsV2Input(
                bucket: bucket
            )
```

```
            // Use "Paginated" to get all the objects.
            // This lets the SDK handle the 'continuationToken' in
"ListObjectsV2Output".
            let output = client.listObjectsV2Paginated(input: input)
            var names: [String] = []

            for try await page in output {
                guard let objList = page.contents else {
                    print("ERROR: listObjectsV2Paginated returned nil contents.")
                    continue
                }

                for obj in objList {
                    if let objName = obj.key {
                        names.append(objName)
                    }
                }
            }


            return names
        }
        catch {
            print("ERROR: ", dump(error, name: "Listing objects."))
            throw error
        }
    }
```

- For API details, see ListObjectsV2 in *AWS SDK for Swift API reference*.

`PutObject`

The following code example shows how to use `PutObject`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSS3
import Smithy

    public func uploadFile(bucket: String, key: String, file: String) async throws {
        let fileUrl = URL(fileURLWithPath: file)
        do {
            let fileData = try Data(contentsOf: fileUrl)
            let dataStream = ByteStream.data(fileData)

            let input = PutObjectInput(
                body: dataStream,
                bucket: bucket,
                key: key
            )

            _ = try await client.putObject(input: input)
        }
        catch {
            print("ERROR: ", dump(error, name: "Putting an object."))
            throw error
        }
    }
```

```
import AWSS3
import Smithy

    public func createFile(bucket: String, key: String, withData data: Data) async
 throws {
        let dataStream = ByteStream.data(data)

        let input = PutObjectInput(
            body: dataStream,
            bucket: bucket,
            key: key
        )

        do {
            _ = try await client.putObject(input: input)
        }
        catch {
            print("ERROR: ", dump(error, name: "Putting an object."))
            throw error
```

```
        }
    }
```

- For API details, see PutObject in *AWS SDK for Swift API reference*.

Scenarios

Download stream of unknown size

The following code example shows how to download a stream of unknown size from an Amazon S3 object.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import ArgumentParser
import AWSClientRuntime
import AWSS3
import Foundation
import Smithy
import SmithyHTTPAPI
import SmithyStreams


    /// Download a file from the specified bucket.
    ///
    /// - Parameters:
    ///   - bucket: The Amazon S3 bucket name to get the file from.
    ///   - key: The name (or path) of the file to download from the bucket.
    ///   - destPath: The pathname on the local filesystem at which to store
    ///     the downloaded file.
    func downloadFile(bucket: String, key: String, destPath: String?) async throws {
        let fileURL: URL

        // If no destination path was provided, use the key as the name to use
```

```
        // for the file in the downloads folder.

        if destPath == nil {
            do {
                try fileURL = FileManager.default.url(
                    for: .downloadsDirectory,
                    in: .userDomainMask,
                    appropriateFor: URL(string: key),
                    create: true
                ).appendingPathComponent(key)
            } catch {
                throw TransferError.directoryError
            }
        } else {
            fileURL = URL(fileURLWithPath: destPath!)
        }

        let config = try await S3Client.S3ClientConfiguration(region: region)
        let s3Client = S3Client(config: config)

        // Create a `FileHandle` referencing the local destination. Then
        // create a `ByteStream` from that.

        FileManager.default.createFile(atPath: fileURL.path, contents: nil,
attributes: nil)
        let fileHandle = try FileHandle(forWritingTo: fileURL)

        // Download the file using `GetObject`.

        let getInput = GetObjectInput(
            bucket: bucket,
            key: key
        )

        do {
            let getOutput = try await s3Client.getObject(input: getInput)

            guard let body = getOutput.body else {
                throw TransferError.downloadError("Error: No data returned for
download")
            }

            // If the body is returned as a `Data` object, write that to the
            // file. If it's a stream, read the stream chunk by chunk,
```

```
            // appending each chunk to the destination file.

            switch body {
            case .data:
                guard let data = try await body.readData() else {
                    throw TransferError.downloadError("Download error")
                }

                // Write the `Data` to the file.

                do {
                    try data.write(to: fileURL)
                } catch {
                    throw TransferError.writeError
                }
                break

            case .stream(let stream as ReadableStream):
                while (true) {
                    let chunk = try await stream.readAsync(upToCount: 5 * 1024 *
1024)
                    guard let chunk = chunk else {
                        break
                    }

                    // Write the chunk to the destination file.

                    do {
                        try fileHandle.write(contentsOf: chunk)
                    } catch {
                        throw TransferError.writeError
                    }
                }

                break
            default:
                throw TransferError.downloadError("Received data is unknown object
type")
            }
        } catch {
            throw TransferError.downloadError("Error downloading the file:
\(error)")
        }
```

```
        print("File downloaded to \(fileURL.path).")
    }
```

Upload stream of unknown size

The following code example shows how to upload a stream of unknown size to an Amazon S3 object.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import ArgumentParser
import AWSClientRuntime
import AWSS3
import Foundation
import Smithy
import SmithyHTTPAPI
import SmithyStreams


    /// Upload a file to the specified bucket.
    ///
    /// - Parameters:
    ///   - bucket: The Amazon S3 bucket name to store the file into.
    ///   - key: The name (or path) of the file to upload to in the `bucket`.
    ///   - sourcePath: The pathname on the local filesystem of the file to
    ///     upload.
    func uploadFile(sourcePath: String, bucket: String, key: String?) async throws {
        let fileURL: URL = URL(fileURLWithPath: sourcePath)
        let fileName: String

        // If no key was provided, use the last component of the filename.

        if key == nil {
            fileName = fileURL.lastPathComponent
```

```
        } else {
            fileName = key!
        }

        let s3Client = try await S3Client()

        // Create a FileHandle for the source file.

        let fileHandle = FileHandle(forReadingAtPath: sourcePath)
        guard let fileHandle = fileHandle else {
            throw TransferError.readError
        }

        // Create a byte stream to retrieve the file's contents. This uses the
        // Smithy FileStream and ByteStream types.

        let stream = FileStream(fileHandle: fileHandle)
        let body = ByteStream.stream(stream)

        // Create a `PutObjectInput` with the ByteStream as the body of the
        // request's data. The AWS SDK for Swift will handle sending the
        // entire file in chunks, regardless of its size.

        let putInput = PutObjectInput(
            body: body,
            bucket: bucket,
            key: fileName
        )

        do {
            _ = try await s3Client.putObject(input: putInput)
        } catch {
            throw TransferError.uploadError("Error uploading the file: \(error)")
        }

        print("File uploaded to \(fileURL.path).")
    }
```

AWS STS examples using SDK for Swift

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for Swift with AWS STS.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

Actions

`AssumeRole`

The following code example shows how to use `AssumeRole`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSSTS

    public func assumeRole(role: IAMClientTypes.Role, sessionName: String)
        async throws -> STSClientTypes.Credentials
    {
        let input = AssumeRoleInput(
            roleArn: role.arn,
            roleSessionName: sessionName
        )
        do {
            let output = try await stsClient.assumeRole(input: input)
```

```
            guard let credentials = output.credentials else {
                throw ServiceHandlerError.authError
            }

            return credentials
        } catch {
            print("Error assuming role: ", dump(error))
            throw error
        }
    }
```

- For API details, see AssumeRole in *AWS SDK for Swift API reference*.

Amazon Transcribe Streaming examples using SDK for Swift

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for Swift with Amazon Transcribe Streaming.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- Actions
- Scenarios

Actions

`StartStreamTranscription`

The following code example shows how to use `StartStreamTranscription`.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        let client = TranscribeStreamingClient(
            config: try await
TranscribeStreamingClient.TranscribeStreamingClientConfiguration(
                                region: region
            )
        )

        // Start the transcription running on the audio stream.

        let output = try await client.startStreamTranscription(
            input: StartStreamTranscriptionInput(
                audioStream: try await createAudioStream(),
                languageCode: TranscribeStreamingClientTypes.LanguageCode(rawValue:
lang),
                mediaEncoding: mediaEncoding,
                mediaSampleRateHertz: sampleRate
            )
        )
```

- For API details, see StartStreamTranscription in *AWS SDK for Swift API reference*.

Scenarios

Transcribe an audio file

The following code example shows how to generate a transcription of a source audio file using Amazon Transcribe streaming.

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Use Amazon Transcribe streaming to transcribe the spoken language in an audio file.

```
/// An example that demonstrates how to watch an transcribe event stream to
/// transcribe audio from a file to the console.

import ArgumentParser
import AWSClientRuntime
import AWSTranscribeStreaming
import Foundation

/// Identify one of the media file formats supported by Amazon Transcribe.
enum TranscribeFormat: String, ExpressibleByArgument {
    case ogg = "ogg"
    case pcm = "pcm"
    case flac = "flac"
}

// -MARK: - Async command line tool

struct ExampleCommand: ParsableCommand {
    // -MARK: Command arguments
    @Flag(help: "Show partial results")
    var showPartial = false
    @Option(help: "Language code to transcribe into")
    var lang: String = "en-US"
    @Option(help: "Format of the source audio file")
    var format: TranscribeFormat
    @Option(help: "Sample rate of the source audio file in Hertz")
    var sampleRate: Int = 16000
    @Option(help: "Path of the source audio file")
    var path: String
    @Option(help: "Name of the Amazon S3 Region to use (default: us-east-1)")
    var region = "us-east-1"

    static var configuration = CommandConfiguration(
```

```
        commandName: "tsevents",
        abstract: """
        This example shows how to use event streaming with Amazon Transcribe.
        """,
        discussion: """
        """
    )

    /// Create and return an Amazon Transcribe audio stream from the file
    /// specified in the arguments.
    ///
    /// - Throws: Errors from `TranscribeError`.
    ///
    /// - Returns: `AsyncThrowingStream<TranscribeStreamingClientTypes.AudioStream,
Error>`
    func createAudioStream() async throws
                -> AsyncThrowingStream<TranscribeStreamingClientTypes.AudioStream,
Error> {

        let fileURL: URL = URL(fileURLWithPath: path)
        let audioData = try Data(contentsOf: fileURL)

        // Properties defining the size of audio chunks and the total size of
        // the audio file in bytes. You should try to send chunks that last on
        // average 125 milliseconds.

        let chunkSizeInMilliseconds = 125.0
        let chunkSize = Int(chunkSizeInMilliseconds  / 1000.0 * Double(sampleRate) *
2.0)
        let audioDataSize = audioData.count

        // Create an audio stream from the source data. The stream's job is
        // to send the audio in chunks to Amazon Transcribe as
        // `AudioStream.audioevent` events.

        let audioStream =
AsyncThrowingStream<TranscribeStreamingClientTypes.AudioStream,
                                Error> { continuation in
            Task {
                var currentStart = 0
                var currentEnd = min(chunkSize, audioDataSize - currentStart)

                // Generate and send chunks of audio data as `audioevent`
                // events until the entire file has been sent. Each event is
```

```
                // yielded to the SDK after being created.

                while currentStart < audioDataSize {
                    let dataChunk = audioData[currentStart ..< currentEnd]

                    let audioEvent =
TranscribeStreamingClientTypes.AudioStream.audioevent(
                        .init(audioChunk: dataChunk)
                    )
                    let yieldResult = continuation.yield(audioEvent)
                    switch yieldResult {
                        case .enqueued(_):
                            // The chunk was successfully enqueued into the
                            // stream. The `remaining` parameter estimates how
                            // much room is left in the queue, but is ignored here.
                            break
                        case .dropped(_):
                            // The chunk was dropped because the queue buffer
                            // is full. This will cause transcription errors.
                            print("Warning: Dropped audio! The transcription will be
incomplete.")
                        case .terminated:
                            print("Audio stream terminated.")
                            continuation.finish()
                            return
                        default:
                            print("Warning: Unrecognized response during audio
streaming.")
                    }

                    currentStart = currentEnd
                    currentEnd = min(currentStart + chunkSize, audioDataSize)
                }

                // Let the SDK's continuation block know the stream is over.

                continuation.finish()
            }
        }

        return audioStream
    }

    /// Run the transcription process.
```

```
    ///
    /// - Throws: An error from `TranscribeError`.
    func transcribe() async throws {
        // Convert the value of the `--format` option into the Transcribe
        // Streaming `MediaEncoding` type.

        let mediaEncoding: TranscribeStreamingClientTypes.MediaEncoding
        switch format {
        case .flac:
            mediaEncoding = .flac
        case .ogg:
            mediaEncoding = .oggOpus
        case .pcm:
            mediaEncoding = .pcm
        }

        // Create the Transcribe Streaming client.

        let client = TranscribeStreamingClient(
            config: try await
TranscribeStreamingClient.TranscribeStreamingClientConfiguration(
                                region: region
            )
        )

        // Start the transcription running on the audio stream.

        let output = try await client.startStreamTranscription(
            input: StartStreamTranscriptionInput(
                audioStream: try await createAudioStream(),
                languageCode: TranscribeStreamingClientTypes.LanguageCode(rawValue:
lang),
                mediaEncoding: mediaEncoding,
                mediaSampleRateHertz: sampleRate
            )
        )

        // Iterate over the events in the returned transcript result stream.
        // Each `transcriptevent` contains a list of result fragments which
        // need to be concatenated together to build the final transcript.
        for try await event in output.transcriptResultStream! {
            switch event {
            case .transcriptevent(let event):
            for result in event.transcript?.results ?? [] {
```

```
                guard let transcript = result.alternatives?.first?.transcript else {
                    continue
                }

                // If showing partial results is enabled and the result is
                // partial, show it. Partial results may be incomplete, and
                // may be inaccurate, with upcoming audio making the
                // transcription complete or by giving more context to make
                // transcription make more sense.

                if (result.isPartial && showPartial) {
                    print("[Partial] \(transcript)")
                }

                // When the complete fragment of transcribed text is ready,
                // print it. This could just as easily be used to draw the
                // text as a subtitle over a playing video, though timing
                // would need to be managed.

                if !result.isPartial {
                    if (showPartial) {
                        print("[Final  ] ", terminator: "")
                    }
                    print(transcript)
                }
            }
            default:
                print("Error: Unexpected message from Amazon Transcribe:")
            }
        }
    }
}

// -MARK: - Entry point

/// The program's asynchronous entry point.
@main
struct Main {
    static func main() async {
        let args = Array(CommandLine.arguments.dropFirst())

        do {
            let command = try ExampleCommand.parse(args)
            try await command.transcribe()
```

```
        } catch let error as TranscribeError {
            print("ERROR: \(error.errorDescription ?? "Unknown error")")
        } catch {
            ExampleCommand.exit(withError: error)
        }
    }
}

/// Errors thrown by the example's functions.
enum TranscribeError: Error {
    /// No transcription stream available.
    case noTranscriptionStream
    /// The source media file couldn't be read.
    case readError

    var errorDescription: String? {
        switch self {
        case .noTranscriptionStream:
            return "No transcription stream returned by Amazon Transcribe."
        case .readError:
            return "Unable to read the source audio file."
        }
    }
}
```

- For API details, see StartStreamTranscription in *AWS SDK for Swift API reference*.

Security in AWS SDK for Swift

Cloud security at Amazon Web Services (AWS) is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations. Security is a shared responsibility between AWS and you. The Shared Responsibility Model describes this as Security of the Cloud and Security in the Cloud.

Security of the Cloud – AWS is responsible for protecting the infrastructure that runs all of the services offered in the AWS Cloud and providing you with services that you can use securely. Our security responsibility is the highest priority at AWS, and the effectiveness of our security is regularly tested and verified by third-party auditors as part of the AWS Compliance Programs.

Security in the Cloud – Your responsibility is determined by the AWS service you are using, and other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

This AWS product or service follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Topics

Data protection in the AWS SDK for Swift

The AWS shared responsibility model applies to data protection in AWS SDK for Swift. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ.

For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with SDK for Swift or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Identity and Access Management

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use AWS resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Authenticating with identities
- Managing access using policies
- How AWS services work with IAM
- Troubleshooting AWS identity and access

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in AWS.

Service user – If you use AWS services to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more AWS features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in AWS, see Troubleshooting AWS identity and access or the user guide of the AWS service you are using.

Service administrator – If you're in charge of AWS resources at your company, you probably have full access to AWS. It's your job to determine which AWS features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with AWS, see the user guide of the AWS service you are using.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to AWS. To view example AWS identity-based policies that you can use in IAM, see the user guide of the AWS service you are using.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Multi-factor authentication in the *AWS IAM Identity Center User Guide* and AWS Multi-factor authentication in IAM in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For

information about IAM Identity Center, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. To temporarily assume an IAM role in the AWS Management Console, you can switch from a user to an IAM role (console). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Methods to assume a role in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Create a role for a third-party identity provider (federation) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.

- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
  - **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.
  - **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.
  - **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Use an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that

support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see Service control policies in the *AWS Organizations User Guide*.

- **Resource control policies (RCPs)** – RCPs are JSON policies that you can use to set the maximum available permissions for resources in your accounts without updating the IAM policies attached to each resource that you own. The RCP limits permissions for resources in member accounts and can impact the effective permissions for identities, including the AWS account root user, regardless of whether they belong to your organization. For more information about Organizations and RCPs, including a list of AWS services that support RCPs, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How AWS services work with IAM

To get a high-level view of how AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

To learn how to use a specific AWS service with IAM, see the security section of the relevant service's User Guide.

Troubleshooting AWS identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with AWS and IAM.

Topics

- I am not authorized to perform an action in AWS
- I am not authorized to perform iam:PassRole
- I want to allow people outside of my AWS account to access my AWS resources

I am not authorized to perform an action in AWS

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional *`my-example-widget`* resource but doesn't have the fictional `awes:`*`GetWidget`* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 awes:GetWidget on resource: my-example-widget
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the *`my-example-widget`* resource by using the `awes:`*`GetWidget`* action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to AWS.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in AWS. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my AWS resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether AWS supports these features, see How AWS services work with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Compliance Validation for this AWS Product or Service

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security Compliance & Governance – These solution implementation guides discuss architectural considerations and provide steps for deploying security and compliance features.
- Architecting for HIPAA Security and Compliance on Amazon Web Services – This whitepaper describes how companies can use AWS to create HIPAA-eligible applications.

> **Note**
>
> Not all AWS services are HIPAA eligible. For more information, see the HIPAA Eligible Services Reference.

- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- AWS Customer Compliance Guides – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).
- Evaluating Resources with Rules in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your compliance against security industry standards and best practices. For a list of supported services and controls, see Security Hub controls reference.
- Amazon GuardDuty – This AWS service detects potential threats to your AWS accounts, workloads, containers, and data by monitoring your environment for suspicious and malicious activities. GuardDuty can help you address various compliance requirements, like PCI DSS, by meeting intrusion detection requirements mandated by certain compliance frameworks.
- AWS Audit Manager – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

This AWS product or service follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Resilience for this AWS Product or Service

The AWS global infrastructure is built around AWS Regions and Availability Zones.

AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking.

With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

This AWS product or service follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Infrastructure Security for this AWS Product or Service

This AWS product or service uses managed services, and therefore is protected by the AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well-Architected Framework*.

You use AWS published API calls to access this AWS Product or Service through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

This AWS product or service follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Document history for the AWS SDK for Swift Developer Guide

Updates about significant changes or changes of interest to the AWS SDK for Swift Developer Guide.

Change	Description	Date
Event streaming	Added a section on using event streaming, using Amazon Transcribe as an example.	December 26, 2024
Binary streaming	Added a section on using binary streaming for both uploads and downloads.	December 13, 2024
Presigned URLs and multipart uploads	Added to the "Using the SDK for Swift" chapter two new sections: one on using multipart uploads and another on presigned URLs and requests.	December 7, 2024
Credential identity resolvers	Added a new section on credential identity resolvers, with examples for both AWS IAM Identity Center (SSO) and static resolvers.	November 8, 2024
Sign In With Apple and paginators	Added the new section covering how to use Sign In With Apple to authenticate for AWS services. Also added a new section covering the use of Paginators with the SDK. Removed the Lambda	October 10, 2024

	runtime warning added on 2024-10-04.	
Compatibility note in the Lambda content	Added a note to the chapter on using the SDK to write Lambda functions, indicating that you temporarily can't use the SDK with the Swift AWS Lambda Runtime package.	October 4, 2024
General availability	Content has been updated for the general availability release of the SDK.	September 17, 2024
Updated logging information	The Testing and Debugging chapter now provides up-to-date information about configuring logging for both the SDK and the Common RunTime library.	August 30, 2024
Added Lambda function information	Added the section covering how to create Lambda functions using the Swift AWS Lambda Runtime from Apple's server-side Swift project.	August 21, 2024
Assorted small corrections	Fixed assorted small problems with the document.	August 19, 2024
Update package file information	Added information missing about how to set up the `Package.swift` file.	February 19, 2024

Maintenance and updates	Updated and added links to the SDK reference now that it's in place in its final home. Additionally, synchroni zed content with latest SDK changes.	February 1, 2024
Updated tool version requirements	Updated the minimum version requirements to Swift 5.7 and Xcode 14.	October 12, 2023
Rewrote section on configuri ng clients	The section named "Client configuration" has been replaced with a mostly rewritten section named "Customize client configura tions". This brings obsolete content up-to-date and provides a working example.	October 10, 2023
Content corrections	Replaced some placeholder text with the correct content and fixed a broken link.	September 12, 2023
Content additions	Renamed the chapter on logging to "Testing and debugging," and added new content covering how to mock the SDK for Swift in tests.	August 28, 2023
Added the section on using waiters	Added the new section on using waiters.	August 9, 2023

Updated setup process to use AWS IAM Identity Center	Updated guide to align with the IAM best practices . For more information, see Security best practices in IAM. This update also features general improvements and updates to bring the documentation closer to being in sync with version 0.20.0 of the SDK.	July 11, 2023
New code examples for Amazon S3 and IAM	Added new examples for Amazon S3 and IAM, cleaned up and removed obsolete information, and fixed reported content errors.	November 4, 2022
Documentation update	Minor corrections and organizational improveme nts to the AWS SDK for Swift Developer Guide.	August 19, 2022
AWS SDK for Swift Developer Preview release	AWS SDK for Swift Developer Preview draft documentation.	December 2, 2021