AWS SDK for .NET

Developer Guide

Table of Contents

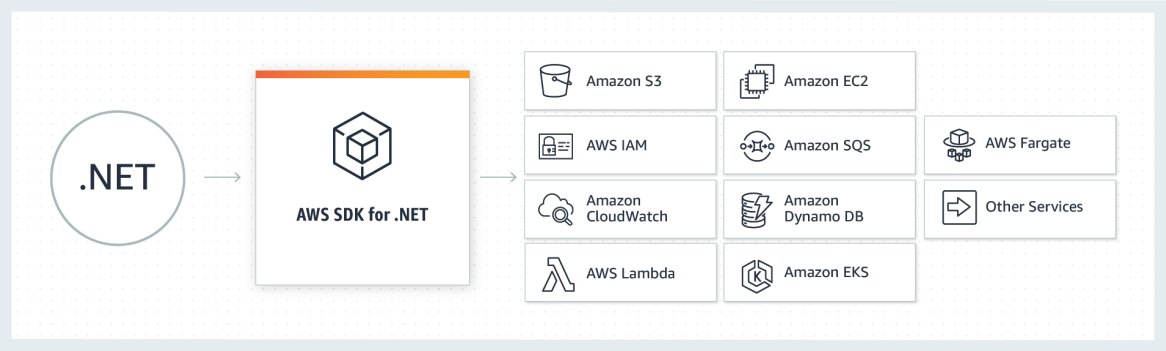
Do you want to deploy your .NET applications to AWS in just a few simple clicks? Try our new .NET CLI tooling for a simplified deployment experience!

Click here to try it now

Read our original blog post as well as the update post and the post on deployment projects. Submit your feedback on GitHub! For additional information, see the section for the deployment tool in this guide.

What is the AWS SDK for .NET

The AWS SDK for .NET makes it easier to build .NET applications that tap into cost-effective, scalable, and reliable AWS services such as Amazon Simple Storage Service (Amazon S3) and Amazon Elastic Compute Cloud (Amazon EC2). The SDK simplifies the use of AWS services by providing a set of libraries that are consistent and familiar for .NET developers..



(**OK, got it!** I'm ready for a tutorial (p. 5) or to start setting up (p. 15).)

About this version

Note
This documentation is for version 3.0 and later of the AWS SDK for .NET. It's mostly centered around .NET Core and ASP.NET Core, but also contains information about .NET Framework and ASP.NET 4.*x*. In addition to Windows and Visual Studio, it gives equal consideration to cross-platform development.
For information about migrating, see Migrating your project (p. 90).
To find deprecated content for earlier versions of the AWS SDK for .NET, see the following item(s):

- AWS SDK for .NET (version 2, deprecated) Developer Guide

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

Common use cases

The AWS SDK for .NET helps you realize several compelling use cases, including the following:

- Manage users and roles with AWS Identity and Access Management (IAM).
- Access Amazon Simple Storage Service (Amazon S3) to create buckets and store objects.

- Manage Amazon Simple Notification Service (Amazon SNS) HTTP subscriptions to topics.
- Use the S3 transfer utility to transfer files to Amazon S3 from your Xamarin applications.
- Use Amazon Simple Queue Service (Amazon SQS) to process messages and workflows between components in a system.
- Perform efficient Amazon S3 transfers by sending SQL statements to Amazon S3 Select.
- Create and launch Amazon EC2 instances, and configure and request Amazon EC2 spot instances.

Additional topics in this section

AWS tools related to the AWS SDK for .NET

Tools for Windows PowerShell and Tools for PowerShell Core

The AWS Tools for Windows PowerShell and AWS Tools for PowerShell Core are PowerShell modules that are built on the functionality exposed by the AWS SDK for .NET. The AWS PowerShell tools enable you to script operations on your AWS resources from the PowerShell prompt. Although the cmdlets are implemented using the service clients and methods from the SDK, the cmdlets provide an idiomatic PowerShell experience for specifying parameters and handling results.

To get started, see AWS Tools for Windows PowerShell.

Toolkit for VS Code

The AWS Toolkit for Visual Studio Code is a plugin for the Visual Studio Code (VS Code) editor. The toolkit makes it easier for you to develop, debug, and deploy applications that use AWS.

With the toolkit, you can do such things as the following:

- Create serverless applications that contain AWS Lambda functions, and then deploy the applications to an AWS CloudFormation stack.
- Work with Amazon EventBridge schemas.
- Use IntelliSense when working with Amazon ECS task-definition files.
- Visualize an AWS Cloud Development Kit (CDK) application.

Toolkit for Visual Studio

The AWS Toolkit for Visual Studio is a plugin for the Visual Studio IDE that makes it easier for you to develop, debug, and deploy .NET applications that use Amazon Web Services. The Toolkit for Visual Studio provides Visual Studio templates for services such as Lambda and deployment wizards for web applications and serverless applications. You can use the AWS Explorer to manage Amazon EC2 instances, work with Amazon DynamoDB tables, publish messages to Amazon Simple Notification Service (Amazon SNS) queues, and more, all within Visual Studio.

To get started, see Setting up the AWS Toolkit for Visual Studio.

Toolkit for Azure DevOps

The AWS Toolkit for Microsoft Azure DevOps adds tasks to easily enable build and release pipelines in Azure DevOps and Azure DevOps Server to work with AWS services. You can work with Amazon S3, AWS Elastic Beanstalk, AWS CodeDeploy, Lambda, AWS CloudFormation, Amazon Simple Queue Service (Amazon SQS), and Amazon SNS. You can also run commands using the Windows PowerShell module and the AWS Command Line Interface (AWS CLI).

To get started with the AWS Toolkit for Azure DevOps, see the AWS Toolkit for Microsoft Azure DevOps User Guide.

AWS SDKs and Tools Reference Guide

The AWS SDKs and Tools Reference Guide contains information that's relevant and important for many of the AWS SDKs and toolkits and the AWS CLI. The following are some examples of the information that the reference contains:

- Information about the shared AWS `config` and `credentials` files and their location.
- Setting up AWS accounts, users, and roles
- Configuration and authentication settings reference
- AWS Common Runtime (CRT) libraries
- AWS SDKs and tools maintenance policy
- AWS SDKs and tools version support matrix

Additional resources

Supported services

The AWS SDK for .NET supports most AWS infrastructure products, and more services are added frequently. For a list of the AWS services supported by the SDK, see the SDK README file.

Revision history

To find out what has changed in various releases, see the following:

- SDK change log
- What's new in the AWS SDK for .NET (p. 90)
- Document history (p. 331)

Home page for the AWS SDK for .NET

For more information about the AWS SDK for .NET, see the home page for the SDK at https://aws.amazon.com/sdk-for-net/.

SDK reference documentation

The SDK reference documentation gives you the ability to browse and search across all code included with the SDK. It provides thorough documentation and usage examples. For more information, see the AWS SDK for .NET API Reference.

AWS re:Post (formerly AWS forums)

Visit AWS re:Post, specifically the topic for the AWS SDK for .NET, to ask questions or provide feedback about AWS. Each documentation page has a **Try AWS re:Post** button at the bottom of the page that takes you to the associated re:Post topic. AWS engineers monitor the topics and respond to questions, feedback, and issues.

If you're signed in to re:Post, you can also follow a topic. To follow the topic for the AWS SDK for .NET, go to the **All Topics** page, find ".NET on AWS", and select the **Follow** button.

Toolkits

- AWS Toolkit for Visual Studio: If you use the Microsoft Visual Studio IDE, you should check out the AWS Toolkit for Visual Studio User Guide.
- AWS Toolkit for Visual Studio Code: If you use the Microsoft Visual Studio IDE, you should check out the AWS Toolkit for Visual Studio Code User Guide.

Helpful libraries, extensions and tools

Visit the aws/dotnet and aws/aws-sdk-net repositories on the GitHub website for links to libraries, tools, and resources you can use to help build .NET applications and services on AWS.

The following are some examples:

- AWS .NET Configuration Extension for Systems Manager
- AWS Extensions .NET Core Setup
- AWS Logging .NET
- Amazon Cognito Authentication Extension Library
- AWS X-Ray SDK for .NET

Other resources

The following are other resources that might prove useful:

- Developer net
- .NET Development Environment on the AWS Cloud - Quick Start Reference Deployment
- Hello, Cloud! blog
- AWS Microservice Extractor for .NET
- Porting Assistant for .NETT
- AWS SDKs and Tools Reference Guide

A quick tour of the AWS SDK for .NET

This section includes basic setup steps and tutorials for developers who are new to the AWS SDK for .NET.

For more advanced information, see the following sections: Setting up your environment (p. 15), Setting up your project (p. 17), and Code examples with guidance (p. 97)

Topics

Simple cross-platform application using the AWS SDK for .NET

This tutorial uses the AWS SDK for .NET and .NET Core for cross-platform development. The tutorial shows you how to use the SDK to list the Amazon S3 buckets that you own and, optionally, create a bucket.

Steps

Setup for this tutorial

This section provides the minimal setup needed to complete this tutorial. You shouldn't consider this to be a full setup. For that, see Setting up your AWS SDK for .NET environment (p. 15).

> **Note**
> If you've already completed any of the following steps through other tutorials or existing configuration, skip those steps.

Create an AWS account

To create an AWS account, see How do I create and activate a new Amazon Web Services account?

Create AWS credentials and a profile

To perform these tutorials, you need to create an AWS Identity and Access Management (IAM) user and obtain credentials for that user. After you have those credentials, you make them available to the SDK in your development environment. Here's how.

To create and use credentials

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose **Users**, and then choose **Add user**.
3. Provide a user name. For this tutorial, we'll use *Dotnet-Tutorial-User*.
4. Under **Select AWS access type**, select **Programmatic access**, and then choose **Next: Permissions**.
5. Choose **Attach existing policies directly**.
6. In **Search**, enter **`s3`**, and then select **AmazonS3FullAccess**.
7. Choose **Next: Tags**, **Next: Review**, and **Create user**.
8. Record the credentials for *Dotnet-Tutorial-User*. You can do so by downloading the `.csv` file or by copying and pasting the *Access key ID* and *Secret access key*.

   Warning
   Use appropriate security measures to keep these credentials safe and rotated.
9. Create or open the shared AWS credentials file. This file is `~/.aws/credentials` on Linux and macOS systems, and `%USERPROFILE%\.aws\credentials` on Windows.
10. Add the following text to the shared AWS credentials file, but replace the example ID and example key with the ones you obtained earlier. Remember to save the file.

```
[dotnet-tutorials]
aws_access_key_id = AKIAIOSFODNN7EXAMPLE
aws_secret_access_key = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

The preceding procedure is the simplest of several possibilities for authentication and authorization. For complete information, see Configure AWS credentials (p. 19).

Install other tools

You'll perform this tutorial using cross-platform tools such as the .NET command line interface (CLI). For other ways to configure your development environment, see Install and configure your toolchain (p. 15).

Required for cross-platform .NET development on Windows, Linux, or macOS:

- Microsoft .NET Core SDK, version 2.1, 3.1, or later, which includes the .NET command line interface (CLI) (**`dotnet`**) and the .NET Core runtime.

- A code editor or integrated development environment (IDE) that is appropriate for your operating system and requirements. This is typically one that provides some support for .NET Core.

  Examples include Microsoft Visual Studio Code (VS Code), JetBrains Rider, and Microsoft Visual Studio.

Create the project

1. Open the command prompt or terminal. Find or create an operating system folder under which you can create a .NET project.
2. In that folder, run the following command to create the .NET project.

```
dotnet new console --name S3CreateAndList
```

3. Go to the newly created `S3CreateAndList` folder and run the following command.

```
dotnet add package AWSSDK.S3
```

The preceding command installs the **AWSSDK.S3** NuGet package from the NuGet package manager. Because we know exactly what NuGet packages we need for this tutorial, we can perform this step now. It's also common that the required packages become known during development. When this happens, a similar command can be run at that time.

4. Add the following temporary environment variables to the environment.

Linux or macOS

```
export AWS_PROFILE='dotnet-tutorials'
export AWS_REGION='us-west-2'
```

Windows

```
set AWS_PROFILE=dotnet-tutorials
set AWS_REGION=us-west-2
```

Create the code

1. In the `S3CreateAndList` folder, find and open `Program.cs` in your code editor.
2. Replace the contents with the following code and save the file.

```
using System;
using System.Threading.Tasks;

// To interact with Amazon S3.
using Amazon.S3;
using Amazon.S3.Model;

namespace S3CreateAndList
{
  class Program
  {
    // Main method
    static async Task Main(string[] args)
    {
      // Before running this app:
      // - Credentials must be specified in an AWS profile. If you use a profile other
 than
      //   the [default] profile, also set the AWS_PROFILE environment variable.
      // - An AWS Region must be specified either in the [default] profile
      //   or by setting the AWS_REGION environment variable.

      // Create an S3 client object.
      var s3Client = new AmazonS3Client();

      // Parse the command line arguments for the bucket name.
      if(GetBucketName(args, out String bucketName))
      {
        // If a bucket name was supplied, create the bucket.
        // Call the API method directly
        try
        {
          Console.WriteLine($"\nCreating bucket {bucketName}...");
          var createResponse = await s3Client.PutBucketAsync(bucketName);
          Console.WriteLine($"Result: {createResponse.HttpStatusCode.ToString()}");
```

```
        }
        catch (Exception e)
        {
          Console.WriteLine("Caught exception when creating a bucket:");
          Console.WriteLine(e.Message);
        }
      }

      // List the buckets owned by the user.
      // Call a class method that calls the API method.
      Console.WriteLine("\nGetting a list of your buckets...");
      var listResponse = await MyListBucketsAsync(s3Client);
      Console.WriteLine($"Number of buckets: {listResponse.Buckets.Count}");
      foreach(S3Bucket b in listResponse.Buckets)
      {
        Console.WriteLine(b.BucketName);
      }
    }


    //
    // Method to parse the command line.
    private static Boolean GetBucketName(string[] args, out String bucketName)
    {
      Boolean retval = false;
      bucketName = String.Empty;
      if (args.Length == 0)
      {
        Console.WriteLine("\nNo arguments specified. Will simply list your Amazon S3
buckets." +
          "\nIf you wish to create a bucket, supply a valid, globally unique bucket
name.");
        bucketName = String.Empty;
        retval = false;
      }
      else if (args.Length == 1)
      {
        bucketName = args[0];
        retval = true;
      }
      else
      {
        Console.WriteLine("\nToo many arguments specified." +
          "\n\ndotnet_tutorials - A utility to list your Amazon S3 buckets and
optionally create a new one." +
          "\n\nUsage: S3CreateAndList [bucket_name]" +
          "\n - bucket_name: A valid, globally unique bucket name." +
          "\n - If bucket_name isn't supplied, this utility simply lists your
buckets.");
        Environment.Exit(1);
      }
      return retval;
    }


    //
    // Async method to get a list of Amazon S3 buckets.
    private static async Task<ListBucketsResponse> MyListBucketsAsync(IAmazonS3
s3Client)
    {
      return await s3Client.ListBucketsAsync();
    }

  }
}
```

Run the application

1. Run the following command.

```
dotnet run
```

2. Examine the output to see the number of Amazon S3 buckets that you own, if any, and their names.
3. Choose a name for a new Amazon S3 bucket. Use "dotnet-quicktour-s3-1-cross-" as a base and add something unique to it, such as a GUID or your name. Be sure to follow the rules for bucket names, as described in Rules for bucket naming in the Amazon Simple Storage Service User Guide.
4. Run the following command, replacing *BUCKET-NAME* with the name of the bucket that you chose.

```
dotnet run BUCKET-NAME
```

5. Examine the output to see the new bucket that was created.

Clean up

While performing this tutorial, you created a few resources that you can choose to clean up at this time.

- If you don't want to keep the bucket that the application created in an earlier step, delete it by using the Amazon S3 console at https://console.aws.amazon.com/s3/.
- If you don't want to keep the user you created during tutorial setup earlier in this topic, delete it by using the IAM console at https://console.aws.amazon.com/iam/home#/users.

  If you do choose to delete the user, you should also remove the **`dotnet-tutorials`** profile that you created in the shared AWS credentials file. You created this profile during tutorial setup earlier in this topic.
- If you don't want to keep your .NET project, remove the `S3CreateAndList` folder from your development environment.

Where to go next

Go back to the quick-tour menu (p. 5) or go straight to the end of this quick tour (p. 14).

Simple Windows-based application using the AWS SDK for .NET

This tutorial uses the AWS SDK for .NET on Windows with Visual Studio and .NET Core. The tutorial shows you how to use the SDK to list the Amazon S3 buckets that you own and optionally create a bucket.

Steps

- Clean up (p. 13)

Setup for this tutorial

This section provides the minimal setup needed to complete this tutorial. You shouldn't consider this to be a full setup. For that, see Setting up your AWS SDK for .NET environment (p. 15).

> **Note**
> If you've already completed any of the following steps through other tutorials or existing configuration, skip those steps.

Create an AWS account

To create an AWS account, see How do I create and activate a new Amazon Web Services account?

Create AWS credentials and a profile

To perform these tutorials, you need to create an AWS Identity and Access Management (IAM) user and obtain credentials for that user. After you have those credentials, you make them available to the SDK in your development environment. Here's how.

To create and use credentials

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose **Users**, and then choose **Add user**.
3. Provide a user name. For this tutorial, we'll use *Dotnet-Tutorial-User*.
4. Under **Select AWS access type**, select **Programmatic access**, and then choose **Next: Permissions**.
5. Choose **Attach existing policies directly**.
6. In **Search**, enter **`s3`**, and then select **AmazonS3FullAccess**.
7. Choose **Next: Tags**, **Next: Review**, and **Create user**.
8. Record the credentials for *Dotnet-Tutorial-User*. You can do so by downloading the `.csv` file or by copying and pasting the *Access key ID* and *Secret access key*.

   > **Warning**
   > Use appropriate security measures to keep these credentials safe and rotated.
9. Create or open the shared AWS credentials file. This file is `~/.aws/credentials` on Linux and macOS systems, and `%USERPROFILE%\.aws\credentials` on Windows.
10. Add the following text to the shared AWS credentials file, but replace the example ID and example key with the ones you obtained earlier. Remember to save the file.

```
[dotnet-tutorials]
aws_access_key_id = AKIAIOSFODNN7EXAMPLE
aws_secret_access_key = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

The preceding procedure is the simplest of several possibilities for authentication and authorization. For complete information, see Configure AWS credentials (p. 19).

Install other tools

You'll perform this tutorial on Windows using Visual Studio and .NET Core. For other ways to configure your development environment, see Install and configure your toolchain (p. 15).

Required for development on Windows with Visual Studio and .NET Core:

- Microsoft Visual Studio

- Microsoft .NET Core 2.1, 3.1 or later

  This is typically included by default when installing a recent version of Visual Studio.

Create the project

1. Open Visual Studio and create a new project that uses the C# version of the **Console Application** template; that is, with description: "...for creating a command-line application that can run on .NET Core...". Name the project `S3CreateAndList`.

   Note
   Don't choose the .NET Framework version of the console app template, or, if you do, be sure to use .NET Framework 4.5 or later.
2. With the newly created project loaded, choose **Tools**, **NuGet Package Manager**, **Manage NuGet Packages for Solution**.
3. Browse for the **AWSSDK.S3** NuGet package and install it into the project.

   This process installs the **AWSSDK.S3** NuGet package from the NuGet package manager. Because we know exactly what NuGet packages we need for this tutorial, we can perform this step now. It's also common that the required packages become known during development. When this happens, follow a similar process to install them at that time.
4. If you intend to run the application from the command prompt, open a command prompt now and navigate to the folder that will contain the build output. This is typically something like `S3CreateAndList\S3CreateAndList\bin\Debug\net6.0`, but will depend on your environment.
5. Add the following temporary environment variables to the environment.

   In the command prompt, use the following.

   ```
   set AWS_PROFILE=dotnet-tutorials
   set AWS_REGION=us-west-2
   ```

   Or, if you intend to run the application in the IDE, go to **Debug**, **S3CreateAndList Debug Properties** and set them there. (In older versions of Visual Studio go to **Project**, **S3CreateAndList Properties**, **Debug**.)

Create the code

1. In the `S3CreateAndList` project, find and open `Program.cs` in the IDE.
2. Replace the contents with the following code and save the file.

   ```
   using System;
   using System.Threading.Tasks;

   // To interact with Amazon S3.
   using Amazon.S3;
   using Amazon.S3.Model;

   namespace S3CreateAndList
   {
   ```

```
 class Program
 {
   // Main method
   static async Task Main(string[] args)
   {
     // Before running this app:
     // - Credentials must be specified in an AWS profile. If you use a profile other
than
     //   the [default] profile, also set the AWS_PROFILE environment variable.
     // - An AWS Region must be specified either in the [default] profile
     //   or by setting the AWS_REGION environment variable.

     // Create an S3 client object.
     var s3Client = new AmazonS3Client();

     // Parse the command line arguments for the bucket name.
     if(GetBucketName(args, out String bucketName))
     {
       // If a bucket name was supplied, create the bucket.
       // Call the API method directly
       try
       {
         Console.WriteLine($"\nCreating bucket {bucketName}...");
         var createResponse = await s3Client.PutBucketAsync(bucketName);
         Console.WriteLine($"Result: {createResponse.HttpStatusCode.ToString()}");
       }
       catch (Exception e)
       {
         Console.WriteLine("Caught exception when creating a bucket:");
         Console.WriteLine(e.Message);
       }
     }

     // List the buckets owned by the user.
     // Call a class method that calls the API method.
     Console.WriteLine("\nGetting a list of your buckets...");
     var listResponse = await MyListBucketsAsync(s3Client);
     Console.WriteLine($"Number of buckets: {listResponse.Buckets.Count}");
     foreach(S3Bucket b in listResponse.Buckets)
     {
       Console.WriteLine(b.BucketName);
     }
   }


   //
   // Method to parse the command line.
   private static Boolean GetBucketName(string[] args, out String bucketName)
   {
     Boolean retval = false;
     bucketName = String.Empty;
     if (args.Length == 0)
     {
       Console.WriteLine("\nNo arguments specified. Will simply list your Amazon S3
buckets." +
         "\nIf you wish to create a bucket, supply a valid, globally unique bucket
name.");
       bucketName = String.Empty;
       retval = false;
     }
     else if (args.Length == 1)
     {
       bucketName = args[0];
       retval = true;
     }
     else
```

```
      {
        Console.WriteLine("\nToo many arguments specified." +
          "\n\ndotnet_tutorials - A utility to list your Amazon S3 buckets and
 optionally create a new one." +
          "\n\nUsage: S3CreateAndList [bucket_name]" +
          "\n - bucket_name: A valid, globally unique bucket name." +
          "\n - If bucket_name isn't supplied, this utility simply lists your
 buckets.");
        Environment.Exit(1);
      }
      return retval;
    }


    //
    // Async method to get a list of Amazon S3 buckets.
    private static async Task<ListBucketsResponse> MyListBucketsAsync(IAmazonS3
 s3Client)
    {
      return await s3Client.ListBucketsAsync();
    }

  }
}
```

3. Build the application.

   Note
   If you're using an older version of Visual Studio, you might get a build error similar to the following:
   "Feature 'async main' is not available in C# 7.0. Please use language version 7.1 or greater."
   If you get this error, set up your project to use a later version of the language. This is typically done in the project properties, **Build**, **Advanced**.

Run the application

1. Run the application with no command line arguments. Do this either in the command prompt (if you opened one earlier) or from the IDE.
2. Examine the output to see the number of Amazon S3 buckets that you own, if any, and their names.
3. Choose a name for a new Amazon S3 bucket. Use "dotnet-quicktour-s3-1-winvs-" as a base and add something unique to it, such as a GUID or your name. Be sure to follow the rules for bucket names, as described in Rules for Bucket Naming in the Amazon Simple Storage Service User Guide.
4. Run the application again, this time supplying the bucket name.

   In the command line, replace *BUCKET-NAME* in the following command with the name of the bucket that you chose.

   ```
   S3CreateAndList BUCKET-NAME
   ```

   Or, if you are running the application in the IDE, choose **Project**, **S3CreateAndList Properties**, **Debug** and enter the bucket name there.
5. Examine the output to see the new bucket that was created.

Clean up

While performing this tutorial, you created a few resources that you can choose to clean up at this time.

- If you don't want to keep the bucket that the application created in an earlier step, delete it by using the Amazon S3 console at https://console.aws.amazon.com/s3/.
- If you don't want to keep the user you created during tutorial setup earlier in this topic, delete it by using the IAM console at https://console.aws.amazon.com/iam/home#/users.

  If you do choose to delete the user, you should also remove the **`dotnet-tutorials`** profile that you created in the shared AWS credentials file. You created this profile during tutorial setup earlier in this topic.
- If you don't want to keep your .NET project, remove the `S3CreateAndList` folder from your development environment.

Where to go next

Go back to the quick-tour menu (p. 5) or go straight to the end of this quick tour (p. 14).

Next steps

Be sure to clean up any leftover resources that you created while performing these tutorials. These might be AWS resources or resources in your development environment such as files and folders.

Now that you've toured the AWS SDK for .NET, you might want to look at more advanced setup (p. 15).

Setting up your AWS SDK for .NET environment

This section shows you how to set up the global features and configuration for the AWS SDK for .NET.

If you're new to .NET development on AWS or at least new to the AWS SDK for .NET, check out the Quick tour (p. 5) topic first. It gives you an introduction to the SDK.

When you are finished with these topics, you can move on to Setting up your project (p. 17).

Topics

Create an AWS account

To use the AWS SDK for .NET to access AWS services, you need an AWS account and AWS credentials.

1. **Create an account.**

   To create an AWS account, see How do I create and activate a new Amazon Web Services account?
2. **Create an administrative user.**

   Avoid using your root user account (the initial account you create) to access the management console and services. Instead, create an administrative user account, as explained in Creating your first IAM admin user and group.

   After you create the administrative user account and record the login details, **sign out of your root user account** and sign back in using the administrative account.

If you need to close your AWS account, see Closing an account.

Next step

Install and configure your toolchain (p. 15)

Additional information

For additional information about how to handle certificates and security, see IAM best practices and use cases in the IAM User Guide.

For information about adding AWS credentials to your applications, see Configure AWS credentials (p. 19).

Install and configure your toolchain

To use the AWS SDK for .NET, you must have certain development tools installed.

Note
If you performed the quick start for the SDK (p. 5), you might already have some of these tools installed. If you didn't do the quick start and are new to .NET development on AWS, consider doing that first for an introduction to the AWS SDK for .NET.

Cross-platform development

The following are required for cross-platform .NET development on Windows, Linux, or macOS:

- Microsoft .NET Core SDK, version 2.1, 3.1, or later, which includes the .NET command line interface (CLI) (**`dotnet`**) and the .NET Core runtime.

- A code editor or integrated development environment (IDE) that is appropriate for your operating system and requirements. This is typically one that provides some support for .NET Core.

  Examples include Microsoft Visual Studio Code (VS Code), JetBrains Rider, and Microsoft Visual Studio.

- (Optional) An AWS toolkit if one is available for the editor you chose and your operating system.

  Examples include the AWS Toolkit for Visual Studio Code, AWS Toolkit for JetBrains, and AWS Toolkit for Visual Studio.

Windows with Visual Studio and .NET Core

The following are required for development on Windows with Visual Studio and .NET Core:

- Microsoft Visual Studio

- Microsoft .NET Core 2.1, 3.1 or later

  This is typically included by default when installing a recent version of Visual Studio.

- (Optional) The AWS Toolkit for Visual Studio, which is a plugin that provides a user interface for managing your AWS resources and local profiles from Visual Studio. To install the toolkit, see Setting up the AWS Toolkit for Visual Studio.

  For more information, see the AWS Toolkit for Visual Studio User Guide.

Next step

Setting up your project (p. 17)

Setting up your AWS SDK for .NET project

In addition to setting up your environment (p. 15), you need to configure each project that you create.

There are a few essential things your application needs to access AWS services through the AWS SDK for .NET:

- An appropriate user account or role
- Credentials for that user account or to assume that role
- Specification of the AWS Region
- AWSSDK packages or assemblies

Some of the topics in this section provide information about how to configure these essential things.

Other topics in this section and other sections provide information about more advanced ways that you can configure your project.

Topics

Start a new project

There are several techniques you can use to start a new project to access AWS services. The following are some of those techniques:

- If you're new to .NET development on AWS or at least new to the AWS SDK for .NET, you can see complete examples in Quick tour (p. 5). It gives you an introduction to the SDK.

- You can start a basic project by using the .NET CLI. To see an example of this, open a command prompt or terminal, create a folder or directory and navigate to it, and then enter the following.

```
dotnet new console --name [SOME-NAME]
```

  An empty project is created to which you can add code and NuGet packages. For more information, see the .NET Core guide.

  To see a list of project templates, use the following: `dotnet new --list`

- The AWS Toolkit for Visual Studio includes C# project templates for a variety of AWS services. After you install the toolkit in Visual Studio, you can access the templates while creating a new project.

  To see this, go to Working with AWS services in the AWS Toolkit for Visual Studio User Guide. Several of the examples in that section create new projects.

- If you develop with Visual Studio on Windows but without the AWS Toolkit for Visual Studio, use your typical techniques for creating a new project.

  To see an example, open Visual Studio and choose **File**, **New**, **Project**. Search for ".net core" and choose the C# version of the **Console App (.NET Core)** or **WPF App (.NET Core)** template. An empty project is created to which you can add code and NuGet packages.

After you create your project, perform additional appropriate tasks for setting up your project (p. 17).

You can find some examples of how to work with AWS services in Code examples with guidance (p. 97).

Create users and roles

As a result of creating an AWS account (p. 15), you have (at least) two user accounts:

- Your root user account, which was created for you and has full access to everything.
- An administrative user account, which you created and gave full access to *almost* everything.

Neither of these user accounts is appropriate for doing .NET development on AWS or for running .NET applications on AWS. As such, you need to create user accounts and service roles that are appropriate for these tasks.

The specific user accounts and service roles that you create, and the way in which you use them, will depend on the requirements of your applications. The following are some of the simplest types of user accounts and service roles, and some information about why they might be used and how to create them.

User accounts

You can use a user account with long-term credentials to access AWS services through your application. This type of access is appropriate if a single user will be using your application (you, for example). The most common scenario for using this type of access is during development, but other scenarios are possible.

The process for creating a user varies depending on the situation, but is essentially the following.

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose **Users**, and then choose **Add user**.
3. Provide a user name.
4. Under **Select AWS access type**, select **Programmatic access**, and then choose **Next: Permissions**.
5. Choose **Attach existing policies directly**, and then select the appropriate policies for the AWS services that your application will use.

   Warning
   Do ***NOT*** choose the **AdministratorAccess** policy because that policy enables read and write permissions to almost everything in your account.

6. Choose **Next: Tags** and enter any tags you want.

   You can find information about tags in Control access using AWS resource tags in the IAM User Guide.
7. Choose **Next: Review**, and then choose **Create user**.
8. Record the credentials for the new user. You can do this by downloading the cleartext `.csv` file or by copying and pasting the *access key ID* and *secret access key*.

   These are the credentials that you will need for your application.

   Warning
   Use appropriate security measures to keep these credentials safe and rotated.

You can find high-level information about IAM users in Identities (users, groups, and roles) in the IAM User Guide. Find detailed information about users in that guide's IAM users topic.

Service roles

You can set up an AWS service role to access AWS services on behalf of users. This type of access is appropriate if multiple people will be running your application remotely; for example, on an Amazon EC2 instance that you have created for this purpose.

The process for creating a service role varies depending on the situation, but is essentially the following.

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose **Roles**, and then choose **Create role**.
3. Choose **AWS service**, find and select **EC2** (for example), and then choose the **EC2** use case (for example).
4. Choose **Next: Permissions**, and select the appropriate policies for the AWS services that your application will use.

   Warning
   Do ***NOT*** choose the **AdministratorAccess** policy because that policy enables read and write permissions to almost everything in your account.
5. Choose **Next: Tags** and enter any tags you want.

   You can find information about tags in Control access using AWS resource tags in the IAM User Guide.
6. Choose **Next: Review** and provide a **Role name** and **Role description**. Then choose **Create role**.

You can find high-level information about IAM roles in Identities (users, groups, and roles) in the IAM User Guide. Find detailed information about roles in that guide's IAM roles topic.

Configure AWS credentials

After you create an AWS account (p. 15) and create the required user accounts (p. 18), you can manage credentials for those user accounts. You need these credentials to perform many of the tasks and examples in this guide.

The following is a high-level process for credential management and use.

1. Create the credentials you need:

   - You can create credentials when you're creating a user account. See User accounts (p. 18) for an example.
   - You can also create credentials for an existing user account. See Managing access keys for IAM users.
2. Store the credentials (for example, in the shared AWS credentials file (p. 20) or the SDK Store (p. 23)).
3. Configure your project so that your application can find the credentials (p. 24).

The following topics provide information you can use to determine how to manage and use credentials in your environment.

Topics

Important warnings and guidance for credentials

Warnings for credentials

- ***Do NOT*** use your account's root credentials to access AWS resources. These credentials provide unrestricted account access and are difficult to revoke.
- ***Do NOT*** put literal access keys in your application files. If you do, you create a risk of accidentally exposing your credentials if, for example, you upload the project to a public repository.
- ***Do NOT*** include files that contain credentials in your project area.
- Credentials in one of the credential-storage mechanisms, the shared AWS credentials file, are stored in plaintext.

Additional guidance for securely managing credentials

For a general discussion of how to securely manage AWS credentials, see Best practices for managing AWS access keys in the AWS General Reference. In addition to that discussion, consider the following:

- Create IAM users and use their credentials instead of using your AWS root user. IAM user credentials can be revoked if necessary. In addition, you can apply a policy to each IAM user for access to certain resources and actions.
- Use IAM roles for tasks for Amazon Elastic Container Service (Amazon ECS) tasks.
- Use IAM roles for applications that are running on Amazon EC2 instances.
- Use temporary credentials (p. 26) or environment variables for applications that are available to users outside your organization.

Using the shared AWS credentials file

(Be sure to review the important warnings and guidance for credentials (p. 20).)

One way to provide credentials for your applications is to create profiles in the *shared AWS credentials file* and then store credentials in those profiles. This file can be used by the other AWS SDKs. It can also be used by the AWS CLI, the AWS Tools for Windows PowerShell, and the AWS toolkits for Visual Studio, JetBrains, and VS Code.

General information

By default, the shared AWS credentials file is located in the `.aws` directory within your home directory and is named `credentials`; that is, `~/.aws/credentials` (Linux or macOS) or `%USERPROFILE%\.aws\credentials` (Windows). For information about alternative locations, see Location of the shared files in the *AWS SDKs and Tools Reference Guide*. Also see Accessing credentials and profiles in an application (p. 326).

The shared AWS credentials file is a plaintext file and follows a certain format. For information about the format of AWS credentials files, see Format of the credentials file in the *AWS SDKs and Tools Reference Guide*.

You can manage the profiles in the shared AWS credentials file in several ways.

- Use any text editor to create and update the shared AWS credentials file.
- Use the Amazon.Runtime.CredentialManagement namespace of the AWS SDK for .NET API, as shown later in this topic.
- Use commands and procedures for the AWS Tools for PowerShell and the AWS toolkits for Visual Studio, JetBrains, and VS Code.
- Use AWS CLI commands; for example, `aws configure set aws_access_key_id` and `aws configure set aws_secret_access_key`.

Examples of profile management

The following sections show examples of profiles in the shared AWS credentials file. Some of the examples show the result, which can be obtained through any of the credential-management methods described earlier. Other examples show how to use a particular method.

The default profile

The shared AWS credentials file will almost always have a profile named *default*. This is where the AWS SDK for .NET looks for credentials if no other profiles are defined.

The `[default]` profile typically looks something like the following.

```
[default]
aws_access_key_id = AKIAIOSFODNN7EXAMPLE
aws_secret_access_key = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

Create a profile programmatically

This example shows you how to create a profile and save it to the shared AWS credentials file programmatically. It uses the following classes of the Amazon.Runtime.CredentialManagement namespace: CredentialProfileOptions, CredentialProfile, and SharedCredentialsFile.

```
using Amazon.Runtime.CredentialManagement;
...

// For illustrative purposes only--do not include credentials in your code.
WriteProfile("my_new_profile", "AKIAIOSFODNN7EXAMPLE", "wJalrXUtnFEMI/K7MDENG/
bPxRfiCYEXAMPLEKEY");
```

```
...

void WriteProfile(string profileName, string keyId, string secret)
{
    Console.WriteLine($"Create the [{profileName}] profile...");
    var options = new CredentialProfileOptions
    {
        AccessKey = keyId,
        SecretKey = secret
    };
    var profile = new CredentialProfile(profileName, options);
    var sharedFile = new SharedCredentialsFile();
    sharedFile.RegisterProfile(profile);
}
```

Warning
Code such as this generally shouldn't be in your application. If you include it in your application, take appropriate precautions to ensure that plaintext keys can't possibly be seen in the code, over the network, or even in computer memory.

The following is the profile that's created by this example.

```
[my_new_profile]
aws_access_key_id=AKIAIOSFODNN7EXAMPLE
aws_secret_access_key=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

Update an existing profile programmatically

This example shows you how to programmatically update the profile that was created earlier. It uses the following classes of the Amazon.Runtime.CredentialManagement namespace: CredentialProfile and SharedCredentialsFile. It also uses the RegionEndpoint class of the Amazon namespace.

```
using Amazon.Runtime.CredentialManagement;
...

AddRegion("my_new_profile", RegionEndpoint.USWest2);
...

void AddRegion(string profileName, RegionEndpoint region)
{
    var sharedFile = new SharedCredentialsFile();
    CredentialProfile profile;
    if (sharedFile.TryGetProfile(profileName, out profile))
    {
        profile.Region = region;
        sharedFile.RegisterProfile(profile);
    }
}
```

The following is the updated profile.

```
[my_new_profile]
aws_access_key_id=AKIAIOSFODNN7EXAMPLE
aws_secret_access_key=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
region=us-west-2
```

Note
You can also set the AWS Region in other locations and by using other methods. For more information, see Configure the AWS Region (p. 26).

Using the SDK Store (Windows only)

(Be sure to review the important warnings and guidelines (p. 20).)

On Windows, the *SDK Store* is another place to create profiles and store encrypted credentials for your AWS SDK for .NET application. It's located in `%USERPROFILE%\AppData\Local\AWSToolkit\RegisteredAccounts.json`. You can use the SDK Store during development as an alternative to the shared AWS credentials file (p. 20).

General information

The SDK Store provides the following benefits:

- The credentials in the SDK Store are encrypted, and the SDK Store resides in the user's home directory. This limits the risk of accidentally exposing your credentials.
- The SDK Store also provides credentials to the AWS Tools for Windows PowerShell and the AWS Toolkit for Visual Studio.

SDK Store profiles are specific to a particular user on a particular host. You can't copy them to other hosts or other users. This means that you can't reuse SDK Store profiles that are on your development machine for other hosts or developer machines. It also means that you can't use SDK Store profiles in production applications.

You can manage the profiles in the SDK Store in the following ways:

- Use the graphical user interface (GUI) in the AWS Toolkit for Visual Studio.
- Use the Amazon.Runtime.CredentialManagement namespace of the AWS SDK for .NET API, as shown later in this topic.
- Use commands from the AWS Tools for Windows PowerShell; for example, `Set-AWSCredential` and `Remove-AWSCredentialProfile`.

Examples of profile management

The following examples show you how to programmatically create and update a profile in the SDK Store.

Create a profile programmatically

This example shows you how to create a profile and save it to the SDK Store programmatically. It uses the following classes of the Amazon.Runtime.CredentialManagement namespace: CredentialProfileOptions, CredentialProfile, and NetSDKCredentialsFile.

```
using Amazon.Runtime.CredentialManagement;
...

// For illustrative purposes only--do not include credentials in your code.
WriteProfile("my_new_profile", "AKIAIOSFODNN7EXAMPLE", "wJalrXUtnFEMI/K7MDENG/
bPxRfiCYEXAMPLEKEY");
...

void WriteProfile(string profileName, string keyId, string secret)
{
    Console.WriteLine($"Create the [{profileName}] profile...");
    var options = new CredentialProfileOptions
    {
        AccessKey = keyId,
        SecretKey = secret
    };
```

```
    var profile = new CredentialProfile(profileName, options);
    var netSdkStore = new NetSDKCredentialsFile();
    netSdkStore.RegisterProfile(profile);
}
```

Warning
Code such as this generally shouldn't be in your application. If it's included in your application, take appropriate precautions to ensure that plaintext keys can't possibly be seen in the code, over the network, or even in computer memory.

The following is the profile that's created by this example.

```
"[generated GUID]" : {
    "AWSAccessKey" : "01000000D08...[etc., encrypted access key ID]",
    "AWSSecretKey" : "01000000D08...[etc., encrypted secret access key]",
    "ProfileType"  : "AWS",
    "DisplayName"  : "my_new_profile",
}
```

Update an existing profile programmatically

This example shows you how to programmatically update the profile that was created earlier. It uses the following classes of the Amazon.Runtime.CredentialManagement namespace: CredentialProfile and NetSDKCredentialsFile. It also uses the RegionEndpoint class of the Amazon namespace.

```
using Amazon.Runtime.CredentialManagement;
...

AddRegion("my_new_profile", RegionEndpoint.USWest2);
...

void AddRegion(string profileName, RegionEndpoint region)
{
    var netSdkStore = new NetSDKCredentialsFile();
    CredentialProfile profile;
    if (netSdkStore.TryGetProfile(profileName, out profile))
    {
        profile.Region = region;
        netSdkStore.RegisterProfile(profile);
    }
}
```

The following is the updated profile.

```
"[generated GUID]" : {
    "AWSAccessKey" : "01000000D08...[etc., encrypted access key ID]",
    "AWSSecretKey" : "01000000D08...[etc., encrypted secret access key]",
    "ProfileType"  : "AWS",
    "DisplayName"  : "my_new_profile",
    "Region"       : "us-west-2"
}
```

Note
You can also set the AWS Region in other locations and by using other methods. For more information, see Configure the AWS Region (p. 26).

Credential and profile resolution

The AWS SDK for .NET searches for credentials in a certain order and uses the first available set for the current application.

Credential search order

1. Credentials that are explicitly set on the AWS service client, as described in Accessing credentials and profiles in an application (p. 326).

   Note
   That topic is in the Special considerations (p. 325) section because it isn't the preferred method for specifying credentials.
2. A credentials profile with the name specified by a value in AWSConfigs.AWSProfileName.
3. A credentials profile with the name specified by the `AWS_PROFILE` environment variable.
4. The `[default]` credentials profile.
5. SessionAWSCredentials that are created from the `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_SESSION_TOKEN` environment variables, if they're all non-empty.
6. BasicAWSCredentials that are created from the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables, if they're both non-empty.
7. IAM Roles for Tasks for Amazon ECS tasks.
8. Amazon EC2 instance metadata.

If your application is running on an Amazon EC2 instance, such as in a production environment, use an IAM role as described in Granting access by using an IAM role (p. 183). Otherwise, such as in prerelease testing, store your credentials in a file that uses the AWS credentials file format that your web application has access to on the server.

Profile resolution

With two different storage mechanisms for credentials, it's important to understand how to configure the AWS SDK for .NET to use them. The AWSConfigs.AWSProfilesLocation property controls how the AWS SDK for .NET finds credential profiles.

AWSProfilesLocation	Profile resolution behavior
null (not set) or empty	Search the SDK Store if the platform supports it, and then search the shared AWS credentials file in the default location (p. 20). If the profile isn't in either of those locations, search `~/.aws/config` (Linux or macOS) or `%USERPROFILE%\.aws\config` (Windows).
The path to a file in the AWS credentials file format	Search *only* the specified file for a profile with the specified name.

Using federated user account credentials

Applications that use the AWS SDK for .NET (AWSSDK.Core version 3.1.6.0 and later) can use federated user accounts through Active Directory Federation Services (AD FS) to access AWS services by using Security Assertion Markup Language (SAML).

Federated access support means users can authenticate using your Active Directory. Temporary credentials are granted to the user automatically. These temporary credentials, which are valid for one hour, are used when your application invokes AWS services. The SDK handles management of the temporary credentials. For domain-joined user accounts, if your application makes a call but the credentials have expired, the user is reauthenticated automatically and fresh credentials are granted. (For non-domain-joined accounts, the user is prompted to enter credentials before reauthentication.)

To use this support in your .NET application, you must first set up the role profile by using a PowerShell cmdlet. To learn how, see the AWS Tools for Windows PowerShell documentation.

After you set up the role profile, reference the profile in your application. There are a number of ways to do this, one of which is by using the AWSConfigs.AWSProfileName property in the same way you would with other credential profiles.

The *AWS Security Token Service* assembly (AWSSDK.SecurityToken) provides the SAML support to obtain AWS credentials. To use federated user account credentials, be sure this assembly is available to your application.

Specifying roles or temporary credentials

For applications that run on Amazon EC2 instances, the most secure way to manage credentials is to use IAM roles, as described in Granting access by using an IAM role (p. 183).

For application scenarios in which the software executable is available to users outside your organization, we recommend that you design the software to use *temporary security credentials*. In addition to providing restricted access to AWS resources, these credentials have the benefit of expiring after a specified period of time. For more information about temporary security credentials, see the following:

- Temporary security credentials
- Amazon Cognito identity pools

Using proxy credentials

If your software communicates with AWS through a proxy, you can specify credentials for the proxy by using the `ProxyCredentials` property of the `Config` class of a service. The `Config` class of a service is typically part of the primary namespace for the service. Examples include the following: AmazonCloudDirectoryConfig in the Amazon.CloudDirectory namespace and AmazonGameLiftConfig in the Amazon.GameLift namespace.

For Amazon S3, for example, you could use code similar to the following, where {my-username} and {my-password} are the proxy user name and password specified in a NetworkCredential object.

```
AmazonS3Config config = new AmazonS3Config();
config.ProxyCredentials = new NetworkCredential("my-username", "my-password");
```

Note
Earlier versions of the SDK used `ProxyUsername` and `ProxyPassword`, but these properties are deprecated.

Configure the AWS Region

AWS Regions allow you to access AWS services that physically reside in a specific geographic region. This can be useful for redundancy and to keep your data and applications running close to where you and your users will access them.

To view the current list of all supported Regions and endpoints for each AWS service, see Service endpoints and quotas in the *AWS General Reference*. To view a list of existing Regional endpoints, see AWS service endpoints. To see detailed information about Regions, see Managing AWS Regions.

You can create an AWS service client that goes to a particular Region (p. 27). You can also configure your application with a Region that will be used for all AWS service clients (p. 27). These two cases are explained next.

Create a service client with a particular Region

You can specify the Region for any of the AWS service clients in your application. Setting the Region in this way takes precedence over any global setting for that particular service client.

Existing Region

This example shows you how to instantiate an Amazon EC2 client in an existing Region. It uses defined RegionEndpoint fields.

```
using (AmazonEC2Client ec2Client = new AmazonEC2Client(RegionEndpoint.USWest2))
{
  // Make a request to EC2 in the us-west-2 Region using ec2Client
}
```

New Region using RegionEndpoint class

This example shows you how to construct a new Region endpoint by using RegionEndpoint.GetBySystemName.

```
var newRegion = RegionEndpoint.GetBySystemName("us-west-new");
using (var ec2Client = new AmazonEC2Client(newRegion))
{
  // Make a request to EC2 in the new Region using ec2Client
}
```

New Region using the service client configuration class

This example shows you how to use the `ServiceURL` property of the service client configuration class to specify the Region; in this case, using the AmazonEC2Config class.

This technique works even if the Region endpoint doesn't follow the regular Region endpoint pattern.

```
var ec2ClientConfig = new AmazonEC2Config
{
    // Specify the endpoint explicitly
    ServiceURL = "https://ec2.us-west-new.amazonaws.com"
};

using (var ec2Client = new AmazonEC2Client(ec2ClientConfig))
{
  // Make a request to EC2 in the new Region using ec2Client
}
```

Specify a Region for all service clients

There are several ways you can specify a Region for all of the AWS service clients that your application creates. This Region is used for service clients that aren't created with a particular Region.

The AWS SDK for .NET looks for a Region value in the following order.

Profiles

Set in a profile that your application or the SDK has loaded. For more information, see Credential and profile resolution (p. 24).

Environment variables

Set in the `AWS_REGION` environment variable.

On Linux or macOS:

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

On Windows:

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

> **Note**
> If you set this environment variable for the whole system (using `export` or `setx`), it affects all SDKs and toolkits, not just the AWS SDK for .NET.

AWSConfigs class

Set as an AWSConfigs.AWSRegion property.

```
AWSConfigs.AWSRegion = "us-west-2";
using (var ec2Client = new AmazonEC2Client())
{
  // Make request to Amazon EC2 in us-west-2 Region using ec2Client
}
```

Special information about the China (Beijing) Region

To use services in the China (Beijing) Region, you must have an account and credentials that are specific to the China (Beijing) Region. Accounts and credentials for other AWS Regions won't work for the China (Beijing) Region. Likewise, accounts and credentials for the China (Beijing) Region won't work for other AWS Regions. For information about endpoints and protocols that are available in the China (Beijing) Region, see China (Beijing) Region.

Special information about new AWS services

New AWS services can be launched initially in a few Regions and then supported in other Regions. In these cases you don't need to install the latest SDK to access the new Regions for that service. You can specify newly added Regions on a per-client basis or globally, as shown earlier.

Install AWSSDK packages with NuGet

NuGet is a package management system for the .NET platform. With NuGet, you can install the AWSSDK packages, as well as several other extensions, to your project. For additional information, see the aws/dotnet repository on the GitHub website.

NuGet always has the most recent versions of the AWSSDK packages, as well as previous versions. NuGet is aware of dependencies between packages and installs all required packages automatically.

> **Warning**
> The list of NuGet packages might include one named simply "AWSSDK" (with no appended identifier). Do NOT install this NuGet package; it is legacy and should not be used for new projects.

Packages installed with NuGet are stored with your project instead of in a central location. This enables you to install assembly versions specific to a given application without creating compatibility issues for other applications. For more information about NuGet, see the NuGet documentation.

Note
If you can't or aren't allowed to download and install NuGet packages on a per-project basis, you can obtain the AWSSDK assemblies and store them locally (or on premises).
If this applies to you and you haven't already obtained the AWSSDK assemblies, see Obtaining AWSSDK assemblies (p. 325). To learn how to use the locally stored assemblies, see Install AWSSDK assemblies without NuGet (p. 30).

Using NuGet from the Command prompt or terminal

1. Go to the AWSSDK packages on NuGet and determine which packages you need in your project; for example, **AWSSDK.S3**.
2. Copy the .NET CLI command from that package's webpage, as shown in the following example.

   ```
   dotnet add package AWSSDK.S3 --version 3.3.110.19
   ```

3. In your project's directory, run that .NET CLI command. NuGet also installs any dependencies, such as AWSSDK.Core.

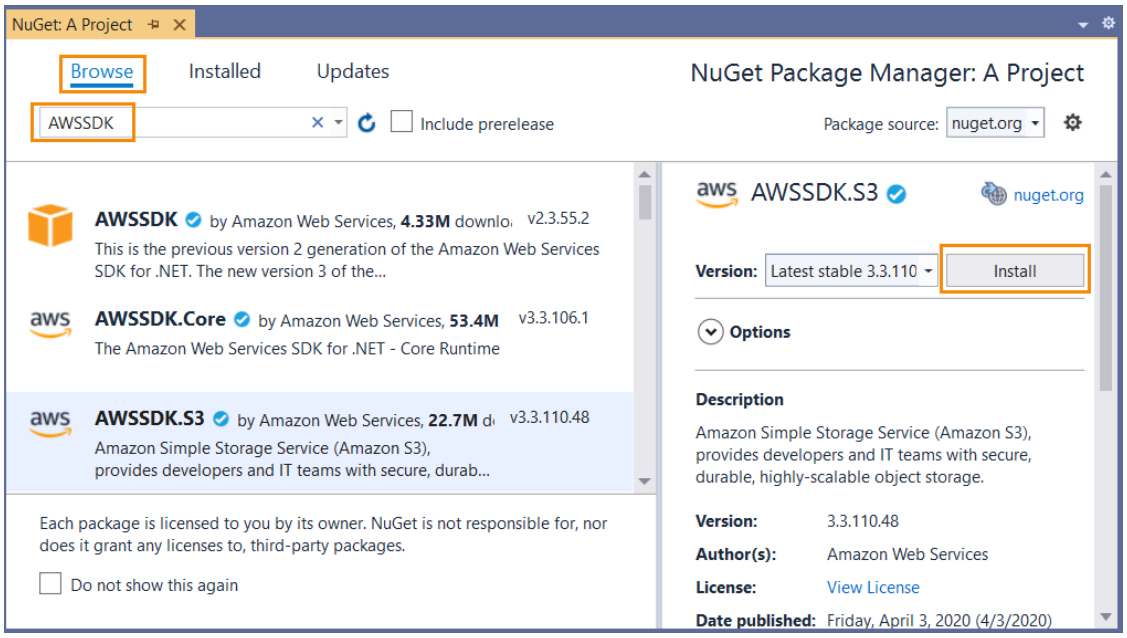
Note
If you want only the latest version of a NuGet package, you can exclude version information from the command, as shown in the following example.

```
dotnet add package AWSSDK.S3
```

Using NuGet from Visual Studio Solution Explorer

1. In **Solution Explorer**, right-click your project, and then choose **Manage NuGet Packages** from the context menu.
2. In the left pane of the **NuGet Package Manager**, choose **Browse**. You can then use the search box to search for the package you want to install. NuGet also installs any dependencies, such as AWSSDK.Core.

   The following figure shows installation of the **AWSSDK.S3** package.



Using NuGet from the Package Manager Console

In Visual Studio, choose **Tools**, **NuGet Package Manager**, **Package Manager Console**.

You can install the AWSSDK packages you want from the Package Manager Console by using the **`Install-Package`** command. For example, to install AWSSDK.S3, use the following command.

```
PM> Install-Package AWSSDK.S3
```

NuGet also installs any dependencies, such as AWSSDK.Core.

If you need to install an earlier version of a package, use the `-Version` option and specify the package version you want, as shown in the following example.

```
PM> Install-Package AWSSDK.S3 -Version 3.3.106.6
```

For more information about Package Manager Console commands, see the PowerShell reference in Microsoft's NuGet documentation.

Install AWSSDK assemblies without NuGet

This topic describes how you can use the AWSSDK assemblies that you obtained and stored locally (or on premises) as described in Obtaining AWSSDK assemblies (p. 325). This is **not** the recommended method for handling SDK references, but is required in some environments.

> **Note**
> The recommended method for handling SDK references is to download and install just the NuGet packages that each project needs. That method is described in Install AWSSDK packages with NuGet (p. 28).

To install AWSSDK assemblies

1. Create a folder in your project area for the required AWSSDK assemblies. As an example, you might call this folder `AwsAssemblies`.
2. If you haven't already done so, obtain the AWSSDK assemblies (p. 325), which places the assemblies in some local download or installation folder. Copy the DLL files for the required assemblies from that download folder into your project (into the `AwsAssemblies` folder, in our example).

   Be sure to also copy any dependencies. You can find information about dependencies on the GitHub website.
3. Make reference to the required assemblies as follows.

   Cross-platform development

   1. Open your project's `.csproj` file and add an `<ItemGroup>` element.
   2. In the `<ItemGroup>` element, add a `<Reference>` element with an `Include` attribute for each required assembly.

      For Amazon S3, for example, you would add the following lines to your project's `.csproj` file.

      On Linux and macOS:

      ```
      <ItemGroup>
        <Reference Include="./AwsAssemblies/AWSSDK.Core.dll" />
        <Reference Include="./AwsAssemblies/AWSSDK.S3.dll" />
      </ItemGroup>
      ```

      On Windows:

      ```
      <ItemGroup>
        <Reference Include="AwsAssemblies\AWSSDK.Core.dll" />
        <Reference Include="AwsAssemblies\AWSSDK.S3.dll" />
      </ItemGroup>
      ```
   3. Save you project's `.csproj` file.

   Windows with Visual Studio and .NET Core

   1. In Visual Studio, load your project and open **Project**, **Add Reference**.
   2. Choose the **Browse** button on the bottom of the dialog box. Navigate to your project's folder and the subfolder that you copied the required DLL files to (`AwsAssemblies`, for example).
   3. Select all the DLL files, choose **Add**, and choose **OK**.
   4. Save your project.

Advanced configuration for your AWS SDK for .NET project

The topics in this section contain information about additional configuration tasks and methods that might be of interest to you.

Topics

Using AWSSDK.Extensions.NETCore.Setup and the IConfiguration interface

(This topic was formerly titled, "Configuring the AWS SDK for .NET with .NET Core")

One of the biggest changes in .NET Core is the removal of `ConfigurationManager` and the standard `app.config` and `web.config` files that were used with .NET Framework and ASP.NET applications.

Configuration in .NET Core is based on key-value pairs established by configuration providers. Configuration providers read configuration data into key-value pairs from a variety of configuration sources, including command-line arguments, directory files, environment variables, and settings files.

> **Note**
> For further information, see Configuration in ASP.NET Core.

To make it easy to use the AWS SDK for .NET with .NET Core, you can use the AWSSDK.Extensions.NETCore.Setup NuGet package. Like many .NET Core libraries, it adds extension methods to the `IConfiguration` interface to make getting the AWS configuration seamless.

Using AWSSDK.Extensions.NETCore.Setup

Suppose that you create an ASP.NET Core Model-View-Controller (MVC) application, which can be accomplished with the **ASP.NET Core Web Application** template in Visual Studio or by running `dotnet new mvc ...` in the .NET Core CLI. When you create such an application, the constructor for `Startup.cs` handles configuration by reading in various input sources from configuration providers such as `appsettings.json`.

```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}
```

To use the `Configuration` object to get the *AWS* options, first add the `AWSSDK.Extensions.NETCore.Setup` NuGet package. Then, add your options to the configuration file as described next.

Notice that one of the files added to your project is `appsettings.Development.json`. This corresponds to an `EnvironmentName` set to **Development**. During development, you put your configuration in this file, which is only read during local testing. When you deploy an Amazon EC2 instance that has `EnvironmentName` set to **Production**, this file is ignored and the AWS SDK for .NET falls back to the IAM credentials and Region that are configured for the Amazon EC2 instance.

The following configuration settings show examples of the values you can add in the `appsettings.Development.json` file in your project to supply AWS settings.

```
{
  "AWS": {
    "Profile": "local-test-profile",
    "Region": "us-west-2"
  },
  "SupportEmail": "TechSupport@example.com"
```

```
}
```

To access a setting in a *CSHTML* file, use the `Configuration` directive.

```
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

<h1>Contact</h1>

<p>
    <strong>Support:</strong> <a
 href='mailto:@Configuration["SupportEmail"]'>@Configuration["SupportEmail"]</a><br />
</p>
```

To access the AWS options set in the file from code, call the `GetAWSOptions` extension method added to `IConfiguration`.

To construct a service client from these options, call `CreateServiceClient`. The following example shows how to create an Amazon S3 service client. (Be sure to add the AWSSDK.S3 NuGet package to your project.)

```
var options = Configuration.GetAWSOptions();
IAmazonS3 client = options.CreateServiceClient<IAmazonS3>();
```

You can also create multiple service clients with incompatible settings by using multiple entries in the `appsettings.Development.json` file, as shown in the following examples where the configuration for `service1` includes the `us-west-2` Region and the configuration for `service2` includes the special endpoint *URL*.

```
{
  "service1": {
    "Profile": "default",
    "Region": "us-west-2"
  },
  "service2": {
    "Profile": "default",
    "ServiceURL": "URL"
  }
}
```

You can then get the options for a specific service by using the entry in the JSON file. For example, to get the settings for `service1` use the following.

```
var options = Configuration.GetAWSOptions("service1");
```

Allowed values in appsettings file

The following app configuration values can be set in the `appsettings.Development.json` file. The field names must use the casing shown. For details on these settings, see the `AWS.Runtime.ClientConfig` class.

- Region
- Profile
- ProfilesLocation
- SignatureVersion
- RegionEndpoint
- UseHttp

- ServiceURL
- AuthenticationRegion
- AuthenticationServiceName
- MaxErrorRetry
- LogResponse
- BufferSize
- ProgressUpdateInterval
- ResignRetries
- AllowAutoRedirect
- LogMetrics
- DisableLogging
- UseDualstackEndpoint

ASP.NET Core dependency injection

The *AWSSDK.Extensions.NETCore.Setup* NuGet package also integrates with a new dependency injection system in ASP.NET Core. The `ConfigureServices` method in your application's `Startup` class is where the MVC services are added. If the application is using Entity Framework, this is also where that is initialized.

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();
}
```

Note
Background on dependency injection in .NET Core is available on the .NET Core documentation site.

The `AWSSDK.Extensions.NETCore.Setup` NuGet package adds new extension methods to `IServiceCollection` that you can use to add AWS services to the dependency injection. The following code shows you how to add the AWS options that are read from `IConfiguration` to add Amazon S3 and DynamoDB to the list of services. (Be sure to add the AWSSDK.S3 and AWSSDK.DynamoDBv2 NuGet packages to your project.)

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddDefaultAWSOptions(Configuration.GetAWSOptions());
    services.AddAWSService<IAmazonS3>();
    services.AddAWSService<IAmazonDynamoDB>();
}
```

Now, if your MVC controllers use either `IAmazonS3` or `IAmazonDynamoDB` as parameters in their constructors, the dependency injection system passes in those services.

```
public class HomeController : Controller
{
    IAmazonS3 S3Client { get; set; }

    public HomeController(IAmazonS3 s3Client)
    {
```

```
        this.S3Client = s3Client;
    }

    ...

}
```

Configuring Other Application Parameters

Note
The information in this topic is specific to projects based on .NET Framework. The `App.config` and `Web.config` files are not present by default in projects based on .NET Core.

Open to view .NET Framework content

In addition to configuring credentials (p. 19), you can configure a number of other application parameters:

- `AWSLogging` (p. 35)
- `AWSLogMetrics` (p. 36)
- `AWSRegion` (p. 36)
- `AWSResponseLogging` (p. 37)
- `AWS.DynamoDBContext.TableNamePrefix` (p. 37)
- `AWS.S3.UseSignatureVersion4` (p. 38)
- `AWSEndpointDefinition` (p. 38)
- AWS Service-Generated Endpoints (p. 39)

These parameters can be configured in the application's `App.config` or `Web.config` file. Although you can also configure these with the AWS SDK for .NET API, we recommend you use the application's `.config` file. Both approaches are described here.

For more information about use of the `<aws>` element as described later in this topic, see Configuration Files Reference for AWS SDK for .NET (p. 40).

AWSLogging

Configures how the SDK should log events, if at all. For example, the recommended approach is to use the `<logging>` element, which is a child element of the `<aws>` element:

```
<aws>
  <logging logTo="Log4Net"/>
</aws>
```

Alternatively:

```
<add key="AWSLogging" value="log4net"/>
```

The possible values are:

`None`

Turn off event logging. This is the default.

`log4net`

Log using log4net.

`SystemDiagnostics`

Log using the `System.Diagnostics` class.

You can set multiple values for the `logTo` attribute, separated by commas. The following example sets both `log4net` and `System.Diagnostics` logging in the `.config` file:

```
<logging logTo="Log4Net, SystemDiagnostics"/>
```

Alternatively:

```
<add key="AWSLogging" value="log4net, SystemDiagnostics"/>
```

Alternatively, using the AWS SDK for .NET API, combine the values of the LoggingOptions enumeration and set the AWSConfigs.Logging property:

```
AWSConfigs.Logging = LoggingOptions.Log4Net | LoggingOptions.SystemDiagnostics;
```

Changes to this setting take effect only for new AWS client instances.

AWSLogMetrics

Specifies whether or not the SDK should log performance metrics. To set the metrics logging configuration in the `.config` file, set the `logMetrics` attribute value in the `<logging>` element, which is a child element of the `<aws>` element:

```
<aws>
  <logging logMetrics="true"/>
</aws>
```

Alternatively, set the `AWSLogMetrics` key in the `<appSettings>` section:

```
<add key="AWSLogMetrics" value="true">
```

Alternatively, to set metrics logging with the AWS SDK for .NET API, set the AWSConfigs.LogMetrics property:

```
AWSConfigs.LogMetrics = true;
```

This setting configures the default `LogMetrics` property for all clients/configs. Changes to this setting take effect only for new AWS client instances.

AWSRegion

Configures the default AWS region for clients that have not explicitly specified a region. To set the region in the `.config` file, the recommended approach is to set the `region` attribute value in the `aws` element:

```
<aws region="us-west-2"/>
```

Alternatively, set the *AWSRegion* key in the `<appSettings>` section:

```
<add key="AWSRegion" value="us-west-2"/>
```

Alternatively, to set the region with the AWS SDK for .NET API, set the AWSConfigs.AWSRegion property:

```
AWSConfigs.AWSRegion = "us-west-2";
```

For more information about creating an AWS client for a specific region, see AWS Region Selection (p. 26). Changes to this setting take effect only for new AWS client instances.

AWSResponseLogging

Configures when the SDK should log service responses. The possible values are:

`Never`

Never log service responses. This is the default.

`Always`

Always log service responses.

`OnError`

Only log service responses when an error occurs.

To set the service logging configuration in the `.config` file, the recommended approach is to set the `logResponses` attribute value in the `<logging>` element, which is a child element of the `<aws>` element:

```
<aws>
  <logging logResponses="OnError"/>
</aws>
```

Alternatively, set the *AWSResponseLogging* key in the `<appSettings>` section:

```
<add key="AWSResponseLogging" value="OnError"/>
```

Alternatively, to set service logging with the AWS SDK for .NET API, set the AWSConfigs.ResponseLogging property to one of the values of the ResponseLoggingOption enumeration:

```
AWSConfigs.ResponseLogging = ResponseLoggingOption.OnError;
```

Changes to this setting take effect immediately.

`AWS.DynamoDBContext.TableNamePrefix`

Configures the default `TableNamePrefix` the `DynamoDBContext` will use if not manually configured.

To set the table name prefix in the `.config` file, the recommended approach is to set the `tableNamePrefix` attribute value in the `<dynamoDBContext>` element, which is a child element of the `<dynamoDB>` element, which itself is a child element of the `<aws>` element:

```
<dynamoDBContext tableNamePrefix="Test-"/>
```

Alternatively, set the `AWS.DynamoDBContext.TableNamePrefix` key in the `<appSettings>` section:

```
<add key="AWS.DynamoDBContext.TableNamePrefix" value="Test-"/>
```

Alternatively, to set the table name prefix with the AWS SDK for .NET API, set the AWSConfigs.DynamoDBContextTableNamePrefix property:

```
AWSConfigs.DynamoDBContextTableNamePrefix = "Test-";
```

Changes to this setting will take effect only in newly constructed instances of `DynamoDBContextConfig` and `DynamoDBContext`.

AWS.S3.UseSignatureVersion4

Configures whether or not the Amazon S3 client should use signature version 4 signing with requests.

To set signature version 4 signing for Amazon S3 in the `.config` file, the recommended approach is to set the `useSignatureVersion4` attribute of the `<s3>` element, which is a child element of the `<aws>` element:

```
<aws>
  <s3 useSignatureVersion4="true"/>
</aws>
```

Alternatively, set the `AWS.S3.UseSignatureVersion4` key to `true` in the `<appSettings>` section:

```
<add key="AWS.S3.UseSignatureVersion4" value="true"/>
```

Alternatively, to set signature version 4 signing with the AWS SDK for .NET API, set the AWSConfigs.S3UseSignatureVersion4 property to `true`:

```
AWSConfigs.S3UseSignatureVersion4 = true;
```

By default, this setting is `false`, but signature version 4 may be used by default in some cases or with some regions. When the setting is `true`, signature version 4 will be used for all requests. Changes to this setting take effect only for new Amazon S3 client instances.

AWSEndpointDefinition

Configures whether the SDK should use a custom configuration file that defines the regions and endpoints.

To set the endpoint definition file in the `.config` file, we recommend setting the `endpointDefinition` attribute value in the `<aws>` element.

```
<aws endpointDefinition="c:\config\endpoints.json"/>
```

Alternatively, you can set the *AWSEndpointDefinition* key in the `<appSettings>` section:

```
<add key="AWSEndpointDefinition" value="c:\config\endpoints.json"/>
```

Alternatively, to set the endpoint definition file with the AWS SDK for .NET API, set the AWSConfigs.EndpointDefinition property:

```
AWSConfigs.EndpointDefinition = @"c:\config\endpoints.json";
```

If no file name is provided, then a custom configuration file will not be used. Changes to this setting take effect only for new AWS client instances. The endpoint.json file is available from https://github.com/aws/aws-sdk-net/blob/master/sdk/src/Core/endpoints.json.

AWS Service-Generated Endpoints

Some AWS services generate their own endpoints instead of consuming a region endpoint. Clients for these services consume a service Url that is specific to that service and your resources. Two examples of these services are Amazon CloudSearch and AWS IoT. The following examples show how you can obtain the endpoints for those services.

Amazon CloudSearch Endpoints Example

The Amazon CloudSearch client is used for accessing the Amazon CloudSearch configuration service. You use the Amazon CloudSearch configuration service to create, configure, and manage search domains. To create a search domain, create a CreateDomainRequest object and provide the `DomainName` property. Create an AmazonCloudSearchClient object by using the request object. Call the CreateDomain method. The CreateDomainResponse object returned from the call contains a `DomainStatus` property that has both the `DocService` and `SearchService` endpoints. Create an AmazonCloudSearchDomainConfig object and use it to initialize `DocService` and `SearchService` instances of the AmazonCloudSearchDomainClient class.

```
// Create domain and retrieve DocService and SearchService endpoints
DomainStatus domainStatus;
using (var searchClient = new AmazonCloudSearchClient())
{
    var request = new CreateDomainRequest
    {
        DomainName = "testdomain"
    };
    domainStatus = searchClient.CreateDomain(request).DomainStatus;
    Console.WriteLine(domainStatus.DomainName + " created");
}

// Test the DocService endpoint
var docServiceConfig = new AmazonCloudSearchDomainConfig
{
    ServiceURL = "https://" + domainStatus.DocService.Endpoint
};
using (var domainDocService = new AmazonCloudSearchDomainClient(docServiceConfig))
{
    Console.WriteLine("Amazon CloudSearchDomain DocService client instantiated using the
 DocService endpoint");
    Console.WriteLine("DocService endpoint = " + domainStatus.DocService.Endpoint);

    using (var docStream = new FileStream(@"C:\doc_source\XMLFile4.xml", FileMode.Open))
    {
        var upload = new UploadDocumentsRequest
        {
            ContentType = ContentType.ApplicationXml,
            Documents = docStream
        };
        domainDocService.UploadDocuments(upload);
    }
}

// Test the SearchService endpoint
var searchServiceConfig = new AmazonCloudSearchDomainConfig
{
    ServiceURL = "https://" + domainStatus.SearchService.Endpoint
};
using (var domainSearchService = new AmazonCloudSearchDomainClient(searchServiceConfig))
{
    Console.WriteLine("Amazon CloudSearchDomain SearchService client instantiated using the
 SearchService endpoint");
    Console.WriteLine("SearchService endpoint = " + domainStatus.SearchService.Endpoint);
```

```
    var searchReq = new SearchRequest
    {
        Query = "Gambardella",
        Sort = "_score desc",
        QueryParser = QueryParser.Simple
    };
    var searchResp = domainSearchService.Search(searchReq);
}
```

AWS IoT Endpoints Example

To obtain the endpoint for AWS IoT, create an AmazonIoTClient object and call the DescribeEndPoint method. The returned DescribeEndPointResponse object contains the `EndpointAddress`. Create an AmazonIotDataConfig object, set the `ServiceURL` property, and use the object to instantiate the AmazonIotDataClient class.

```
string iotEndpointAddress;
using (var iotClient = new AmazonIoTClient())
{
    var endPointResponse = iotClient.DescribeEndpoint();
    iotEndpointAddress = endPointResponse.EndpointAddress;
}

var ioTdocServiceConfig = new AmazonIotDataConfig
{
    ServiceURL = "https://" + iotEndpointAddress
};
using (var dataClient = new AmazonIotDataClient(ioTdocServiceConfig))
{
    Console.WriteLine("AWS IoTData client instantiated using the endpoint from the
 IotClient");
}
```

Configuration Files Reference for AWS SDK for .NET

Note
The information in this topic is specific to projects based on .NET Framework. The `App.config` and `Web.config` files are not present by default in projects based on .NET Core.

Open to view .NET Framework content

You can use a .NET project's `App.config` or `Web.config` file to specify AWS settings, such as AWS credentials, logging options, AWS service endpoints, and AWS regions, as well as some settings for AWS services, such as Amazon DynamoDB, Amazon EC2, and Amazon S3. The following information describes how to properly format an `App.config` or `Web.config` file to specify these types of settings.

Note
Although you can continue to use the `<appSettings>` element in an `App.config` or `Web.config` file to specify AWS settings, we recommend you use the `<configSections>` and `<aws>` elements as described later in this topic. For more information about the `<appSettings>` element, see the `<appSettings>` element examples in Configuring Your AWS SDK for .NET Application (p. 17).

Note
Although you can continue to use the following AWSConfigs class properties in a code file to specify AWS settings, the following properties are deprecated and may not be supported in future releases:

- `DynamoDBContextTableNamePrefix`
- `EC2UseSignatureVersion4`

- `LoggingOptions`
- `LogMetrics`
- `ResponseLoggingOption`
- `S3UseSignatureVersion4`

In general, we recommend that instead of using `AWSConfigs` class properties in a code file to specify AWS settings, you should use the `<configSections>` and `<aws>` elements in an `App.config` or `Web.config` file to specify AWS settings, as described later in this topic. For more information about the preceding properties, see the `AWSConfigs` code examples in Configuring Your AWS SDK for .NET Application (p. 17).

Topics

Declaring an AWS Settings Section

You specify AWS settings in an `App.config` or `Web.config` file from within the `<aws>` element. Before you can begin using the `<aws>` element, you must create a `<section>` element (which is a child element of the `<configSections>` element) and set its `name` attribute to `aws` and its `type` attribute to `Amazon.AWSSection, AWSSDK.Core`, as shown in the following example:

```
<?xml version="1.0"?>
<configuration>
  ...
  <configSections>
    <section name="aws" type="Amazon.AWSSection, AWSSDK.Core"/>
  </configSections>
  <aws>
    <!-- Add your desired AWS settings declarations here. -->
  </aws>
  ...
</configuration>
```

The Visual Studio Editor does not provide automatic code completion (IntelliSense) for the `<aws>` element or its child elements.

To assist you in creating a correctly formatted version of the `<aws>` element, call the `Amazon.AWSConfigs.GenerateConfigTemplate` method. This outputs a canonical version of the `<aws>` element as a pretty-printed string, which you can adapt to your needs. The following sections describe the `<aws>` element's attributes and child elements.

Allowed Elements

The following is a list of the logical relationships among the allowed elements in an AWS settings section. You can generate the latest version of this list by calling the `Amazon.AWSConfigs.GenerateConfigTemplate` method, which outputs a canonical version of the `<aws>` element as a string you can adapt to your needs.

```
<aws
  endpointDefinition="string value"
  region="string value"
  profileName="string value"
  profilesLocation="string value">
  <logging
```

```
    logTo="None, Log4Net, SystemDiagnostics"
    logResponses="Never | OnError | Always"
    logMetrics="true | false"
    logMetricsFormat="Standard | JSON"
    logMetricsCustomFormatter="NameSpace.Class, Assembly" />
  <dynamoDB
    conversionSchema="V1 | V2">
    <dynamoDBContext
      tableNamePrefix="string value">
      <tableAliases>
        <alias
          fromTable="string value"
          toTable="string value" />
      </tableAliases>
      <map
        type="NameSpace.Class, Assembly"
        targetTable="string value">
        <property
          name="string value"
          attribute="string value"
          ignore="true | false"
          version="true | false"
          converter="NameSpace.Class, Assembly" />
      </map>
    </dynamoDBContext>
  </dynamoDB>
  <s3
    useSignatureVersion4="true | false" />
  <ec2
    useSignatureVersion4="true | false" />
  <proxy
    host="string value"
    port="1234"
    username="string value"
    password="string value" />
</aws>
```

Elements Reference

The following is a list of the elements that are allowed in an AWS settings section. For each element, its allowed attributes and parent-child elements are listed.

Topics

alias

The `<alias>` element represents a single item in a collection of one or more from-table to to-table mappings that specifies a different table than one configured for a type. This element maps to an instance of the `Amazon.Util.TableAlias` class from the

`Amazon.AWSConfigs.DynamoDBConfig.Context.TableAliases` property in the AWS SDK for .NET. Remapping is done before applying a table name prefix.

This element can include the following attributes:

`fromTable`

The from-table portion of the from-table to to-table mapping. This attribute maps to the `Amazon.Util.TableAlias.FromTable` property in the AWS SDK for .NET.

`toTable`

The to-table portion of the from-table to to-table mapping. This attribute maps to the `Amazon.Util.TableAlias.ToTable` property in the AWS SDK for .NET.

The parent of the `<alias>` element is the `<tableAliases>` element.

The `<alias>` element contains no child elements.

The following is an example of the `<alias>` element in use:

```
<alias
  fromTable="Studio"
  toTable="Studios" />
```

aws

The `<aws>` element represents the top-most element in an AWS settings section. This element can include the following attributes:

`endpointDefinition`

The absolute path to a custom configuration file that defines the AWS regions and endpoints to use. This attribute maps to the `Amazon.AWSConfigs.EndpointDefinition` property in the AWS SDK for .NET.

`profileName`

The profile name for stored AWS credentials that will be used to make service calls. This attribute maps to the `Amazon.AWSConfigs.AWSProfileName` property in the AWS SDK for .NET.

`profilesLocation`

The absolute path to the location of the credentials file shared with other AWS SDKs. By default, the credentials file is stored in the `.aws` directory in the current user's home directory. This attribute maps to the `Amazon.AWSConfigs.AWSProfilesLocation` property in the AWS SDK for .NET.

`region`

The default AWS region ID for clients that have not explicitly specified a region. This attribute maps to the `Amazon.AWSConfigs.AWSRegion` property in the AWS SDK for .NET.

The `<aws>` element has no parent element.

The `<aws>` element can include the following child elements:

- `<dynamoDB>`
- `<ec2>`
- `<logging>`
- `<proxy>`
- `<s3>`

The following is an example of the <aws> element in use:

```
<aws
  endpointDefinition="C:\Configs\endpoints.xml"
  region="us-west-2"
  profileName="development"
  profilesLocation="C:\Configs">
  <!-- ... -->
</aws>
```

dynamoDB

The <dynamoDB> element represents a collection of settings for Amazon DynamoDB. This element can include the *conversionSchema* attribute, which represents the version to use for converting between .NET and DynamoDB objects. Allowed values include V1 and V2. This attribute maps to the `Amazon.DynamoDBv2.DynamoDBEntryConversion` class in the AWS SDK for .NET. For more information, see DynamoDB Series - Conversion Schemas.

The parent of the <dynamoDB> element is the <aws> element.

The <dynamoDB> element can include the <dynamoDBContext> child element.

The following is an example of the <dynamoDB> element in use:

```
<dynamoDB
  conversionSchema="V2">
  <!-- ... -->
</dynamoDB>
```

dynamoDBContext

The <dynamoDBContext> element represents a collection of Amazon DynamoDB context-specific settings. This element can include the *tableNamePrefix* attribute, which represents the default table name prefix the DynamoDB context will use if it is not manually configured. This attribute maps to the `Amazon.Util.DynamoDBContextConfig.TableNamePrefix` property from the `Amazon.AWSConfigs.DynamoDBConfig.Context.TableNamePrefix` property in the AWS SDK for .NET. For more information, see Enhancements to the DynamoDB SDK.

The parent of the <dynamoDBContext> element is the <dynamoDB> element.

The <dynamoDBContext> element can include the following child elements:

- <alias> (one or more instances)
- <map> (one or more instances)

The following is an example of the <dynamoDBContext> element in use:

```
<dynamoDBContext
  tableNamePrefix="Test-">
  <!-- ... -->
</dynamoDBContext>
```

ec2

The <ec2> element represents a collection of Amazon EC2 settings. This element can include the *useSignatureVersion4* attribute, which specifies whether signature version 4 signing will be used for all requests (true) or whether signature version 4 signing will not be used for all requests (false, the default).

This attribute maps to the `Amazon.Util.EC2Config.UseSignatureVersion4` property from the `Amazon.AWSConfigs.EC2Config.UseSignatureVersion4` property in the AWS SDK for .NET.

The parent of the `<ec2>` element is the element.

The `<ec2>` element contains no child elements.

The following is an example of the `<ec2>` element in use:

```
<ec2
  useSignatureVersion4="true" />
```

logging

The `<logging>` element represents a collection of settings for response logging and performance metrics logging. This element can include the following attributes:

`logMetrics`

Whether performance metrics will be logged for all clients and configurations (true); otherwise, false. This attribute maps to the `Amazon.Util.LoggingConfig.LogMetrics` property from the `Amazon.AWSConfigs.LoggingConfig.LogMetrics` property in the AWS SDK for .NET.

`logMetricsCustomFormatter`

The data type and assembly name of a custom formatter for logging metrics. This attribute maps to the `Amazon.Util.LoggingConfig.LogMetricsCustomFormatter` property from the `Amazon.AWSConfigs.LoggingConfig.LogMetricsCustomFormatter` property in the AWS SDK for .NET.

`logMetricsFormat`

The format in which the logging metrics are presented (maps to the `Amazon.Util.LoggingConfig.LogMetricsFormat` property from the `Amazon.AWSConfigs.LoggingConfig.LogMetricsFormat` property in the AWS SDK for .NET).

Allowed values include:

`JSON`

Use JSON format.

`Standard`

Use the default format.

`logResponses`

When to log service responses (maps to the `Amazon.Util.LoggingConfig.LogResponses` property from the `Amazon.AWSConfigs.LoggingConfig.LogResponses` property in the AWS SDK for .NET).

Allowed values include:

`Always`

Always log service responses.

`Never`

Never log service responses.

`OnError`

Only log service responses when there are errors.

`logTo`

Where to log to (maps to the `LogTo` property from the `Amazon.AWSConfigs.LoggingConfig.LogTo` property in the AWS SDK for .NET).

Allowed values include one or more of:

`Log4Net`

Log to log4net.

`None`

Disable logging.

`SystemDiagnostics`

Log to System.Diagnostics.

The parent of the `<logging>` element is the `<aws>` element.

The `<logging>` element contains no child elements.

The following is an example of the `<logging>` element in use:

```
<logging
  logTo="SystemDiagnostics"
  logResponses="OnError"
  logMetrics="true"
  logMetricsFormat="JSON"
  logMetricsCustomFormatter="MyLib.Util.MyMetricsFormatter, MyLib" />
```

map

The `<map>` element represents a single item in a collection of type-to-table mappings from .NET types to DynamoDB tables (maps to an instance of the `TypeMapping` class from the `Amazon.AWSConfigs.DynamoDBConfig.Context.TypeMappings` property in the AWS SDK for .NET). For more information, see Enhancements to the DynamoDB SDK.

This element can include the following attributes:

`targetTable`

The DynamoDB table to which the mapping applies. This attribute maps to the `Amazon.Util.TypeMapping.TargetTable` property in the AWS SDK for .NET.

`type`

The type and assembly name to which the mapping applies. This attribute maps to the `Amazon.Util.TypeMapping.Type` property in the AWS SDK for .NET.

The parent of the `<map>` element is the `<dynamoDBContext>` element.

The `<map>` element can include one or more instances of the `<property>` child element.

The following is an example of the `<map>` element in use:

```
<map
  type="SampleApp.Models.Movie, SampleDLL"
  targetTable="Movies">
  <!-- ... -->
```

```
</map>
```

property

The <property> element represents a DynamoDB property. (This element maps to an instance of the Amazon.Util.PropertyConfig class from the AddProperty method in the AWS SDK for .NET) For more information, see Enhancements to the DynamoDB SDK and DynamoDB Attributes.

This element can include the following attributes:

attribute

The name of an attribute for the property, such as the name of a range key. This attribute maps to the Amazon.Util.PropertyConfig.Attribute property in the AWS SDK for .NET.

converter

The type of converter that should be used for this property. This attribute maps to the Amazon.Util.PropertyConfig.Converter property in the AWS SDK for .NET.

ignore

Whether the associated property should be ignored (true); otherwise, false. This attribute maps to the Amazon.Util.PropertyConfig.Ignore property in the AWS SDK for .NET.

name

The name of the property. This attribute maps to the Amazon.Util.PropertyConfig.Name property in the AWS SDK for .NET.

version

Whether this property should store the item version number (true); otherwise, false. This attribute maps to the Amazon.Util.PropertyConfig.Version property in the AWS SDK for .NET.

The parent of the <property> element is the <map> element.

The <property> element contains no child elements.

The following is an example of the <property> element in use:

```
<property
  name="Rating"
  converter="SampleApp.Models.RatingConverter, SampleDLL" />
```

proxy

The <proxy> element represents settings for configuring a proxy for the AWS SDK for .NET to use. This element can include the following attributes:

host

The host name or IP address of the proxy server. This attributes maps to the Amazon.Util.ProxyConfig.Host property from the Amazon.AWSConfigs.ProxyConfig.Host property in the AWS SDK for .NET.

password

The password to authenticate with the proxy server. This attributes maps to the Amazon.Util.ProxyConfig.Password property from the Amazon.AWSConfigs.ProxyConfig.Password property in the AWS SDK for .NET.

port

The port number of the proxy. This attributes maps to the `Amazon.Util.ProxyConfig.Port` property from the `Amazon.AWSConfigs.ProxyConfig.Port` property in the AWS SDK for .NET.

username

The user name to authenticate with the proxy server. This attributes maps to the `Amazon.Util.ProxyConfig.Username` property from the `Amazon.AWSConfigs.ProxyConfig.Username` property in the AWS SDK for .NET.

The parent of the `<proxy>` element is the `<aws>` element.

The `<proxy>` element contains no child elements.

The following is an example of the `<proxy>` element in use:

```
<proxy
  host="192.0.2.0"
  port="1234"
  username="My-Username-Here"
  password="My-Password-Here" />
```

s3

The `<s3>` element represents a collection of Amazon S3 settings. This element can include the *useSignatureVersion4* attribute, which specifies whether signature version 4 signing will be used for all requests (true) or whether signature version 4 signing will not be used for all requests (false, the default). This attribute maps to the `Amazon.AWSConfigs.S3Config.UseSignatureVersion4` property in the AWS SDK for .NET.

The parent of the `<s3>` element is the `<aws>` element.

The `<s3>` element contains no child elements.

The following is an example of the `<s3>` element in use:

```
<s3 useSignatureVersion4="true" />
```

Features of the AWS SDK for .NET

This section provides information about features of the AWS SDK for .NET that you might need to consider when creating your applications.

Be sure you have set up your project (p. 17) first.

For information about developing software for specific AWS services, see Code examples with guidance for the AWS SDK for .NET (p. 97).

Topics

AWS asynchronous APIs for .NET

The AWS SDK for .NET uses the *Task-based Asynchronous Pattern (TAP)* for its asynchronous implementation. To learn more about the TAP, see Task-based Asynchronous Pattern (TAP) on docs.microsoft.com.

This topic gives you an overview of how to use TAP in your calls to AWS service clients.

The asynchronous methods in the AWS SDK for .NET API are operations based on the `Task` class or the `Task<TResult>` class. See docs.microsoft.com for information about these classes: Task class, Task<TResult> class.

When these API methods are called in your code, they must be called within a function that is declared with the `async` keyword, as shown in the following example.

```
static async Task Main(string[] args)
{
  ...
  // Call the function that contains the asynchronous API method.
  // Could also call the asynchronous API method directly from Main
  //  because Main is declared async
  var response = await ListBucketsAsync();
  Console.WriteLine($"Number of buckets: {response.Buckets.Count}");
  ...
}

// Async method to get a list of Amazon S3 buckets.
private static async Task<ListBucketsResponse> ListBucketsAsync()
{
  ...
  var response = await s3Client.ListBucketsAsync();
  return response;
}
```

As shown in the preceding code snippet, the preferred scope for the `async` declaration is the `Main` function. Setting this `async` scope ensures that all calls to AWS service clients are required to be asynchronous. If you can't declare `Main` to be asynchronous for some reason, you can use the `async` keyword on functions other than `Main` and then call the API methods from there, as shown in the following example.

```
static void Main(string[] args)
{
  ...
  Task<ListBucketsResponse> response = ListBucketsAsync();
  Console.WriteLine($"Number of buckets: {response.Result.Buckets.Count}");
  ...
}

// Async method to get a list of Amazon S3 buckets.
private static async Task<ListBucketsResponse> ListBucketsAsync()
{
  ...
  var response = await s3Client.ListBucketsAsync();
  return response;
}
```

Notice the special `Task<>` syntax that's needed in `Main` when you use this pattern. In addition, you must use the **`Result`** member of the response to get the data.

You can see full examples of asynchronous calls to AWS service clients in the Quick tour (p. 5) section (Simple cross-platform app (p. 5) and Simple Windows-based app (p. 9)) and in Code examples with guidance (p. 97).

Retries and timeouts

The AWS SDK for .NET enables you to configure the number of retries and the timeout values for HTTP requests to AWS services. If the default values for retries and timeouts are not appropriate for your application, you can adjust them for your specific requirements, but it is important to understand how doing so will affect the behavior of your application.

To determine which values to use for retries and timeouts, consider the following:

- How should the AWS SDK for .NET and your application respond when network connectivity degrades or an AWS service is unreachable? Do you want the call to fail fast, or is it appropriate for the call to keep retrying on your behalf?
- Is your application a user-facing application or website that must be responsive, or is it a background processing job that has more tolerance for increased latencies?
- Is the application deployed on a reliable network with low latency, or is it deployed at a remote location with unreliable connectivity?

Retries

Overview

The AWS SDK for .NET can retry requests that fail due to server-side throttling or dropped connections. There are two properties of service configuration classes that you can use to specify the retry behavior of a service client. Service configuration classes inherit these properties from the abstract Amazon.Runtime.ClientConfig class of the AWS SDK for .NET API Reference:

- `RetryMode` specifies one of three retry modes, which are defined in the Amazon.Runtime.RequestRetryMode enumeration.

  The default value for your application can be controlled by using the `AWS_RETRY_MODE` environment variable or the *retry_mode* setting in the shared AWS config file.

- `MaxErrorRetry` specifies the number of retries allowed at the service client level; the SDK retries the operation the specified number of times before failing and throwing an exception.

  The default value for your application can be controlled by using the `AWS_MAX_ATTEMPTS` environment variable or the *max_attempts* setting in the shared AWS config file.

Detailed descriptions for these properties can be found in the abstract Amazon.Runtime.ClientConfig class of the AWS SDK for .NET API Reference. Each value of `RetryMode` corresponds by default to a particular value of `MaxErrorRetry`, as shown in the following table.

RetryMode	Corresponding MaxErrorRetry (Amazon DynamoDB)	Corresponding MaxErrorRetry (all others)
Legacy	10	4
Standard	10	2
Adaptive (experimental)	10	2

Behavior

When your application starts

When your application starts, default values for `RetryMode` and `MaxErrorRetry` are configured by the SDK. These default values are used when you create a service client unless you specify other values.

- If the properties aren't set in your environment, the default for `RetryMode` is configured as *Legacy* and the default for `MaxErrorRetry` is configured with the corresponding value from the preceding table.
- If the retry mode has been set in your environment, that value is used as the default for `RetryMode`. The default for `MaxErrorRetry` is configured with the corresponding value from the preceding table unless the value for maximum errors has also been set in your environment (described next).
- If the value for maximum errors has been set in your environment, that value is used as the default for `MaxErrorRetry`. Amazon DynamoDB is the exception to this rule; the default DynamoDB value for `MaxErrorRetry` is always the value from the preceding table.

As your application runs

When you create a service client, you can use the default values for `RetryMode` and `MaxErrorRetry`, as described earlier, or you can specify other values. To specify other values, create and include a service configuration object such as AmazonDynamoDBConfig or AmazonSQSConfig when you create the service client.

These values can't be changed for a service client after it has been created.

Considerations

When a retry occurs, the latency of your request is increased. You should configure your retries based on your application limits for total request latency and error rates.

Timeouts

The AWS SDK for .NET enables you to configure the request timeout and socket read/write timeout values at the service client level. These values are specified in the `Timeout` and the `ReadWriteTimeout` properties of the abstract Amazon.Runtime.ClientConfig class. These values are passed on as the

`Timeout` and `ReadWriteTimeout` properties of the HttpWebRequest objects created by the AWS service client object. By default, the `Timeout` value is 100 seconds and the `ReadWriteTimeout` value is 300 seconds.

When your network has high latency, or conditions exist that cause an operation to be retried, using long timeout values and a high number of retries can cause some SDK operations to seem unresponsive.

> **Note**
> The version of the AWS SDK for .NET that targets the portable class library (PCL) uses the HttpClient class instead of the `HttpWebRequest` class, and supports the Timeout property only.

The following are the exceptions to the default timeout values. These values are overridden when you explicitly set the timeout values.

- `Timeout` and `ReadWriteTimeout` are set to the maximum values if the method being called uploads a stream, such as AmazonS3Client.PutObjectAsync(), AmazonS3Client.UploadPartAsync(), AmazonGlacierClient.UploadArchiveAsync(), and so on.
- The versions of the AWS SDK for .NET that target .NET Framework set `Timeout` and `ReadWriteTimeout` to the maximum values for all AmazonS3Client and AmazonGlacierClient objects.
- The versions of the AWS SDK for .NET that target the portable class library (PCL) and .NET Core set `Timeout` to the maximum value for all AmazonS3Client and AmazonGlacierClient objects.

Example

The following example shows you how to specify *Standard* retry mode, a maximum of 3 retries, a timeout of 10 seconds, and a read/write timeout of 10 seconds (if applicable). The AmazonS3Client constructor is given an AmazonS3Config object.

```
var s3Client = new AmazonS3Client(
  new AmazonS3Config
  {
    Timeout = TimeSpan.FromSeconds(10),
    // NOTE: The following property is obsolete for
    //       versions of the AWS SDK for .NET that target .NET Core.
    ReadWriteTimeout = TimeSpan.FromSeconds(10),
    RetryMode = RequestRetryMode.Standard,
    MaxErrorRetry = 3
  });
```

Paginators

Some AWS services collect and store a large amount of data, which you can retrieve by using the API calls of the AWS SDK for .NET. If the amount of data you want to retrieve becomes too large for a single API call, you can break the results into more manageable pieces through the use of *pagination*.

To enable you to perform pagination, the request and response objects for many service clients in the SDK provide a *continuation token* (typically named `NextToken`). Some of these service clients also provide paginators.

Paginators enable you to avoid the overhead of the continuation token, which might involve loops, state variables, multiple API calls, and so on. When you use a paginator, you can retrieve data from an AWS service through a single line of code, a `foreach` loop's declaration. If multiple API calls are needed to retrieve the data, the paginator handles this for you.

Where do I find paginators?

Not all services provide paginators. One way to determine whether a service provides a paginator for a particular API is to look at the definition of a service client class in the AWS SDK for .NET API Reference.

For example, if you examine the definition for the AmazonCloudWatchLogsClient class, you see a `Paginators` property. This is the property that provides a paginator for Amazon CloudWatch Logs.

What do paginators give me?

Paginators contain properties that enable you to see full responses. They also typically contain one or more properties that enable to you access the most interesting portions of the responses, which we will call the *key results*.

For example, in the `AmazonCloudWatchLogsClient` mentioned earlier, the `Paginator` object contains a `Responses` property with the full DescribeLogGroupsResponse object from the API call. This `Responses` property contains, among other things, a collection of the log groups.

The Paginator object also contains one key result named `LogGroups`. This property holds just the log groups portion of the response. Having this key result enables you to reduce and simplify your code in many circumstances.

Synchronous vs. asynchronous pagination

Paginators provide both synchronous and asynchronous mechanisms for pagination. Synchronous pagination is available in .NET Framework 4.5 (or later) projects. Asynchronous pagination is available in .NET Core projects.

Because asynchronous operations and .NET Core are recommended, the example that comes next shows you asynchronous pagination. Information about how to perform the same tasks using synchronous pagination and .NET Framework 4.5 (or later) is shown after the example in Additional considerations for paginators (p. 56).

Example

The following example shows you how to use the AWS SDK for .NET to display a list of log groups. For contrast, the example shows how to do this both with and without paginators. Before looking at the full code, shown later, consider the following snippets.

Getting CloudWatch log groups without paginators

```
// Loop as many times as needed to get all the log groups
var request = new DescribeLogGroupsRequest{Limit = LogGroupLimit};
do
{
  Console.WriteLine($"Getting up to {LogGroupLimit} log groups...");
  var response = await cwClient.DescribeLogGroupsAsync(request);
  foreach(var logGroup in response.LogGroups)
  {
    Console.WriteLine($"{logGroup.LogGroupName}");
  }
  request.NextToken = response.NextToken;
} while(!string.IsNullOrEmpty(request.NextToken));
```

Getting CloudWatch log groups by using paginators

```
     // No need to loop to get all the log groups--the SDK does it for us behind the
 scenes
     var paginatorForLogGroups =
       cwClient.Paginators.DescribeLogGroups(new DescribeLogGroupsRequest());
     await foreach(var logGroup in paginatorForLogGroups.LogGroups)
     {
       Console.WriteLine(logGroup.LogGroupName);
     }
```

The results of these two snippets are exactly the same, so the advantage in using paginators can clearly be seen.

> **Note**
> Before you try to build and run the full code, be sure you have set up your environment (p. 15). Also review the information in Setting up your project (p. 17).
> You might also need the Microsoft.Bcl.AsyncInterfaces NuGet package because asynchronous paginators use the `IAsyncEnumerable` interface.

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.CloudWatch

Programming elements:

- Namespace Amazon.CloudWatch

  Class AmazonCloudWatchLogsClient
- Namespace Amazon.CloudWatchLogs.Model

  Class DescribeLogGroupsRequest

  Class DescribeLogGroupsResponse

  Class LogGroup

Full code

```
using System;
using System.Threading.Tasks;
using Amazon.CloudWatchLogs;
using Amazon.CloudWatchLogs.Model;

namespace CWGetLogGroups
{
  class Program
  {
    // A small limit for demonstration purposes
    private const int LogGroupLimit = 3;

    //
    // Main method
    static async Task Main(string[] args)
    {
```

```
      var cwClient = new AmazonCloudWatchLogsClient();
      await DisplayLogGroupsWithoutPaginators(cwClient);
      await DisplayLogGroupsWithPaginators(cwClient);
    }


    //
    // Method to get CloudWatch log groups without paginators
    private static async Task DisplayLogGroupsWithoutPaginators(IAmazonCloudWatchLogs
cwClient)
    {
      Console.WriteLine("\nGetting list of CloudWatch log groups without using
paginators...");

Console.WriteLine("-----------------------------------------------------------------");

      // Loop as many times as needed to get all the log groups
      var request = new DescribeLogGroupsRequest{Limit = LogGroupLimit};
      do
      {
        Console.WriteLine($"Getting up to {LogGroupLimit} log groups...");
        DescribeLogGroupsResponse response = await
cwClient.DescribeLogGroupsAsync(request);
        foreach(LogGroup logGroup in response.LogGroups)
        {
          Console.WriteLine($"{logGroup.LogGroupName}");
        }
        request.NextToken = response.NextToken;
      } while(!string.IsNullOrEmpty(request.NextToken));
    }


    //
    // Method to get CloudWatch log groups by using paginators
    private static async Task DisplayLogGroupsWithPaginators(IAmazonCloudWatchLogs
cwClient)
    {
      Console.WriteLine("\nGetting list of CloudWatch log groups by using paginators...");
      Console.WriteLine("-----------------------------------------------------------------");

      // Access the key results; i.e., the log groups
      // No need to loop to get all the log groups--the SDK does it for us behind the
scenes
      Console.WriteLine("\nFrom the key results...");
      Console.WriteLine("-------------------------");
      IDescribeLogGroupsPaginator paginatorForLogGroups =
        cwClient.Paginators.DescribeLogGroups(new DescribeLogGroupsRequest());
      await foreach(LogGroup logGroup in paginatorForLogGroups.LogGroups)
      {
        Console.WriteLine(logGroup.LogGroupName);
      }

      // Access the full response
      // Create a new paginator, do NOT reuse the one from above
      Console.WriteLine("\nFrom the full response...");
      Console.WriteLine("-------------------------");
      IDescribeLogGroupsPaginator paginatorForResponses =
        cwClient.Paginators.DescribeLogGroups(new DescribeLogGroupsRequest());
      await foreach(DescribeLogGroupsResponse response in paginatorForResponses.Responses)
      {
        Console.WriteLine($"Content length: {response.ContentLength}");
        Console.WriteLine($"HTTP result: {response.HttpStatusCode}");
        Console.WriteLine($"Metadata: {response.ResponseMetadata}");
        Console.WriteLine("Log groups:");
        foreach(LogGroup logGroup in response.LogGroups)
        {
```

```
            Console.WriteLine($"\t{logGroup.LogGroupName}");
        }
      }
    }
  }
}
```

Additional considerations for paginators

- **Paginators can't be used more than once**

  If you need the results of a particular AWS paginator in multiple locations in your code, you must not use a paginator object more than once. Instead, create a new paginator each time you need it. This concept is shown in the preceding example code in the `DisplayLogGroupsWithPaginators` method.

- **Synchronous pagination**

  Synchronous pagination is available for .NET Framework 4.5 (or later) projects.

  To see this, create a .NET Framework 4.5 (or later) project and copy the preceding code to it. Then simply remove the `await` keyword from the two `foreach` paginator calls, as shown in the following example.

  ```
  /*await*/ foreach(var logGroup in paginatorForLogGroups.LogGroups)
  {
    Console.WriteLine(logGroup.LogGroupName);
  }
  ```

  Build and run the project to see the same results you saw with asynchronous pagination.

Additional tools

The following are some additional tools that you can use to ease the work of developing, deploying, and maintaining your .NET applications.

AWS .NET deployment tool

This is prerelease documentation for a feature in preview release. It is subject to change.

After you've developed your cloud-native .NET Core application on a development machine, you can use the AWS .NET deployment tool for the .NET CLI to more easily deploy your application to AWS.

For more information, see the deployment tool (p. 75) in Deploying applications with the AWS SDK for .NET (p. 75).

Advanced authentication and authorization with the AWS SDK for .NET

The topics in this section provide information about advanced techniques for authentication and authorization in your AWS SDK for .NET application.

Topics

Single sign-on (SSO) with the AWS SDK for .NET

AWS Single Sign-On (AWS SSO) is a cloud-based single sign-on service that makes it easy to centrally manage SSO access to all of your AWS accounts and cloud applications. For full details, see the AWS Single Sign-On User Guide.

If you're unfamiliar with how an SDK interacts with AWS SSO, see the following information.

High-level pattern of interaction

At a high level, SDKs interact with AWS SSO in a manner similar to the following pattern:

1. AWS SSO is configured, typically through the AWS SSO console, and an SSO user is invited to participate.
2. The shared AWS `config` file on the user's computer is updated with SSO information.
3. The user signs in through AWS SSO and is given short-term credentials for the AWS Identity and Access Management (IAM) permissions that have been configured for them. This sign-in can be initiated through a non-SDK tool like the AWS CLI, or programmatically through a .NET application.
4. The user proceeds to do their work. When they run other applications that are using AWS SSO, they don't need to sign in again to open the applications.

Topics

Prerequisites

Before using AWS SSO, you must perform certain tasks, such as choosing an identity source and configuring the relevant AWS accounts and applications. For additional information, see the following:

- For general information about these tasks, see Getting started in the *AWS Single Sign-On User Guide*.

- For specific task examples, see the list of tutorials (p. 63) at the end of this topic. However, be sure to review the information in this topic before trying out the tutorials.

Setting up an SSO profile

After AWS SSO is configured in the relevant AWS account, a named profile for SSO must be added to the user's shared AWS `config` file. This profile is used to connect to the AWS SSO *user portal*, which returns short-term credentials for the IAM permissions that have been configured for the user.

The shared `config` file is typically named `%USERPROFILE%\.aws\config` in Windows and `~/.aws/config` in Linux and macOS. You can use your preferred text editor to add a new profile for SSO. Alternatively, you can use the `aws configure sso` command. For more information about this command, see Configuring the AWS CLI to use AWS Single Sign-On in the *AWS Command Line Interface User Guide*.

The new profile is similar to the following:

```
[profile my-sso-profile]
sso_start_url = https://my-sso-portal.awsapps.com/start
sso_region = us-west-2
sso_account_id = 123456789012
sso_role_name = SSOReadOnlyRole
```

The settings for the new profile are defined below. The first two settings define the AWS SSO user portal. The other two settings are a pair that, taken together, define the permissions that have been configured for a user. All four settings are required.

`sso_start_url`

The URL that points to the organization's AWS SSO user portal. To find this value, open the AWS SSO console, choose **Settings**, and find **User portal URL**.

`sso_region`

The AWS Region that contains the user portal host. This is the Region that was selected as you enabled AWS SSO. It can be different from the Regions that you use for other tasks.

For a complete list of the AWS Regions and their codes, see Regional Endpoints in the *Amazon Web Services General Reference*.

`sso_account_id`

The ID of an AWS account that was added through the AWS Organizations service. To see the list of available accounts, go to the AWS SSO console and open the **AWS accounts** page. The account ID that you choose for this setting will correspond to the value that you plan to give to the `sso_role_name` setting, which is shown next.

`sso_role_name`

The name of an AWS SSO permission set. This permission set defines the permissions that a user is given through AWS SSO.

The following procedure is one way to find the value for this setting.

1. Go to the AWS SSO console and open the **AWS accounts** page.
2. Choose an account to display its details. The account you choose will be the one that contains the SSO user or group that you want to give SSO permissions to.
3. Look at the list of users and groups that are assigned to the account and find the user or group of interest. The permission set that you specify in the `sso_role_name` setting is one of the sets associated with this user or group.

When giving a value to this setting, use the name of the permission set, not the Amazon Resource Name (ARN).

Permission sets have IAM policies and custom-permissions policies attached to them. For more information, see Permission sets in the *AWS Single Sign-On User Guide*.

Generating and using SSO tokens

To use AWS SSO, a user must first generate a temporary token and then use that token to access appropriate AWS applications and resources. For .NET applications, you can use the following methods to generate and use these temporary tokens:

- Generate a token with the AWS CLI and then use the token in .NET applications.
- Create .NET applications that generate a token first, if necessary, and then use the token.

These methods are described in the following sections and demonstrated in the tutorials (p. 63).

AWS CLI and .NET application

This section shows you how to generate a temporary SSO token by using the AWS CLI, and how to use that token in an application. For a full tutorial of this process, see Tutorial for AWS SSO using the AWS CLI and .NET applications (p. 63).

Generate an SSO token by using the AWS CLI

The following information shows you how to use the AWS CLI to generate a temporary token for later use.

After the user creates an SSO-enabled profile as shown in a previous section (p. 58), they run the `aws sso login` command from the AWS CLI. They must be sure to include the `--profile` parameter with the name of the SSO-enabled profile. This is shown in the following example:

```
aws sso login --profile my-sso-profile
```

If the user wants to generate a new temporary token after the current one expires, they can run the same command again.

Use the generated SSO token in a .NET application

The following information shows you how to use a temporary token that has already been generated.

Important
Your application must reference the following NuGet packages so that SSO resolution can work:

- `AWSSDK.SSO`
- `AWSSDK.SSOOIDC`

Failure to reference these packages will result in a *runtime* exception.

Your application creates an `AWSCredentials` object for the SSO profile, which loads the temporary credentials generated earlier by the AWS CLI. This is similar to the methods shown in Accessing credentials and profiles in an application (p. 326) and has the following form:

```
static AWSCredentials LoadSsoCredentials()
{
    var chain = new CredentialProfileStoreChain();
```

```
    if (!chain.TryGetAWSCredentials("my-sso-profile", out var credentials))
        throw new Exception("Failed to find the my-sso-profile profile");

    return credentials;
}
```

The `AWSCredentials` object is then passed to the constructor for a service client. For example:

```
var S3Client_SSO = new AmazonS3Client(LoadSsoCredentials());
```

Note
Using `AWSCredentials` to load temporary credentials isn't necessary if your application has been built to use the `[default]` profile for SSO. In that case, the application can create AWS service clients without parameters, similar to "`var client = new AmazonS3Client();`".

Tutorial for AWS SSO using the AWS CLI and .NET applications (p. 63)

.NET application only

This section shows you how to create a .NET application that generates a temporary SSO token, if necessary, and then uses that token. For a full tutorial of this process, see Tutorial for AWS SSO using only .NET applications (p. 68).

Generate and use an SSO token programmatically

In addition to using the AWS CLI, you can also generate an SSO token programmatically.

To do this, your application creates an `AWSCredentials` object for the SSO profile, which loads temporary credentials if any are available. Then, your application must cast the `AWSCredentials` object to an `SSOAWSCredentials` object and set some Options properties, including a callback method that is used to prompt the user for sign-in information, if necessary.

This method is shown in the following code example:

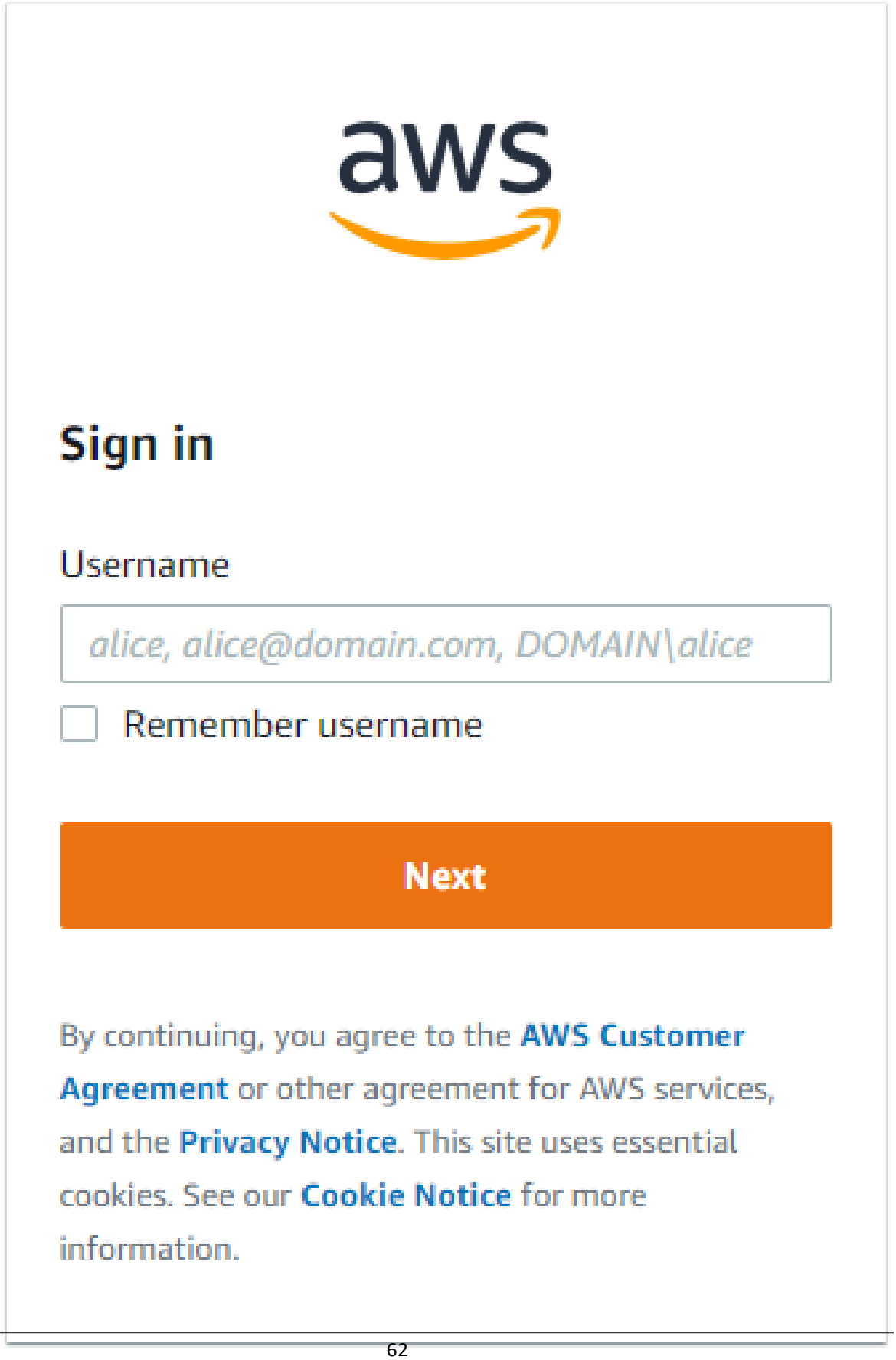
```
static AWSCredentials LoadSsoCredentials()
{
    var chain = new CredentialProfileStoreChain();
    if (!chain.TryGetAWSCredentials("my-sso-profile", out var credentials))
        throw new Exception("Failed to find the my-sso-profile profile");

    var ssoCredentials = credentials as SSOAWSCredentials;

    ssoCredentials.Options.ClientName = "Example-SSO-App";
    ssoCredentials.Options.SsoVerificationCallback = args =>
    {
        // Launch a browser window that prompts the SSO user to complete an SSO sign-in.
        // This method is only invoked if the session doesn't already have a valid SSO
 token.
        // NOTE: Process.Start might not support launching a browser on macOS or Linux. If
 not,
        //       use an appropriate mechanism on those systems instead.
        Process.Start(new ProcessStartInfo
        {
            FileName = args.VerificationUriComplete,
            UseShellExecute = true
        });
    };

    return ssoCredentials;
}
```

If an appropriate SSO token isn't available, the default browser window is launched and the appropriate sign-in page is opened. For example, if you're using AWS SSO as the **Identity source**, the user sees a sign-in page similar to the following:


aws
Sign in
Username
alice, alice@domain.com, DOMAIN\alice
Remember username
Next
By continuing, you agree to the AWS Customer Agreement or other agreement for AWS services, and the Privacy Notice. This site uses essential cookies. See our Cookie Notice for more information.

Note
The text string that you provide for `SSOAWSCredentials.Options.ClientName` can't have spaces. If the string does have spaces, you'll get a *runtime* exception.

Tutorial for AWS SSO using only .NET applications (p. 68)

Additional resources

For additional help, see the following resources.

- What is AWS Single Sign-On?
- Configuring the AWS CLI to use AWS SSO
- Using AWS SSO credentials in the AWS Toolkit for Visual Studio

Tutorials

Topics

Tutorial for AWS SSO using the AWS CLI and .NET applications

This tutorial shows you how to enable AWS Single Sign-On for a basic .NET application and a test SSO user. It uses the AWS CLI to generate a temporary SSO token instead of generating it programmatically (p. 68).

Before you start this tutorial, see the background information (p. 57) for using AWS SSO with the AWS SDK for .NET. Also see the high-level description for this scenario in the subsection called AWS CLI and .NET application (p. 59).

Several of the steps in this tutorial help you configure services like AWS Organizations and AWS SSO. If you've already performed those configurations, or if you're only interested in the code, you can skip to the section with the example code (p. 65).

Prerequisites

- Configure your development environment if you haven't already done so. This is described in sections like Install and configure your toolchain (p. 15) and Setting up your project (p. 17).
- Identify or create at least one AWS account that you can use to test AWS SSO. For the purposes of this tutorial, this is called the *test AWS account* or simply *test account*.
- Identify an *SSO user* who can test AWS SSO for you. This is a person who will be using SSO and the basic applications that you create. For this tutorial, that person might be you (the developer), or someone else. We also recommend a setup in which the SSO user is working on a computer that is not in your development environment. However, this isn't strictly necessary.
- The SSO user's computer must have a .NET framework installed that's compatible with the one you used to set up your development environment.

- Be sure that the AWS CLI version 2 is installed on the SSO user's computer. You can check this by running `aws --version` in a command prompt or terminal.

Set up AWS

This section shows you how to set up various AWS services for this tutorial.

To perform this setup, first sign in to the test AWS account as an administrator. Then, do the following:

Amazon S3

Go to the Amazon S3 console and add some innocuous buckets. Later in this tutorial, the SSO user will retrieve a list of these buckets.

AWS IAM

Go to the IAM console and add a few IAM users. If you give the IAM users permissions, limit the permissions to a few innocuous read-only permissions. Later in this tutorial, the SSO user will retrieve a list of these IAM users.

AWS Organizations

Go to the AWS Organizations console and enable Organizations. For more information, see Creating an organization in the AWS Organizations User Guide.

This action adds the test AWS account to the organization as the *management account*. If you have additional test accounts, you can invite them to join the organization, but doing so isn't necessary for this tutorial.

AWS SSO

Go to the AWS Single Sign-On console and enable AWS SSO. Perform email verification if necessary. For more information, see Enable AWS SSO in the AWS Single Sign-On User Guide.

Then, perform the following configuration.

Configure AWS SSO

1. Go to the **Settings** page. Look for the **"User portal URL"** and record the value for later use in the `sso_start_url` setting.
2. In the banner of the AWS Management Console, look for the AWS Region that was set when you enabled AWS SSO. This is the dropdown menu to the left of the AWS account ID. Record the Region code for later use in the `sso_region` setting. This code will be similar to `us-east-1`.
3. Create an SSO user as follows:
   a. Go to the **Users** page.
   b. Choose **Add user** and enter the user's **Username**, **Email address**, **First name**, and **Last name**. Then, choose **Next**.
   c. Choose **Next** on the page for groups, then review the information and choose **Add user**.
4. Create a group as follows:
   a. Go to the **Groups** page.
   b. Choose **Create group** and enter the group's **Group name** and **Description**.
   c. In the **Add users to group** section, select the test SSO user that you created earlier. Then, select **Create group.**
5. Create a permission set as follows:
   a. Go to the **Permission sets** page and choose **Create permission set**.
   b. Select **Create a custom permission set** and choose **Next: Details**.
   c. For this tutorial, enter `SSOReadOnlyRole` for the **Name** and enter a **Description**.

d. Select **Create a custom permissions policy** and enter the following policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "s3:ListAllMyBuckets",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

e. Choose **Next: Tags**, **Next: Review**, and **Create**.

f. Record the name of the permission set for later use in the `sso_role_name` setting.

6. Go to the **AWS accounts** page and choose the AWS account that you added to the organization earlier.
7. In the **Overview** section of that page, find the **Account ID** and record it for later use in the `sso_account_id` setting.
8. Choose the **Users and groups** tab and then choose **Assign users or groups**.
9. On the **Assign users and groups** page, choose the **Groups** tab, select the group that you created earlier, and choose **Next**.
10. Select the permission set that you created earlier and choose **Next**, then choose **Submit**. The configuration takes a few moments.

Create example applications

Create the following applications. They will be run on the SSO user's computer.

List Amazon S3 buckets

Include NuGet packages `AWSSDK.SSO` and `AWSSDK.SSOOIDC` in addition to `AWSSDK.S3` and `AWSSDK.SecurityToken`.

```
using System;
using System.Threading.Tasks;

// NuGet packages: AWSSDK.S3, AWSSDK.SecurityToken, AWSSDK.SSO, AWSSDK.SSOOIDC
using Amazon.Runtime;
using Amazon.Runtime.CredentialManagement;
using Amazon.S3;
using Amazon.S3.Model;
using Amazon.SecurityToken;
using Amazon.SecurityToken.Model;

namespace SSOExample.S3.CLI_login
{
    class Program
    {
        // Requirements:
        // - An SSO profile in the SSO user's shared config file.
        // - An active SSO Token.
        //    If an active SSO token isn't available, the SSO user should do the
 following:
```

```
        //    In a terminal, the SSO user must call "aws sso login --profile my-sso-
profile".

        // Class members.
        private static string profile = "my-sso-profile";
        static async Task Main(string[] args)
        {
            // Get SSO credentials from the information in the shared config file.
            var ssoCreds = LoadSsoCredentials(profile);

            // Display the caller's identity.
            var ssoProfileClient = new AmazonSecurityTokenServiceClient(ssoCreds);
            Console.WriteLine($"\nSSO Profile:\n {await
 ssoProfileClient.GetCallerIdentityArn()}");

            // Display a list of the account's S3 buckets.
            // The S3 client is created using the SSO credentials obtained earlier.
            var s3Client = new AmazonS3Client(ssoCreds);
            Console.WriteLine("\nGetting a list of your buckets...");
            var listResponse = await s3Client.ListBucketsAsync();
            Console.WriteLine($"Number of buckets: {listResponse.Buckets.Count}");
            foreach (S3Bucket b in listResponse.Buckets)
            {
                Console.WriteLine(b.BucketName);
            }
            Console.WriteLine();
        }

        // Method to get SSO credentials from the information in the shared config file.
        static AWSCredentials LoadSsoCredentials(string profile)
        {
            var chain = new CredentialProfileStoreChain();
            if (!chain.TryGetAWSCredentials(profile, out var credentials))
                throw new Exception($"Failed to find the {profile} profile");
            return credentials;
        }
    }

    // Class to read the caller's identity.
    public static class Extensions
    {
        public static async Task<string> GetCallerIdentityArn(this
 IAmazonSecurityTokenService stsClient)
        {
            var response = await stsClient.GetCallerIdentityAsync(new
 GetCallerIdentityRequest());
            return response.Arn;
        }
    }
}
```

List IAM users

Include NuGet packages `AWSSDK.SSO` and `AWSSDK.SSOOIDC` in addition to `AWSSDK.IdentityManagement` and `AWSSDK.SecurityToken`.

```
using System;
using System.Threading.Tasks;

// NuGet packages: AWSSDK.IdentityManagement, AWSSDK.SecurityToken, AWSSDK.SSO,
 AWSSDK.SSOOIDC
using Amazon.Runtime;
using Amazon.Runtime.CredentialManagement;
using Amazon.IdentityManagement;
```

```
using Amazon.IdentityManagement.Model;
using Amazon.SecurityToken;
using Amazon.SecurityToken.Model;

namespace SSOExample.IAM.CLI_login
{
    class Program
    {
        // Requirements:
        // - An SSO profile in the SSO user's shared config file.
        // - An active SSO Token.
        //    If an active SSO token isn't available, the SSO user should do the
 following:
        //    In a terminal, the SSO user must call "aws sso login --profile my-sso-
profile".

        // Class members.
        private static string profile = "my-sso-profile";
        static async Task Main(string[] args)
        {
            // Get SSO credentials from the information in the shared config file.
            var ssoCreds = LoadSsoCredentials(profile);

            // Display the caller's identity.
            var ssoProfileClient = new AmazonSecurityTokenServiceClient(ssoCreds);
            Console.WriteLine($"\nSSO Profile:\n {await
 ssoProfileClient.GetCallerIdentityArn()}");

            // Display a list of the account's IAM users.
            // The IAM client is created using the SSO credentials obtained earlier.
            var iamClient = new AmazonIdentityManagementServiceClient(ssoCreds);
            Console.WriteLine("\nGetting a list of IAM users...");
            var listResponse = await iamClient.ListUsersAsync();
            Console.WriteLine($"Number of IAM users: {listResponse.Users.Count}");
            foreach (User u in listResponse.Users)
            {
                Console.WriteLine(u.UserName);
            }
            Console.WriteLine();
        }

        // Method to get SSO credentials from the information in the shared config file.
        static AWSCredentials LoadSsoCredentials(string profile)
        {
            var chain = new CredentialProfileStoreChain();
            if (!chain.TryGetAWSCredentials(profile, out var credentials))
                throw new Exception($"Failed to find the {profile} profile");
            return credentials;
        }
    }

    // Class to read the caller's identity.
    public static class Extensions
    {
        public static async Task<string> GetCallerIdentityArn(this
 IAmazonSecurityTokenService stsClient)
        {
            var response = await stsClient.GetCallerIdentityAsync(new
 GetCallerIdentityRequest());
            return response.Arn;
        }
    }
}
```

In addition to displaying lists of Amazon S3 buckets and IAM users, these applications display the user identity ARN for the SSO-enabled profile, which is `my-sso-profile` in this tutorial.

Instruct SSO user

Ask the SSO user to check their email and accept the SSO invitation. They are prompted to set a password. The message might take a few minutes to arrive in the SSO user's inbox.

Give the SSO user the applications that you created earlier.

Then, have the SSO user do the following:

1. If the folder that contains the shared AWS `config` file doesn't exist, create it. If the folder does exist and has a subfolder called `.sso`, delete that subfolder.

   The location of this folder is typically `%USERPROFILE%\.aws` in Windows and `~/.aws` in Linux and macOS.
2. Create a shared AWS `config` file in that folder, if necessary, and add a profile to it as follows:

```
[default]
region = <default Region>

[profile my-sso-profile]
sso_start_url = <user portal URL recorded earlier>
sso_region = <Region code recorded earlier>
sso_account_id = <account ID recorded earlier>
sso_role_name = SSOReadOnlyRole
```

3. Run the Amazon S3 application. A runtime exception appears.
4. Run the following AWS CLI command:

```
aws sso login --profile my-sso-profile
```

5. In the resulting web sign-in page, sign in. Use the user name from the invitation message and the password that was created in response to the message.
6. Run the Amazon S3 application again. The application now displays the list of S3 buckets.
7. Run the IAM application. The application displays the list of IAM users. This is true even though a second sign-in wasn't performed. The IAM application uses the temporary token that was created earlier.

Cleanup

If you don't want to keep the resources that you created during this tutorial, clean them up. These might be AWS resources or resources in your development environment such as files and folders.

Tutorial for AWS SSO using only .NET applications

This tutorial shows you how to enable AWS Single Sign-On for a basic application and a test SSO user. It configures the application to generate a temporary SSO token programmatically instead of using the AWS CLI (p. 63).

Before you start this tutorial, see the background information (p. 57) for using AWS SSO with the AWS SDK for .NET. Also see the high-level description for this scenario in the subsection called .NET application only (p. 60).

Several of the steps in this tutorial help you configure services like AWS Organizations and AWS SSO. If you've already performed that configuration, or if you're only interested in the code, you can skip to the section with the example code (p. 70).

Prerequisites

- Configure your development environment if you haven't already done so. This is described in sections like Install and configure your toolchain (p. 15) and Setting up your project (p. 17).
- Identify or create at least one AWS account that you can use to test AWS SSO. For the purposes of this tutorial, this is called the *test AWS account* or simply *test account*.
- Identify an *SSO user* who can test AWS SSO for you. This is a person who will be using SSO and the basic applications that you create. For this tutorial, that person might be you (the developer), or someone else. We also recommend a setup in which the SSO user is working on a computer that is not in your development environment. However, this isn't strictly necessary.
- The SSO user's computer must have a .NET framework installed that's compatible with the one you used to set up your development environment.

Set up AWS

This section shows you how to set up various AWS services for this tutorial.

To perform this setup, first sign in to the test AWS account as an administrator. Then, do the following:

Amazon S3

Go to the Amazon S3 console and add some innocuous buckets. Later in this tutorial, the SSO user will retrieve a list of these buckets.

AWS IAM

Go to the IAM console and add a few IAM users. If you give the IAM users permissions, limit the permissions to a few innocuous read-only permissions. Later in this tutorial, the SSO user will retrieve a list of these IAM users.

AWS Organizations

Go to the AWS Organizations console and enable Organizations. For more information, see Creating an organization in the AWS Organizations User Guide.

This action adds the test AWS account to the organization as the *management account*. If you have additional test accounts, you can invite them to join the organization, but doing so isn't necessary for this tutorial.

AWS SSO

Go to the AWS Single Sign-On console and enable AWS SSO. Perform email verification if necessary. For more information, see Enable AWS SSO in the AWS Single Sign-On User Guide.

Then, perform the following configuration.

Configure AWS SSO

1. Go to the **Settings** page. Look for the **"User portal URL"** and record the value for later use in the `sso_start_url` setting.
2. In the banner of the AWS Management Console, look for the AWS Region that was set when you enabled AWS SSO. This is the dropdown menu to the left of the AWS account ID. Record the Region code for later use in the `sso_region` setting. This code will be similar to `us-east-1`.

3. Create an SSO user as follows:
   a. Go to the **Users** page.
   b. Choose **Add user** and enter the user's **Username**, **Email address**, **First name**, and **Last name**. Then, choose **Next**.
   c. Choose **Next** on the page for groups, then review the information and choose **Add user**.
4. Create a group as follows:
   a. Go to the **Groups** page.
   b. Choose **Create group** and enter the group's **Group name** and **Description**.
   c. In the **Add users to group** section, select the test SSO user that you created earlier. Then, select **Create group.**
5. Create a permission set as follows:
   a. Go to the **Permission sets** page and choose **Create permission set**.
   b. Select **Create a custom permission set** and choose **Next: Details**.
   c. For this tutorial, enter `SSOReadOnlyRole` for the **Name** and enter a **Description**.
   d. Select **Create a custom permissions policy** and enter the following policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "s3:ListAllMyBuckets",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

   e. Choose **Next: Tags**, **Next: Review**, and **Create**.
   f. Record the name of the permission set for later use in the `sso_role_name` setting.
6. Go to the **AWS accounts** page and choose the AWS account that you added to the organization earlier.
7. In the **Overview** section of that page, find the **Account ID** and record it for later use in the `sso_account_id` setting.
8. Choose the **Users and groups** tab and then choose **Assign users or groups**.
9. On the **Assign users and groups** page, choose the **Groups** tab, select the group that you created earlier, and choose **Next**.
10. Select the permission set that you created earlier and choose **Next**, then choose **Submit**. The configuration takes a few moments.

Create example applications

Create the following applications. They will be run on the SSO user's computer.

List Amazon S3 buckets

Include NuGet packages `AWSSDK.SSO` and `AWSSDK.SSOOIDC` in addition to `AWSSDK.S3` and `AWSSDK.SecurityToken`.

```
using System;
using System.Threading.Tasks;
using System.Diagnostics;


// NuGet packages: AWSSDK.S3, AWSSDK.SecurityToken, AWSSDK.SSO, AWSSDK.SSOOIDC
using Amazon.Runtime;
using Amazon.Runtime.CredentialManagement;
using Amazon.S3;
using Amazon.S3.Model;
using Amazon.SecurityToken;
using Amazon.SecurityToken.Model;

namespace SSOExample.S3.Programmatic_login
{
    class Program
    {
        // Requirements:
        // - An SSO profile in the SSO user's shared config file.

        // Class members.
        private static string profile = "my-sso-profile";

        static async Task Main(string[] args)
        {
            // Get SSO credentials from the information in the shared config file.
            var ssoCreds = LoadSsoCredentials(profile);

            // Display the caller's identity.
            var ssoProfileClient = new AmazonSecurityTokenServiceClient(ssoCreds);
            Console.WriteLine($"\nSSO Profile:\n {await
 ssoProfileClient.GetCallerIdentityArn()}");

            // Display a list of the account's S3 buckets.
            // The S3 client is created using the SSO credentials obtained earlier.
            var s3Client = new AmazonS3Client(ssoCreds);
            Console.WriteLine("\nGetting a list of your buckets...");
            var listResponse = await s3Client.ListBucketsAsync();
            Console.WriteLine($"Number of buckets: {listResponse.Buckets.Count}");
            foreach (S3Bucket b in listResponse.Buckets)
            {
                Console.WriteLine(b.BucketName);
            }
            Console.WriteLine();
        }

        // Method to get SSO credentials from the information in the shared config file.
        static AWSCredentials LoadSsoCredentials(string profile)
        {
            var chain = new CredentialProfileStoreChain();
            if (!chain.TryGetAWSCredentials(profile, out var credentials))
                throw new Exception($"Failed to find the {profile} profile");

            var ssoCredentials = credentials as SSOAWSCredentials;

            ssoCredentials.Options.ClientName = "Example-SSO-App";
            ssoCredentials.Options.SsoVerificationCallback = args =>
            {
                // Launch a browser window that prompts the SSO user to complete an SSO
 login.
                //  This method is only invoked if the session doesn't already have a valid
 SSO token.
                // NOTE: Process.Start might not support launching a browser on macOS or
 Linux. If not,
                //       use an appropriate mechanism on those systems instead.
```

```
                Process.Start(new ProcessStartInfo
                {
                    FileName = args.VerificationUriComplete,
                    UseShellExecute = true
                });
            };

            return ssoCredentials;
        }

    }

    // Class to read the caller's identity.
    public static class Extensions
    {
        public static async Task<string> GetCallerIdentityArn(this
 IAmazonSecurityTokenService stsClient)
        {
            var response = await stsClient.GetCallerIdentityAsync(new
 GetCallerIdentityRequest());
            return response.Arn;
        }
    }
}
```

List IAM users

Include NuGet packages `AWSSDK.SSO` and `AWSSDK.SSOOIDC` in addition to `AWSSDK.IdentityManagement` and `AWSSDK.SecurityToken`.

```
using System;
using System.Threading.Tasks;
using System.Diagnostics;


// NuGet packages: AWSSDK.IdentityManagement, AWSSDK.SecurityToken, AWSSDK.SSO,
 AWSSDK.SSOOIDC
using Amazon.Runtime;
using Amazon.Runtime.CredentialManagement;
using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;
using Amazon.SecurityToken;
using Amazon.SecurityToken.Model;

namespace SSOExample.IAM.Programmatic_login
{
    class Program
    {
        // Requirements:
        // - An SSO profile in the SSO user's shared config file.

        // Class members.
        private static string profile = "my-sso-profile";

        static async Task Main(string[] args)
        {
            // Get SSO credentials from the information in the shared config file.
            var ssoCreds = LoadSsoCredentials(profile);

            // Display the caller's identity.
            var ssoProfileClient = new AmazonSecurityTokenServiceClient(ssoCreds);
            Console.WriteLine($"\nSSO Profile:\n {await
 ssoProfileClient.GetCallerIdentityArn()}");
```

```
            // Display a list of the account's IAM users.
            // The IAM client is created using the SSO credentials obtained earlier.
            var iamClient = new AmazonIdentityManagementServiceClient(ssoCreds);
            Console.WriteLine("\nGetting a list of IAM users...");
            var listResponse = await iamClient.ListUsersAsync();
            Console.WriteLine($"Number of IAM users: {listResponse.Users.Count}");
            foreach (User u in listResponse.Users)
            {
                Console.WriteLine(u.UserName);
            }
            Console.WriteLine();
        }

        // Method to get SSO credentials from the information in the shared config file.
        static AWSCredentials LoadSsoCredentials(string profile)
        {
            var chain = new CredentialProfileStoreChain();
            if (!chain.TryGetAWSCredentials(profile, out var credentials))
                throw new Exception($"Failed to find the {profile} profile");

            var ssoCredentials = credentials as SSOAWSCredentials;

            ssoCredentials.Options.ClientName = "Example-SSO-App";
            ssoCredentials.Options.SsoVerificationCallback = args =>
            {
                // Launch a browser window that prompts the SSO user to complete an SSO
 login.
                //  This method is only invoked if the session doesn't already have a valid
 SSO token.
                // NOTE: Process.Start might not support launching a browser on macOS or
 Linux. If not,
                //       use an appropriate mechanism on those systems instead.
                Process.Start(new ProcessStartInfo
                {
                    FileName = args.VerificationUriComplete,
                    UseShellExecute = true
                });
            };

            return ssoCredentials;
        }

    }

    // Class to read the caller's identity.
    public static class Extensions
    {
        public static async Task<string> GetCallerIdentityArn(this
 IAmazonSecurityTokenService stsClient)
        {
            var response = await stsClient.GetCallerIdentityAsync(new
 GetCallerIdentityRequest());
            return response.Arn;
        }
    }
}
```

In addition to displaying lists of Amazon S3 buckets and IAM users, these applications display the user identity ARN for the SSO-enabled profile, which is `my-sso-profile` in this tutorial.

These applications perform SSO sign-in tasks by providing a callback method in the Options property of an SSOAWSCredentials object.

Instruct SSO user

Ask the SSO user to check their email and accept the SSO invitation. They are prompted to set a password. The message might take a few minutes to arrive in the SSO user's inbox.

Give the SSO user the applications that you created earlier.

Then, have the SSO user do the following:

1. If the folder that contains the shared AWS `config` file doesn't exist, create it. If the folder does exist and has a subfolder called `.sso`, delete that subfolder.

   The location of this folder is typically `%USERPROFILE%\.aws` in Windows and `~/.aws` in Linux and macOS.
2. Create a shared AWS `config` file in that folder, if necessary, and add a profile to it as follows:

```
[default]
region = <default Region>

[profile my-sso-profile]
sso_start_url = <user portal URL recorded earlier>
sso_region = <Region code recorded earlier>
sso_account_id = <account ID recorded earlier>
sso_role_name = SSOReadOnlyRole
```

3. Run the Amazon S3 application.
4. In the resulting web sign-in page, sign in. Use the user name from the invitation message and the password that was created in response to the message.
5. When sign-in is complete, the application displays the list of S3 buckets.
6. Run the IAM application. The application displays the list of IAM users. This is true even though a second sign-in wasn't performed. The IAM application uses the temporary token that was created earlier.

Cleanup

If you don't want to keep the resources that you created during this tutorial, clean them up. These might be AWS resources or resources in your development environment such as files and folders.

Deploying applications with the AWS SDK for .NET

This section provides information about various ways in which you can deploy and maintain applications that have been created with the AWS SDK for .NET.

You use these tools along with setting up your environment (p. 15), setting up your project (p. 17), and creating applications to work with AWS services, examples of which are given in Code examples with guidance (p. 97).

Topics

AWS .NET deployment tool for the .NET CLI

This is prerelease documentation for a feature in preview release. It is subject to change.

After you've developed your cloud-native .NET Core application on a development machine, you'll most likely want to deploy it to AWS.

Deployment to AWS sometimes involves multiple AWS services and resources, each of which must be configured. To ease this deployment work, you can use the AWS .NET deployment tool for the .NET CLI, or *deployment tool* for short.

> **Note**
> If you develop on Windows with Visual Studio, you might have the AWS Toolkit for Visual Studio installed. The toolkit provides similar deployment functionality in its **Publish to AWS** feature. For information about toolkit versions and using the feature, see Publish to AWS in the AWS Toolkit for Visual Studio User Guide.

When you run the deployment tool for a .NET Core application, the tool shows you all of the AWS compute-service options that are available to deploy your application. It suggests the most likely choice, as well as the most likely settings to go along with that choice. It then builds and packages your application as required by the chosen compute service. It generates the deployment infrastructure, deploys your application by using the AWS Cloud Development Kit (CDK), and then displays the endpoint.

You can select deployment options interactively or specify them in a JSON configuration file (p. 84). You can also keep the default values that the tool selects for you.

Capabilities

- Deploys to AWS Elastic Beanstalk or Amazon ECS (using AWS Fargate).
- Deploys cloud-native .NET applications that are built with .NET Core 2.1 and later, and that are written with the intent to deploy to Linux. Such an application isn't tied to any Windows-specific technology such as the Windows Registry, IIS, or MSMQ, and can be deployed on virtualized compute.
- Deploys ASP.NET Core web apps, Blazor WebAssembly apps, long-running service apps, and scheduled tasks. For more information, see the README in the GitHub repo.

Additional information

- The aws-dotnet-deploy GitHub repo.
- Blog post Reimagining the AWS .NET deployment experience.
- Blog post Update on our new AWS .NET Deployment Experience.
- Blog post Deployment Projects with the new AWS .NET Deployment Experience.

Topics

Setting up your environment

This is prerelease documentation for a feature in preview release. It is subject to change.

The following sections show you how to set up your environment to run the deployment tool.

Node.js

The deployment tool requires the AWS Cloud Development Kit (CDK), and the AWS CDK requires Node.js version 10.13.0 or later (excluding versions 13.0.0 through 13.6.0). To see which version of Node.js you have installed, run the following command at the command prompt or in a terminal:

```
node --version
```

> **Note**
> If the AWS CDK isn't installed on your machine or if the AWS CDK that's installed is earlier than the required minimum version (1.95.2), the deployment tool will install a temporary and "private" copy of the CDK that will be used only by the tool, leaving the global configuration of your machine untouched.
> If instead you want to install the AWS CDK, see Install the AWS CDK in the AWS Cloud Development Kit (CDK) Developer Guide

.NET Core and .NET

Your application must be built from .NET Core 3.1 or later (for example, .NET Core 3.1, .NET 5.0, etc.). To see what version you have, run the following on the command prompt or in a terminal:

```
dotnet --version
```

For information about how to install or update .NET, see https://dotnet.microsoft.com/.

(Optional) Docker

If you plan to deploy your application to Amazon Elastic Container Service (Amazon ECS) using AWS Fargate, you must have Docker installed where you run the deployment tool. For more information, see https://docs.docker.com/engine/install/.

(Linux and macOS) ZIP CLI

The ZIP CLI is used when creating ZIP packages for deployment bundles. It is used to maintain Linux file permissions.

Setting up the deployment tool

This is prerelease documentation for a feature in preview release. It is subject to change.

The following instructions show you how to install, update, and uninstall the deployment tool.

To install the deployment tool

1. Open a command prompt or terminal.
2. Install the tool: `dotnet tool install --global aws.deploy.tools`
3. Verify the installation by checking the version: `dotnet aws --version`

To update the deployment tool

1. Open a command prompt or terminal.
2. Check the version: `dotnet aws --version`
3. (Optional) Check to see if a later version of the tool is available on the NuGet page for the deployment tool.
4. Update the tool: `dotnet tool update -g aws.deploy.tools`
5. Verify the installation by checking the version again: `dotnet aws --version`

To remove the deployment tool from your system

1. Open a command prompt or terminal.
2. Uninstall the tool: `dotnet tool uninstall -g aws.deploy.tools`
3. Verify that the tool is no longer installed: `dotnet aws --version`

Setting up credentials for the deployment tool

This is prerelease documentation for a feature in preview release. It is subject to change.

This information is about how to set up credentials for the deployment tool. If you're looking for information about setting up credentials for your project, see Configure AWS credentials (p. 19) instead.

To run the deployment tool against your AWS account, you must have a credential profile. The profile must be set up with at least an access key ID and a secret access key for an AWS Identity and Access Management (IAM) user. For information about how to do this, see Create users and roles (p. 18) and Using the shared AWS credentials file (p. 20).

The credentials that you use to run the deployment tool must have permissions for certain services, depending on the tasks that you're trying to perform. The following are some examples of the typical permissions that are required to run the tool. Additional permissions might be required, depending on the type of application you're deploying and the services it uses.

Task	Permissions for services
Display a list of AWS CloudFormation stacks (list-deployments)	CloudFormation
Deploy and redeploy to Elastic Beanstalk (deploy)	CloudFormation, Elastic Beanstalk
Deploy and redeploy to Amazon ECS (deploy)	CloudFormation, Elastic Beanstalk, Elastic Container Registry

The deployment tool automatically uses the `[default]` profile from your shared AWS config and credentials files (p. 20) if that profile exists. You can change this behavior by specifying a profile for the tool to use, either system-wide or in a particular context.

To specify a system-wide profile, do the following:

- Specify the `AWS_PROFILE` environment variable globally, as appropriate for your operating system. Be sure to reopen command prompts or terminals as necessary. If the profile you specify doesn't include an AWS Region, the tool might ask you to choose one.

  Warning
  If you set the `AWS_PROFILE` environment variable globally for your system, other SDKs, CLIs, and tools will also use that profile. If this behavior is unacceptable, specify a profile for a particular context instead.

To specify a profile for a particular context, do one of the following:

- Specify the `AWS_PROFILE` environment variable in the command prompt or terminal session from which you're running the tool (as appropriate for your operating system).
- Specify the `--profile` and `--region` switches within the command. For example: `dotnet aws list-stacks --region us-west-2`. For additional information about the deployment tool's commands, see Running the deployment tool (p. 78).
- Specify nothing and the tool will ask you to choose a profile and an AWS Region.

Running the deployment tool

This is prerelease documentation for a feature in preview release. It is subject to change.

To help you become familiar with the deployment tool, this topic contains a quick tour of the tool's command line.

Command line help

For basic help, run the following:

```
dotnet aws --help
```

The available commands are listed in the **Commands** section of the help text; for example, `deploy` and `list-deployments`.

To get help on a specific command, use `--help` in the context of that command. For example:

```
dotnet aws deploy --help
```

Some common commands

To deploy or redeploy an application:

```
cd <dotnet_core_app_directory>
dotnet aws deploy [--profile <profile_name>] [--region <region_code>] [--apply
 <configuration_file>] [-s|--silent]
```

To show a list of the CloudFormation stacks that you've created by using the tool:

```
dotnet aws list-deployments [—region <region_code>] [--profile <profile_name>]
```

Tutorials for deploying your application

This is prerelease documentation for a feature in preview release. It is subject to change.

After you develop your application, you deploy it to AWS. When you use the deployment tool, the tool shows you all of the AWS compute-service options that are available to deploy your application. It then suggests the most likely choice, as well as the most likely settings to go along with that choice.

The following sections show some examples of deployments to AWS. The examples use rudimentary applications, the `[default]` credentials profile, and the default settings that are provided by the deployment tool.

For an example that demonstrates deployment projects, see the blog post Deployment Projects with the new AWS .NET Deployment Experience.

Tutorials

- Deploying a webapp to Elastic Beanstalk (p. 79)
- Deploying a webapp to Amazon ECS (p. 81)

Deploying a webapp to Elastic Beanstalk

This is prerelease documentation for a feature in preview release. It is subject to change.

This tutorial shows a deployment to Elastic Beanstalk. The tutorial uses the defaults that are provided by the deployment tool.

Prerequisites

- You have completed environment setup (p. 76) and tool setup (p. 77).
- The `[default]` credentials profile has the required permissions.

Deploy

When you're ready to deploy your application to AWS for the first time, this is where you start.

To deploy the example web app to Elastic Beanstalk

1. Go to a directory where you want to work and create the basic web app:

   ```
   dotnet new webapp --name SimpleWebAppForBeanstalk
   ```
2. Go to the application directory:

   ```
   cd SimpleWebAppForBeanstalk
   ```
3. Run the deployment tool:

   ```
   dotnet aws deploy
   ```
4. In **Name the AWS stack to deploy your application to**, press the **Enter** key to accept the default name. If there are existing stacks, this shows **Select the AWS stack to deploy your application to** instead. In this case, choose the last option to **Create new**, then press the **Enter** key to accept the default name.
5. For the next inquiry, **Choose deployment option**, choose the option for **AWS Elastic Beanstalk on Linux** and press the **Enter** key. For this tutorial, this is the default option.

   Note
   When deploying a real application, if the tool doesn't find a Dockerfile for the project, the deployment tool displays **AWS Elastic Beanstalk on Linux** as the default option. If there is a Dockerfile for the project, the tool displays a different default. For more information about this alternative scenario, see Deploying a webapp to Amazon ECS (p. 81).
6. Press the **Enter** key again to accept the defaults for application and stack settings and start the deployment.
7. Wait for the deployment process to finish.
8. At the end of the tool's output, you see the following line: "SimpleWebAppForBeanstalk.EndpointURL...". This line contains the URL for the resulting web site. You can open this URL in a web browser. Alternatively, you can open the resulting web site from the Elastic Beanstalk console, as shown next.
9. Sign in to the AWS Management Console and open the Elastic Beanstalk console at https://console.aws.amazon.com/elasticbeanstalk/.

   Select the appropriate AWS Region, if necessary.
10. On the **Environments** page, choose the **SimpleWebAppForBeanstalk-dev** environment.
11. In the top section of the environment's page, verify that the **Health** status is **Ok**, and then open the link to see the resulting web site. Leave this web site open for now.

Update and redeploy

Now that you have deployed an application and can see the resulting web site, it's time make some changes to the content and redeploy the application.

To make changes to the web content and redeploy the application

1. In the `Pages` sub-directory of the tutorial project, open `Index.cshtml` in a text editor.
2. Make some changes to the HTML content and save the file.
3. In the main directory for the project, run the deployment tool again:

   ```
   dotnet aws deploy
   ```

4. In **Select the AWS stack to deploy your application to**, choose the stack name that corresponds to this tutorial and press the **Enter** key. For this tutorial, this is **SimpleWebAppForBeanstalk**, and it's the default choice.
5. Press the **Enter** key again to accept the same defaults as before and wait for the application to redeploy.
6. In the Elastic Beanstalk console, look at the **SimpleWebAppForBeanstalk-dev** environment again. Verify that the **Health** status is **Ok**, and then refresh the application's web site to see your changes.

Cleanup

To avoid unexpected costs, be sure to remove the tutorial's environment and application when you're finished with them.

You can also do this cleanup manually by using the Elastic Beanstalk console at https://console.aws.amazon.com/elasticbeanstalk.

To remove tutorial artifacts

1. Get a list of the existing cloud applications:

   ```
   dotnet aws list-deployments.
   ```

   The list includes the deployment for this tutorial: **SimpleWebAppForBeanstalk**.
2. Delete the deployment:

   ```
   dotnet aws delete-deployment SimpleWebAppForBeanstalk
   ```
3. Enter "y" to confirm deletion and wait for the deployment to be deleted.
4. In the Elastic Beanstalk console, look at the **Environments** and **Applications** pages to verify that the tutorial deployment has been deleted.
5. Refresh the web site that had been created during the tutorial to verify that it's no longer available.

Deploying a webapp to Amazon ECS

This is prerelease documentation for a feature in preview release. It is subject to change.

This tutorial shows a deployment to Amazon ECS using AWS Fargate. The tutorial uses the defaults that are provided by the deployment tool.

Prerequisites

- You have completed environment setup (p. 76) and tool setup (p. 77).
- The `[default]` credentials profile has the required permissions.
- Docker is installed and running, but no Dockerfile exists for the tutorial.

Deploy

When you're ready to deploy your application to AWS for the first time, this is where you start.

To deploy the example web app to Amazon ECS using AWS Fargate

1. Go to a directory where you want to work and create the basic web app:

   ```
   dotnet new webapp --name SimpleWebAppForECS
   ```

2. Go to the application directory:

   ```
   cd SimpleWebAppForECS
   ```

3. Run the deployment tool:

   ```
   dotnet aws deploy
   ```

4. In **Name the AWS stack to deploy your application to**, press the **Enter** key to accept the default name. If there are existing stacks, this shows **Select the AWS stack to deploy your application to** instead. In this case, choose the last option to **Create new**, then press the **Enter** key to accept the default name.
5. For the next inquiry, **Choose deployment option**, choose the option for **Amazon ECS using Fargate** and press the **Enter** key. For this tutorial, this is the second option (under **Additional Deployment Options**), not the default.

   Note
   When deploying a real application, if the deployment tool finds a Dockerfile for the project, it displays **Amazon ECS using Fargate** as the default option. If there is no Dockerfile for the project, but you choose the **Amazon ECS using Fargate** option, the tool generates a Dockerfile.

6. Press the **Enter** key again to accept the defaults for application and stack settings and start the deployment. For this tutorial, since no Dockerfile was found for the project but the **Amazon ECS using Fargate** option was chosen, the tool also generates a Dockerfile.
7. Wait for the deployment process to finish.
8. At the end of the tool's output, you see the following line: "SimpleWebAppForECS.FargateServiceServiceURL...". This line contains the URL for the resulting web site. Open the URL in a web browser to see resulting web site. Leave this web site open for now.
9. If you want to see the resources that were created by the tool, open the Amazon ECS console at https://console.aws.amazon.com/ecs/. Select the appropriate AWS Region, if necessary. On the **Clusters** page you can see the new cluster that was created: **SimpleWebAppForECS**.

Update and redeploy

Now that you have deployed an application and can see the resulting web site, it's time make some changes to the content and redeploy the application.

To make changes to the web content and redeploy the application

1. In the `Pages` sub-directory of the tutorial project, open `Index.cshtml` in a text editor.
2. Make some changes to the HTML content and save the file.
3. In the main directory for the project, run the deployment tool again:

   ```
   dotnet aws deploy
   ```

4. In **Select the AWS stack to deploy your application to**, choose the stack name that corresponds to this tutorial and press the **Enter** key. For this tutorial, this is **SimpleWebAppForECS**, and it's the default choice.
5. Press the **Enter** key again to accept the same defaults as before and wait for the application to redeploy.
6. Refresh the application's web site to see your changes.

Cleanup

To avoid unexpected costs, be sure to remove the tutorial's clusters, tasks, and ECR repositories when you're finished with them.

You can also do this cleanup manually by using the Amazon ECS console at https://console.aws.amazon.com/ecs/.

To remove tutorial artifacts

1. Get a list of the existing cloud applications:

   ```
   dotnet aws list-deployments
   ```

   The list includes the deployment for this tutorial: **SimpleWebAppForECS**.
2. Delete the deployment:

   ```
   dotnet aws delete-deployment SimpleWebAppForECS
   ```
3. Enter "y" to confirm deletion and wait for the deployment to be deleted.
4. In the Amazon ECS console, look at the **Clusters** and **Task Definitions** pages to verify that the tutorial deployment has been deleted.
5. (Optional) On the **Amazon ECR**, **Repositories** page, you can delete the repository that was created for the tutorial: `simplewebappforecs`
6. Refresh the web site that had been created during the tutorial to verify that it's no longer available.

Redeploying your application after changes

This is prerelease documentation for a feature in preview release. It is subject to change.

When you make changes to your application, you can use the deployment tool to redeploy it. To do so, you go to the application's project directory and run `dotnet aws deploy` again.

At a certain point in the redeployment, the tool displays **Select the AWS stack to deploy your application to**. Scan the list of deployment stacks and do one of the following:

- Choose the stack name that corresponds to your .NET application.

  If you do this, the settings from the last deployment of the application are shown. Only a few of the advanced settings can be changed, the rest are read-only.
- Choose the last option, **Create new**, and then enter a new name.

  If you do this, the tool will create a new deployment stack and you can decide what you want to do with the settings: accept the defaults or make changes.

Deployment settings available through the deployment tool

This is prerelease documentation for a feature in preview release. It is subject to change.

One of the steps during deployment or redeployment is to choose the compute service you want to use, which is often the deployment tool's default suggestion. After you have chosen the deployment option you want, the tool shows you a list of the common settings that are available for that option. Advanced settings are also available, which you can see by entering "`more`" at the command prompt.

Many of the settings are self-explanatory. The following settings are less obvious.

Deployment to AWS Elastic Beanstalk

- Application IAM Role

  The **Application IAM Role** provides AWS credentials for your application, which has been deployed to AWS on your behalf. With an IAM role, the application can access AWS resources like Amazon S3 buckets.

  Using this setting, you can choose from existing IAM roles or allow the tool to create a role specifically for your application, which is the default behavior. You can see the definitions of the existing IAM roles by opening the IAM console at https://console.aws.amazon.com/iam/ and choosing the **Roles** page.
- EC2 Instance Type

  Using this setting, you can specify an Amazon EC2 instance type other than the one that the tool suggests. Find the available instance types by opening the Amazon EC2 console at https://console.aws.amazon.com/ec2/ and choosing the **Instance Types** page under **Instances**. For detailed descriptions of EC2 instance types, see https://aws.amazon.com/ec2/instance-types/.
- Key Pair

  The default for this setting is "EMPTY" (no key pair), meaning that you won't be able to SSH into the EC2 instance. If you want to SSH into the EC2 instance you can choose a key pair from the list. You can see the key pairs in the list by opening the EC2 console and choosing the **Key Pairs** page under **Network & Security**. You can also have the tool create a new key pair. If you choose this option, enter a name for the key pair and (when asked) a directory where the private key will be stored.

  > **Warning**
  > If you chose to create a new key pair and store it on your computer, take appropriate precautions to protect the key pair.

Deployment to Amazon ECS using AWS Fargate

- Application IAM Role

  The **Application IAM Role** provides AWS credentials for your application, which has been deployed to AWS on your behalf. With an IAM role, the application can access AWS resources like Amazon S3 buckets.

  Using this setting, you can choose from existing IAM roles or allow the tool to create a role specifically for your application, which is the default behavior. You can see the definitions of the existing IAM roles by opening the IAM console at https://console.aws.amazon.com/iam/ and choosing the **Roles** page. If no IAM roles are listed, it means that no appropriate IAM role exists, so the only choice is for the tool to create one.
- Virtual Private Cloud

  The default value for this setting is the account Default VPC, which you can find by opening the VPC console at https://console.aws.amazon.com/vpc/ and choosing **Your VPCs**. The account Default VPC has **Yes** in the **Default VPC** column of the VPC table. You might have to scroll horizontally to see that column. You can choose another VPC, or you can have the tool create a new VPC for your application.

Working with configuration files

This is prerelease documentation for a feature in preview release. It is subject to change.

When you run the `deploy` command of the AWS .NET deployment tool, you can select deployment options in response to prompts from the tool.

Alternatively, you can provide a JSON configuration file by using the `--apply` option of the command. The tool reads deployment options from the JSON parameters that are specified in the file, and uses those options in its prompts. It uses default values for any parameters that aren't specified.

If you use a JSON configuration file in conjunction with the `-s` (`--silent`) option, you can specify deployment options without being prompted by the deployment tool at all.

This section defines the JSON definitions and syntax that you use to construct a configuration file.

Note
Support for JSON configuration files was added in version 0.11.16 of the deployment tool.

Common JSON parameters

The parameters you can include in a JSON configuration file depend on the type of deployment you're doing, as shown later in this topic. The following parameters are common to all deployment types.

AWSProfile

The name of the AWS profile to use if not using the `[default]` profile.

AWSRegion

The name of the AWS Region to use if not using the region from the `[default]` profile.

StackName

The name of the AWS CloudFormation stack to use for your application. It can be the name of an existing stack or the name of a new stack to create.

JSON syntax for Amazon ECS deployments

```
{
    "AWSProfile": "string",
    "AWSRegion": "string",
    "StackName": "string",
    "RecipeId": "AspNetAppEcsFargate" | "ConsoleAppEcsFargateService" |
 "ConsoleAppEcsFargateScheduleTask",
    "OptionSettingsConfig":
    {
        "ECSCluster":
        {
            "CreateNew": boolean,
            "NewClusterName": "string" | "ClusterArn": "string"
        },
        "ECSServiceName": "string",
        "DesiredCount": integer,
        "ApplicationIAMRole":
        {
            "CreateNew": boolean,
            "RoleArn": "string"
        },
        "Schedule": "string",
        "Vpc":
        {
            "IsDefault": boolean,
            "CreateNew": boolean | "VpcId": "string"
        },
        "AdditionalECSServiceSecurityGroups": "string",
```

```
        "ECSServiceSecurityGroups": "string",
        "TaskCpu": integer,
        "TaskMemory": integer,
        "DockerExecutionDirectory": "string"
    }
}
```

The following parameter definitions are specific to the JSON syntax for an Amazon ECS deployment. Also see the section called “Common JSON parameters” (p. 85).

RecipeId

A value that identifies the type of Amazon ECS deployment you want to perform: an ASP.NET Core web app, a long running service app, or a scheduled task.

OptionSettingsConfig

The following options are available to configure an Amazon ECS deployment.

ECSCluster

The ECS cluster to use for your deployment. It can be a new cluster (the default) or an existing one. If this parameter isn't present, a new cluster is created with the same name as your project. If you want to give the new cluster a name, provide the name in `NewClusterName`. If you're using an existing cluster, set `CreateNew` to `false` and include the cluster's ARN in `ClusterArn`.

ECSServiceName

This parameter is valid only for the `AspNetAppEcsFargate` and `ConsoleAppEcsFargateService` recipes.

The name of the ECS service running in the cluster. If this parameter isn't present, the service will be named "<YourProjectName>-service".

DesiredCount

This parameter is valid only for the `AspNetAppEcsFargate` and `ConsoleAppEcsFargateService` recipes.

The number of ECS tasks you want to run for the service. If given, the value must be at least 1 and at most 5000. The default is 3 for the `AspNetAppEcsFargate` recipe and 1 for the `ConsoleAppEcsFargateService` recipe.

ApplicationIAMRole

The IAM role that provides AWS credentials to the application to access AWS services. You can create a new role (the default) or use an existing role. To use an existing role, set `CreateNew` to `false` and include the role's ARN in `RoleArn`.

Schedule

This parameter is valid only for the `ConsoleAppEcsFargateScheduleTask` recipe.

The schedule or rate (frequency) that determines when Amazon CloudWatch Events runs the task. For details about the format of this value, see Schedule Expressions for Rules in the Amazon CloudWatch Events User Guide.

Vpc

The Amazon Virtual Private Cloud (VPC) in which to launch the application. It can be the Default VPC (the default behavior), a new VPC, or an existing VPC. To create a new VPC, set `IsDefault` to `false` and `CreateNew` to `true`. To use an existing VPC, set `IsDefault` to `false` and include the VPC ID in `VpcId`.

AdditionalECSServiceSecurityGroups

This parameter is valid only for the `AspNetAppEcsFargate` recipe.

A comma-delimited list of EC2 security groups to assign to the ECS service.

ECSServiceSecurityGroups

This parameter is valid only for the `ConsoleAppEcsFargateService` recipe.

A comma-delimited list of EC2 security groups to assign to the ECS service.

TaskCpu

The number of CPU units used by the task. Valid values are "256" (the default), "512", "1024", "2048", and "4096". For more information, see Amazon ECS on AWS Fargate in the Amazon Elastic Container Service Developer Guide, specifically, **Task CPU and memory**.

TaskMemory

The amount of memory (in MB) used by the task. Valid values are "512" (the default), "1024", "2048", "3072", "4096", "5120", "6144", "7168", "8192", "9216", "10240", "11264", "12288", "13312", "14336", "15360", "16384", "17408", "18432", "19456", "20480", "21504", "22528", "23552", "24576", "25600", "26624", "27648", "28672", "29696", and "30720". For more information, see Amazon ECS on AWS Fargate in the Amazon Elastic Container Service Developer Guide, specifically, **Task CPU and memory**.

DockerExecutionDirectory

If you're using Docker, the path to the Docker execution environment, formatted as a string that's properly escaped for your operating system (for example on Windows: "C:\\codebase").

JSON syntax for AWS Elastic Beanstalk deployments

```
{
    "AWSProfile": "string",
    "AWSRegion": "string",
    "StackName": "string",
    "RecipeId": "AspNetAppElasticBeanstalkLinux",
    "OptionSettingsConfig":
    {
        "BeanstalkApplication":
        {
            "CreateNew": boolean,
            "ApplicationName": "string"
        },
        "EnvironmentName": "string",
        "InstanceType": "string",
        "EnvironmentType": "SingleInstance" | "LoadBalanced",
        "LoadBalancerType": "application" | "classic" | "network",
        "ApplicationIAMRole":
        {
            "CreateNew": boolean,
            "RoleArn": "string"
        },
        "EC2KeyPair": "string",
        "ElasticBeanstalkPlatformArn": "string",
        "ElasticBeanstalkManagedPlatformUpdates":
        {
            "ManagedActionsEnabled": boolean,
            "PreferredStartTime": "Mon:12:00",
            "UpdateLevel": "minor"
        }
    }
```

```
}
```

The following parameter definitions are specific to the JSON syntax for an Elastic Beanstalk deployment. Also see the section called “Common JSON parameters” (p. 85).

RecipeId

A value that identifies the type of AWS Elastic Beanstalk deployment you want to perform; in this case, an ASP.NET Core web app.

OptionSettingsConfig

The following options are available to configure an Elastic Beanstalk deployment.

BeanstalkApplication

The name of the Elastic Beanstalk application. It can be a new application (the default) or an existing one. If you don't supply a name through `ApplicationName`, the application will have the same name as your project.

EnvironmentName

The name of the Elastic Beanstalk environment in which to run the application. If this parameter isn't present, the environment will be named "<YourProjectName>-dev".

InstanceType

The Amazon EC2 instance type of the EC2 instances created for the environment; for example, "t2.micro". If this parameter isn't included, an instance type is chosen based on the requirements of your project.

EnvironmentType

The type of Elastic Beanstalk environment to create: a single instance for development work (the default) or load balanced for production.

LoadBalancerType

The type of load balancer you want for your environment: classic, application (the default), or network.

This parameter is valid only if the value of `EnvironmentType` is "LoadBalanced".

ApplicationIAMRole

The IAM role that provides AWS credentials to the application to access AWS services. You can create a new role (the default) or use an existing role. To use an existing role, set `CreateNew` to `false` and include the role's ARN in `RoleArn`.

EC2KeyPair

The EC2 key pair that you can use to SSH into EC2 instances for the Elastic Beanstalk environment. If you don't include this parameter and you don't choose a key pair interactively when the deployment tool is running, you won't be able to SSH into the EC2 instance.

ElasticBeanstalkPlatformArn

The ARN of the AWS Elastic Beanstalk platform to use with the environment. If this parameter isn't present, the ARN of the latest Elastic Beanstalk platform is used.

For information about how to construct this ARN, see ARN format in the AWS Elastic Beanstalk Developer Guide.

ElasticBeanstalkManagedPlatformUpdates

Use this parameter to configure automatic updates for your Elastic Beanstalk platform using managed platform updates, as described in the AWS Elastic Beanstalk Developer Guide. If

`ManagedActionsEnabled` is set to `true` (the default), you can specify the weekly maintenance window through `PreferredStartTime`, which defaults to "Sun:00:00". Additionally, you can use `UpdateLevel` to specify the patch level to apply: "minor" (the default) or "patch". These options are described in Managed action option namespaces in the Elastic Beanstalk developer guide.

Your application continues to be available during the update process.

JSON syntax for Blazor WebAssembly app deployments

```
{
    "AWSProfile": "string",
    "AWSRegion": "string",
    "StackName": "string",
    "RecipeId": "BlazorWasm",
    "OptionSettingsConfig":
    {
        "IndexDocument": "string",
        "ErrorDocument": "string",
        "Redirect404ToRoot": boolean
    }
}
```

Note
This deployment task deploys a Blazor WebAssembly application to an Amazon S3 bucket. The bucket created during deployment is configured for web hosting and its contents are open to the public with read access.

The following parameter definitions are specific to the JSON syntax for a Blazor WebAssembly deployment. Also see the section called “Common JSON parameters” (p. 85).

RecipeId

A value that identifies the type of deployment you want to perform; in this case, a Blazor WebAssembly application.

OptionSettingsConfig

The following options are available to configure a Blazor WebAssembly deployment.

IndexDocument

The name of the web page to use when the endpoint for your WebAssembly app is accessed with no resource path. The default page name is `index.html`.

ErrorDocument

The name of the web page to use when an error occurs while accessing a resource path. The default value is an empty string.

Redirect404ToRoot

If this parameter is set to `true` (the default), requests that result in a 404 are redirected the index document of the web app, which is specified by `IndexDocument`.

Migrating your project for the AWS SDK for .NET

This section provides information about migration tasks that might apply to you, and instructions about how to perform those tasks.

Topics

What's new in the AWS SDK for .NET

See the product page at https://aws.amazon.com/sdk-for-net/ for high-level information about new developments related to the AWS SDK for .NET.

The following is what's new in the AWS SDK for .NET.

2021-03-17: Developer Preview of the AWS .NET deployment tool

This is prerelease documentation for a service in preview release. It is subject to change.

The AWS .NET deployment tool, which is an opinionated deployment tool for the .NET CLI, is available for Developer Preview. You can use the deployment tool to deploy cloud-native .NET applications from the .NET CLI in just a few steps.

2020-08-24: Version 3.5 of the SDK has been released

- Standardized the .NET experience by transitioning support for all non-Framework variations of the SDK to .NET Standard 2.0. See Migrating to version 3.5 (p. 93) for more information.
- Added paginators to many service clients, which make pagination of API results more convenient. For more information, see Paginators (p. 52).

Platforms supported by the AWS SDK for .NET

The AWS SDK for .NET provides distinct groups of assemblies for developers to target different platforms. However, not all SDK functionality is the same on each of these platforms. This topic describes the differences in support for each platform.

.NET Core

The AWS SDK for .NET supports applications written for .NET Core. AWS service clients only support asynchronous calling patterns in .NET core. This also affects many of the high level abstractions built on

top of service clients like Amazon S3's `TransferUtility` which will only support asynchronous calls in the .NET Core environment.

.NET Standard 2.0

Non-Framework variations of the AWS SDK for .NET comply with .NET Standard 2.0.

.NET Framework 4.5

This version of the AWS SDK for .NET is compiled against .NET Framework 4.5 and runs in the .NET 4.0 runtime. AWS service clients support synchronous and asynchronous calling patterns and use the async and await keywords introduced in C# 5.0.

.NET Framework 3.5

This version of the AWS SDK for .NET is compiled against .NET Framework 3.5, and runs in either the .NET 2.0 or .NET 4.0 runtime. AWS service clients support synchronous and asynchronous calling patterns and use the older Begin and End pattern.

> **Note**
> The AWS SDK for .NET is not Federal Information Processing Standard (FIPS) compliant when used by applications built against version 2.0 of the CLR. For details on how you can substitute a FIPS compliant implementation in that environment, refer to CryptoConfig on the Microsoft blog and the CLR Security team's HMACSHA256 class (HMACSHA256Cng) in Security.Cryptography.dll.

Portable Class Library and Xamarin

The AWS SDK for .NET also contains a Portable Class Library implementation. The Portable Class Library implementation can target multiple platforms, including Universal Windows Platform (UWP) and Xamarin on iOS and Android. See the Mobile SDK for .NET and Xamarin for more details. AWS service clients only support asynchronous calling patterns.

Unity support

For information about Unity support, see Special considerations for Unity support (p. 328).

More information

Migrating to version 3.5 of the AWS SDK for .NET (p. 93)

Migrating to Version 3 of the AWS SDK for .NET

This topic describes changes in version 3 of the AWS SDK for .NET and how to migrate your code to this version of the SDK.

About the AWS SDK for .NET Versions

The AWS SDK for .NET, originally released in November 2009, was designed for .NET Framework 2.0. Since that release, .NET has improved with .NET Framework 4.0 and .NET Framework 4.5, and added new target platforms: WinRT and Windows Phone.

AWS SDK for .NET version 2 was updated to take advantage of the new features of the .NET platform and to target WinRT and Windows Phone.

AWS SDK for .NET version 3 has been updated to make the assemblies modular.

Architecture Redesign for the SDK

The entire version 3 of the AWS SDK for .NET is redesigned to be modular. Each service is now implemented in its own assembly, instead of in one global assembly. You no longer have to add the entire AWS SDK for .NET to your application. You can now add assemblies only for the AWS services your application uses.

Breaking Changes

The following sections describe changes to version 3 of the AWS SDK for .NET.

AWSClientFactory Removed

The `Amazon.AWSClientFactory` class was removed. Now, to create a service client, use the constructor of the service client. For example, to create an `AmazonEC2Client`:

```
var ec2Client = new Amazon.EC2.AmazonEC2Client();
```

Amazon.Runtime.AssumeRoleAWSCredentials Removed

The `Amazon.Runtime.AssumeRoleAWSCredentials` class was removed because it was in a core namespace but had a dependency on the AWS Security Token Service, and because it has been obsolete in the SDK for some time. Use the `Amazon.SecurityToken.AssumeRoleAWSCredentials` class instead.

SetACL Method Removed from S3Link

The `S3Link` class is part of the `Amazon.DynamoDBv2` package and is used for storing objects in Amazon S3 that are references in a DynamoDB item. This is a useful feature, but we didn't want to create a compile dependency on the `Amazon.S3` package for DynamoDB. Consequently, we simplified the exposed `Amazon.S3` methods from the `S3Link` class, replacing the `SetACL` method with the `MakeS3ObjectPublic` method. For more control over the access control list (ACL) on the object, use the `Amazon.S3` package directly.

Removal of Obsolete Result Classes

For most services in the AWS SDK for .NET, operations return a response object that contains metadata for the operation, such as the request ID and a result object. Having a separate response and result class was redundant and created extra typing for developers. In version 2 of the AWS SDK for .NET, we put all the information in the result class into the response class. We also marked the result classes obsolete to discourage their use. In version 3 of the AWS SDK for .NET, we removed these obsolete result classes to help reduce the SDK's size.

AWS Config Section Changes

It is possible to do advanced configuration of the AWS SDK for .NET through the `App.config` or `Web.config` file. You do this through an `<aws>` config section like the following, which references the SDK assembly name.

```
<configuration>
  <configSections>
    <section name="aws" type="Amazon.AWSSection, AWSSDK"/>
  </configSections>
  <aws region="us-west-2">
    <logging logTo="Log4Net"/>
  </aws>
</configuration>
```

In version 3 of the AWS SDK for .NET, the `AWSSDK` assembly no longer exists. We put the common code into the `AWSSDK.Core` assembly. As a result, you will need to change the references to the `AWSSDK` assembly in your `App.config` or `Web.config` file to the `AWSSDK.Core` assembly, as follows.

```
<configuration>
  <configSections>
    <section name="aws" type="Amazon.AWSSection, AWSSDK.Core"/>
  </configSections>
  <aws region="us-west-2">
    <logging logTo="Log4Net"/>
  </aws>
</configuration>
```

You can also manipulate the config settings with the `Amazon.AWSConfigs` class. In version 3 of the AWS SDK for .NET, we moved the config settings for DynamoDB from the `Amazon.AWSConfigs` class to the `Amazon.AWSConfigsDynamoDB` class.

Migrating to version 3.5 of the AWS SDK for .NET

Version 3.5 of the AWS SDK for .NET further standardizes the .NET experience by transitioning support for all non-Framework variations of the SDK to .NET Standard 2.0. Depending on your environment and code base, to take advantage of version 3.5 features, you might need to perform certain migration work.

This topic describes the changes in version 3.5 and possible work that you might need to do to migrate your environment or code from version 3.

What's changed for version 3.5

The following describes what has or hasn't changed in the AWS SDK for .NET version 3.5.

.NET Framework and .NET Core

Support for .NET Framework and .NET Core has not changed.

Xamarin

Xamarin projects (new and existing) must target .NET Standard 2.0. See .NET Standard 2.0 Support in Xamarin.Forms and .NET implementation support.

Unity

Unity apps must target .NET Standard 2.0 or .NET 4.x profiles using Unity 2018.1 or later. For more information, see .NET profile support. In addition, if you're using **IL2CPP** to build, you must disable code stripping by adding a *link.xml* file, as described in Referencing the AWS SDK for .NET Standard 2.0 from Unity, Xamarin, or UWP. After you port your code to one of the recommended code bases, your Unity app can access all of the services offered by the SDK.

Because Unity supports .NET Standard 2.0, the **AWSSDK.Core** package of the SDK version 3.5 no longer has Unity-specific code, including some higher-level functionality. To provide a better transition, all of the ***legacy*** Unity code is available for reference in the aws/aws-sdk-unity-net GitHub repository. If you find missing functionality that impacts your use of AWS with Unity, you can file a feature request at https://github.com/aws/dotnet/issues.

Also see Special considerations for Unity support (p. 328).

Universal Windows Platform (UWP)

Target your UWP application to version 16299 or later (Fall Creators update, version 1709, released October 2017).

Windows Phone and Silverlight

Version 3.5 of the AWS SDK for .NET does not support these platforms because Microsoft is no longer actively developing them. For more information, see the following:

- Windows 10 Mobile end of support
- Silverlight end of support

Legacy portable class libraries (profile-based PCLs)

Consider retargeting your library to .NET Standard. For more information, see Comparison to Portable Class Libraries from Microsoft.

Amazon Cognito Sync Manager and Amazon Mobile Analytics Manager

High-level abstractions that ease the use of Amazon Cognito Sync and Amazon Mobile Analytics are removed from version 3.5 of the AWS SDK for .NET. AWS AppSync is the preferred replacement for Amazon Cognito Sync. Amazon Pinpoint is the preferred replacement for Amazon Mobile Analytics.

If your code is affected by the lack of higher-level library code for AWS AppSync and Amazon Pinpoint, you can record your interest in one or both of the following GitHub issues: https://github.com/aws/dotnet/issues/20 and https://github.com/aws/dotnet/issues/19. You can also obtain the libraries for Amazon Cognito Sync Manager and Amazon Mobile Analytics Manager from the following GitHub repositories: aws/amazon-cognito-sync-manager-net and aws/aws-mobile-analytics-manager-net.

Migrating synchronous code

Version 3.5 of the AWS SDK for .NET supports only asynchronous calls to AWS services. You must change synchronous code you want to run using version 3.5 so that it runs asynchronously.

The following code snippets show how you might change synchronous code into asynchronous code. The code in these snippets is used to display the number of Amazon S3 buckets.

The original code calls ListBuckets.

```
private static ListBucketsResponse MyListBuckets()
{
  var s3Client = new AmazonS3Client();
  var response = s3Client.ListBuckets();
  return response;
}
```

```
// From the calling function
ListBucketsResponse response = MyListBuckets();
Console.WriteLine($"Number of buckets: {response.Buckets.Count}");
```

To use version 3.5 of the SDK, call ListBucketsAsync instead.

```
private static async Task<ListBucketsResponse> MyListBuckets()
{
  var s3Client = new AmazonS3Client();
  var response = await s3Client.ListBucketsAsync();
  return response;
}


// From an **asynchronous** calling function
ListBucketsResponse response = await MyListBuckets();
Console.WriteLine($"Number of buckets: {response.Buckets.Count}");

// OR From a **synchronous** calling function
Task<ListBucketsResponse> response = MyListBuckets();
Console.WriteLine($"Number of buckets: {response.Result.Buckets.Count}");
```

Migrating to version 3.7 of the AWS SDK for .NET

As of version 3.7, the AWS SDK for .NET no longer supports .NET Standard 1.3.

For information about migrating from .NET Standard 1.3, see Migrating from .NET Standard 1.3 (p. 95).

Migrating from .NET Standard 1.3

On June 27 2019 Microsoft ended support for .NET Core 1.0 and .NET Core 1.1 versions. Following this announcement, AWS ended support for .NET Standard 1.3 on the AWS SDK for .NET on December 31, 2020.

AWS continued to provide service updates and security fixes on the AWS SDK for .NET targeting .NET Standard 1.3 until October 1, 2020. After that date, the .NET Standard 1.3 target went into Maintenance mode, which meant that no new updates were released; AWS applied critical bug fixes and security patches only.

On December 31, 2020, support for .NET Standard 1.3 on the AWS SDK for .NET came to its end of life. After that date no bug fixes or security patches were applied. Artifacts built with that target remain available for download on NuGet.

What you need to do

- If you're running applications using .NET Framework, you're not affected.
- If you're running applications using .NET Core 2.0 or higher, you're not affected.
- If you're running applications using .NET Core 1.0 or .NET Core 1.1, migrate your applications to a newer version of .NET Core by following Microsoft migration instructions. We recommend a minimum of .NET Core 3.1.

- If you're running business critical applications that cannot be upgraded at this time, you can continue using your current version of AWS SDK for .NET.

If you have questions or concerns, contact AWS Support.

Working with AWS services in the AWS SDK for .NET

The following sections contain examples, tutorials, tasks, and guides that show you how to use the AWS SDK for .NET to work with AWS services.

If you're new to the AWS SDK for .NET, you might want to check out the Quick tour (p. 5) topic first. It gives you an introduction to the SDK.

You can find more code examples in the AWS Code Examples Repository and the awslabs repository on GitHub.

Before you begin, be sure you have set up your environment (p. 15). Also review the information in Setting up your project (p. 17) and SDK features (p. 49).

Topics

Code examples with guidance for the AWS SDK for .NET

The following sections contain code examples and provide guidance for the examples. They can help you learn how to use the AWS SDK for .NET to work with AWS services.

If you're new to the AWS SDK for .NET, you might want to check out the Quick tour (p. 5) topic first. It gives you an introduction to the SDK.

Before you begin, be sure you have set up your environment (p. 15). Also review the information in Setting up your project (p. 17) and SDK features (p. 49).

Topics

Accessing AWS CloudFormation with the AWS SDK for .NET

The AWS SDK for .NET supports AWS CloudFormation, which creates and provisions AWS infrastructure deployments predictably and repeatedly.

APIs

The AWS SDK for .NET provides APIs for AWS CloudFormation clients. The APIs enable you to work with AWS CloudFormation features such as templates and stacks. This section contains a small number of examples that show you the patterns you can follow when working with these APIs. To view the full set of APIs, see the AWS SDK for .NET API Reference (and scroll to "Amazon.CloudFormation").

The AWS CloudFormation APIs are provided by the AWSSDK.CloudFormation package.

Prerequisites

Before you begin, be sure you have set up your environment (p. 15). Also review the information in Setting up your project (p. 17) and SDK features (p. 49).

Topics

Topics

Listing AWS resources using AWS CloudFormation

This example shows you how to use the AWS SDK for .NET to list the resources in AWS CloudFormation stacks. The example uses the low-level API. The application takes no arguments, but simply gathers information for all stacks that are accessible to the user's credentials and then displays information about those stacks.

SDK references

NuGet packages:

- AWSSDK.CloudFormation

Programming elements:

- Namespace Amazon.CloudFormation

  Class AmazonCloudFormationClient
- Namespace Amazon.CloudFormation.Model

  Class DescribeStackResourcesRequest

  Class DescribeStackResourcesResponse

  Class DescribeStacksResponse

  Class Stack

  Class StackResource

  Class Tag

```
using System;
using System.Threading.Tasks;
using Amazon.CloudFormation;
using Amazon.CloudFormation.Model;

namespace CFNListResources
{
  class Program
  {
    static async Task Main(string[] args)
    {
      // Create the CloudFormation client
      var cfnClient = new AmazonCloudFormationClient();

      // List the resources for each stack
      await ListResources(cfnClient, await cfnClient.DescribeStacksAsync());
    }

    //
    // Method to list stack resources and other information
    private static async Task ListResources(
      IAmazonCloudFormation cfnClient, DescribeStacksResponse responseDescribeStacks)
    {
      Console.WriteLine("Getting CloudFormation stack information...");

      foreach (Stack stack in responseDescribeStacks.Stacks)
      {
        // Basic information for each stack
        Console.WriteLine("\n----------------------------------------------");
        Console.WriteLine($"\nStack: {stack.StackName}");
        Console.WriteLine($"  Status: {stack.StackStatus.Value}");
        Console.WriteLine($"  Created: {stack.CreationTime}");

        // The tags of each stack (etc.)
        if(stack.Tags.Count > 0)
        {
          Console.WriteLine("  Tags:");
          foreach (Tag tag in stack.Tags)
            Console.WriteLine($"    {tag.Key}, {tag.Value}");
        }

        // The resources of each stack
        DescribeStackResourcesResponse responseDescribeResources =
          await cfnClient.DescribeStackResourcesAsync(new DescribeStackResourcesRequest{
            StackName = stack.StackName});
        if(responseDescribeResources.StackResources.Count > 0)
        {
          Console.WriteLine("  Resources:");
          foreach(StackResource resource in responseDescribeResources.StackResources)
            Console.WriteLine($"    {resource.LogicalResourceId}:
 {resource.ResourceStatus}");
        }
      }
      Console.WriteLine("\n----------------------------------------------");
    }
  }
}
```

Authenticating users with Amazon Cognito

Note
The information in this topic is specific to projects based on .NET Framework and the AWS SDK for .NET version 3.3 and earlier.

Using Amazon Cognito Identity, you can create unique identities for your users and authenticate them for secure access to your AWS resources such as Amazon S3 or Amazon DynamoDB. Amazon Cognito Identity supports public identity providers such as Amazon, Facebook, Twitter/Digits, Google, or any OpenID Connect-compatible provider as well as unauthenticated identities. Amazon Cognito also supports developer authenticated identities, which let you register and authenticate users using your own backend authentication process, while still using Amazon Cognito Sync to synchronize user data and access AWS resources.

For more information on Amazon Cognito, see the Amazon Cognito Developer Guide.

The following code examples show how to easily use Amazon Cognito Identity. The Credentials provider (p. 100) example shows how to create and authenticate user identities. The CognitoAuthentication extension library (p. 102) example shows how to use the CognitoAuthentication extension library to authenticate Amazon Cognito user pools.

Topics

Amazon Cognito credentials provider

> **Note**
> The information in this topic is specific to projects based on .NET Framework and the AWS SDK for .NET version 3.3 and earlier.

`Amazon.CognitoIdentity.CognitoAWSCredentials`, found in the AWSSDK.CognitoIdentity NuGet package, is a credentials object that uses Amazon Cognito and the AWS Security Token Service (AWS STS) to retrieve credentials to make AWS calls.

The first step in setting up `CognitoAWSCredentials` is to create an “identity pool”. (An identity pool is a store of user identity information that is specific to your account. The information is retrievable across client platforms, devices, and operating systems, so that if a user starts using the app on a phone and later switches to a tablet, the persisted app information is still available for that user. You can create a new identity pool from the Amazon Cognito console. If you are using the console, it will also provide you the other pieces of information you need:

- Your account number- A 12-digit number, such as 123456789012, that is unique to your account.
- The unauthenticated role ARN- A role that unauthenticated users will assume. For example, this role can provide read-only permissions to your data.
- The authenticated role ARN- A role that authenticated users will assume. This role can provide more extensive permissions to your data.

Set up CognitoAWSCredentials

The following code example shows how to set up `CognitoAWSCredentials`, which you can then use to make a call to Amazon S3 as an unauthenticated user. This enables you to make calls with just a minimum amount of data required to authenticate the user. User permissions are controlled by the role, so you can configure access as you need.

```
CognitoAWSCredentials credentials = new CognitoAWSCredentials(
    accountId,        // Account number
    identityPoolId,   // Identity pool ID
    unAuthRoleArn,    // Role for unauthenticated users
    null,             // Role for authenticated users, not set
    region);
using (var s3Client = new AmazonS3Client(credentials))
```

```
{
    s3Client.ListBuckets();
}
```

Use AWS as an unauthenticated user

The following code example shows how you can start using AWS as an unauthenticated user, then authenticate through Facebook and update the credentials to use Facebook credentials. Using this approach, you can grant different capabilities to authenticated users via the authenticated role. For instance, you might have a phone application that permits users to view content anonymously, but allows them to post if they are logged on with one or more of the configured providers.

```
CognitoAWSCredentials credentials = new CognitoAWSCredentials(
    accountId, identityPoolId,
    unAuthRoleArn,    // Role for unauthenticated users
    authRoleArn,      // Role for authenticated users
    region);
using (var s3Client = new AmazonS3Client(credentials))
{
    // Initial use will be unauthenticated
    s3Client.ListBuckets();

    // Authenticate user through Facebook
    string facebookToken = GetFacebookAuthToken();

    // Add Facebook login to credentials. This clears the current AWS credentials
    // and retrieves new AWS credentials using the authenticated role.
    credentials.AddLogin("graph.facebook.com", facebookAccessToken);

    // This call is performed with the authenticated role and credentials
    s3Client.ListBuckets();
}
```

The `CognitoAWSCredentials` object provides even more functionality if you use it with the `AmazonCognitoSyncClient` that is part of the AWS SDK for .NET. If you're using both `AmazonCognitoSyncClient` and `CognitoAWSCredentials`, you don't have to specify the `IdentityPoolId` and `IdentityId` properties when making calls with the `AmazonCognitoSyncClient`. These properties are automatically filled in from `CognitoAWSCredentials`. The next code example illustrates this, as well as an event that notifies you whenever the `IdentityId` for `CognitoAWSCredentials` changes. The `IdentityId` can change in some cases, such as when changing from an unauthenticated user to an authenticated one.

```
CognitoAWSCredentials credentials = GetCognitoAWSCredentials();

// Log identity changes
credentials.IdentityChangedEvent += (sender, args) =>
{
    Console.WriteLine("Identity changed: [{0}] => [{1}]", args.OldIdentityId,
 args.NewIdentityId);
};

using (var syncClient = new AmazonCognitoSyncClient(credentials))
{
    var result = syncClient.ListRecords(new ListRecordsRequest
    {
        DatasetName = datasetName
        // No need to specify these properties
        //IdentityId = "...",
        //IdentityPoolId = "..."
    });
}
```

Amazon CognitoAuthentication extension library examples

Note
The information in this topic is specific to projects based on .NET Framework and the AWS SDK for .NET version 3.3 and earlier.

The CognitoAuthentication extension library, found in the Amazon.Extensions.CognitoAuthentication NuGet package, simplifies the authentication process of Amazon Cognito user pools for .NET Core and Xamarin developers. The library is built on top of the Amazon Cognito Identity provider API to create and send user authentication API calls.

Using the CognitoAuthentication extension library

Amazon Cognito has some built-in `AuthFlow` and `ChallengeName` values for a standard authentication flow to validate username and password through the Secure Remote Password (SRP). For more information about authentication flow, see Amazon Cognito User Pool Authentication Flow.

The following examples require these `using` statements:

```
// Required for all examples
using System;
using Amazon;
using Amazon.CognitoIdentity;
using Amazon.CognitoIdentityProvider;
using Amazon.Extensions.CognitoAuthentication;
using Amazon.Runtime;
// Required for the GetS3BucketsAsync example
using Amazon.S3;
using Amazon.S3.Model;
```

Use basic authentication

Create an AmazonCognitoIdentityProviderClient using AnonymousAWSCredentials, which do not require signed requests. You do not need to supply a region, the underlying code calls `FallbackRegionFactory.GetRegionEndpoint()` if a region is not provided. Create `CognitoUserPool` and `CognitoUser` objects. Call the `StartWithSrpAuthAsync` method with an `InitiateSrpAuthRequest` that contains the user password.

```
public static async void GetCredsAsync()
{
    AmazonCognitoIdentityProviderClient provider =
        new AmazonCognitoIdentityProviderClient(new
 Amazon.Runtime.AnonymousAWSCredentials());
    CognitoUserPool userPool = new CognitoUserPool("poolID", "clientID", provider);
    CognitoUser user = new CognitoUser("username", "clientID", userPool, provider);
    InitiateSrpAuthRequest authRequest = new InitiateSrpAuthRequest()
    {
        Password = "userPassword"
    };

    AuthFlowResponse authResponse = await
 user.StartWithSrpAuthAsync(authRequest).ConfigureAwait(false);
    accessToken = authResponse.AuthenticationResult.AccessToken;

}
```

Authenticate with challenges

Continuing the authentication flow with challenges, such as with NewPasswordRequired and Multi-Factor Authentication (MFA), is also simpler. The only requirements are the CognitoAuthentication

objects, user's password for SRP, and the necessary information for the next challenge, which is acquired after prompting the user to enter it. The following code shows one way to check the challenge type and get the appropriate responses for MFA and NewPasswordRequired challenges during the authentication flow.

Do a basic authentication request as before, and `await` an `AuthFlowResponse`. When the response is received loop through the returned `AuthenticationResult` object. If the `ChallengeName` type is `NEW_PASSWORD_REQUIRED`, call the `RespondToNewPasswordRequiredAsync` method.

```
public static async void GetCredsChallengesAsync()
{
    AmazonCognitoIdentityProviderClient provider =
        new AmazonCognitoIdentityProviderClient(new
 Amazon.Runtime.AnonymousAWSCredentials());
    CognitoUserPool userPool = new CognitoUserPool("poolID", "clientID", provider);
    CognitoUser user = new CognitoUser("username", "clientID", userPool, provider);
    InitiateSrpAuthRequest authRequest = new InitiateSrpAuthRequest(){
        Password = "userPassword"
    };

    AuthFlowResponse authResponse = await
 user.StartWithSrpAuthAsync(authRequest).ConfigureAwait(false);

    while (authResponse.AuthenticationResult == null)
    {
        if (authResponse.ChallengeName == ChallengeNameType.NEW_PASSWORD_REQUIRED)
        {
            Console.WriteLine("Enter your desired new password:");
            string newPassword = Console.ReadLine();

            authResponse = await user.RespondToNewPasswordRequiredAsync(new
 RespondToNewPasswordRequiredRequest()
            {
                SessionID = authResponse.SessionID,
                NewPassword = newPassword
            });
            accessToken = authResponse.AuthenticationResult.AccessToken;
        }
        else if (authResponse.ChallengeName == ChallengeNameType.SMS_MFA)
        {
            Console.WriteLine("Enter the MFA Code sent to your device:");
            string mfaCode = Console.ReadLine();

            AuthFlowResponse mfaResponse = await user.RespondToSmsMfaAuthAsync(new
 RespondToSmsMfaRequest()
            {
                SessionID = authResponse.SessionID,
                MfaCode = mfaCode

            }).ConfigureAwait(false);
            accessToken = authResponse.AuthenticationResult.AccessToken;
        }
        else
        {
            Console.WriteLine("Unrecognized authentication challenge.");
            accessToken = "";
            break;
        }
    }

    if (authResponse.AuthenticationResult != null)
    {
        Console.WriteLine("User successfully authenticated.");
    }
```

```
    else
    {
        Console.WriteLine("Error in authentication process.");
    }

}
```

Use AWS resources after authentication

Once a user is authenticated using the CognitoAuthentication library, the next step is to allow the user to access the appropriate AWS resources. To do this you must create an identity pool through the Amazon Cognito Federated Identities console. By specifying the Amazon Cognito user pool you created as a provider, using its poolID and clientID, you can allow your Amazon Cognito user pool users to access AWS resources connected to your account. You can also specify different roles to enable both unauthenticated and authenticated users to access different resources. You can change these rules in the IAM console, where you can add or remove permissions in the **Action** field of the role's attached policy. Then, using the appropriate identity pool, user pool, and Amazon Cognito user information, you can make calls to different AWS resources. The following example shows a user authenticated with SRP accessing the different Amazon S3 buckets permitted by the associated identity pool's role

```
public async void GetS3BucketsAsync()
{
    var provider = new AmazonCognitoIdentityProviderClient(new AnonymousAWSCredentials());
    CognitoUserPool userPool = new CognitoUserPool("poolID", "clientID", provider);
    CognitoUser user = new CognitoUser("username", "clientID", userPool, provider);

    string password = "userPassword";

    AuthFlowResponse context = await user.StartWithSrpAuthAsync(new
 InitiateSrpAuthRequest()
    {
        Password = password
    }).ConfigureAwait(false);

    CognitoAWSCredentials credentials =
        user.GetCognitoAWSCredentials("identityPoolID", RegionEndpoint.<
 YourIdentityPoolRegion >);

    using (var client = new AmazonS3Client(credentials))
    {
        ListBucketsResponse response =
            await client.ListBucketsAsync(new ListBucketsRequest()).ConfigureAwait(false);

        foreach (S3Bucket bucket in response.Buckets)
        {
            Console.WriteLine(bucket.BucketName);
        }
    }
}
```

More authentication options

In addition to SRP, NewPasswordRequired, and MFA, the CognitoAuthentication extension library offers an easier authentication flow for:

- Custom - Initiate with a call to `StartWithCustomAuthAsync(InitiateCustomAuthRequest customRequest)`
- RefreshToken - Initiate with a call to `StartWithRefreshTokenAuthAsync(InitiateRefreshTokenAuthRequest refreshTokenRequest)`

- RefreshTokenSRP - Initiate with a call to `StartWithRefreshTokenAuthAsync(InitiateRefreshTokenAuthRequest refreshTokenRequest)`
- AdminNoSRP - Initiate with a call to `StartWithAdminNoSrpAuthAsync(InitiateAdminNoSrpAuthRequest adminAuthRequest)`

Call the appropriate method depending on the flow you want. Then continue prompting the user with challenges as they are presented in the `AuthFlowResponse` objects of each method call. Also call the appropriate response method, such as `RespondToSmsMfaAuthAsync` for MFA challenges and `RespondToCustomAuthAsync` for custom challenges.

Using Amazon DynamoDB NoSQL databases

Note
The information in this topic is specific to projects based on .NET Framework and the AWS SDK for .NET version 3.3 and earlier.

The AWS SDK for .NET supports Amazon DynamoDB, which is a fast NoSQL database service offered by AWS. The SDK provides three programming models for communicating with DynamoDB: the *low-level* model, the *document* model, and the *object persistence* model.

The following information introduces these models and their APIs, provides examples for how and when to use them, and gives you links to additional DynamoDB programming resources in the AWS SDK for .NET.

Topics

Low-Level Model

The low-level programming model wraps direct calls to the DynamoDB service. You access this model through the Amazon.DynamoDBv2 namespace.

Of the three models, the low-level model requires you to write the most code. For example, you must convert .NET data types to their equivalents in DynamoDB. However, this model gives you access to the most features.

The following examples show you how to use the low-level model to create a table, modify a table, and insert items into a table in DynamoDB.

Creating a Table

In the following example, you create a table by using the `CreateTable` method of the `AmazonDynamoDBClient` class. The `CreateTable` method uses an instance of the `CreateTableRequest` class that contains characteristics such as required item attribute names, primary key definition, and throughput capacity. The `CreateTable` method returns an instance of the `CreateTableResponse` class.

```
// using Amazon.DynamoDBv2;
// using Amazon.DynamoDBv2.Model;

var client = new AmazonDynamoDBClient();

Console.WriteLine("Getting list of tables");
List<string> currentTables = client.ListTables().TableNames;
Console.WriteLine("Number of tables: " + currentTables.Count);
if (!currentTables.Contains("AnimalsInventory"))
{
    var request = new CreateTableRequest
    {
        TableName = "AnimalsInventory",
        AttributeDefinitions = new List<AttributeDefinition>
      {
        new AttributeDefinition
        {
          AttributeName = "Id",
          // "S" = string, "N" = number, and so on.
          AttributeType = "N"
        },
        new AttributeDefinition
        {
          AttributeName = "Type",
          AttributeType = "S"
        }
      },
        KeySchema = new List<KeySchemaElement>
      {
        new KeySchemaElement
        {
          AttributeName = "Id",
          // "HASH" = hash key, "RANGE" = range key.
          KeyType = "HASH"
        },
        new KeySchemaElement
        {
          AttributeName = "Type",
          KeyType = "RANGE"
        },
      },
        ProvisionedThroughput = new ProvisionedThroughput
        {
            ReadCapacityUnits = 10,
            WriteCapacityUnits = 5
        },
    };

    var response = client.CreateTable(request);

    Console.WriteLine("Table created with request ID: " +
      response.ResponseMetadata.RequestId);
}
```

Verifying That a Table is Ready to Modify

Before you can change or modify a table, the table has to be ready for modification. The following example shows how to use the low-level model to verify that a table in DynamoDB is ready. In this example, the target table to check is referenced through the `DescribeTable` method of the `AmazonDynamoDBClient` class. Every five seconds, the code checks the value of the table's `TableStatus` property. When the status is set to `ACTIVE`, the table is ready to be modified.

```
// using Amazon.DynamoDBv2;
```

```
// using Amazon.DynamoDBv2.Model;

var client = new AmazonDynamoDBClient();
var status = "";

do
{
  // Wait 5 seconds before checking (again).
  System.Threading.Thread.Sleep(TimeSpan.FromSeconds(5));

  try
  {
    var response = client.DescribeTable(new DescribeTableRequest
    {
      TableName = "AnimalsInventory"
    });

    Console.WriteLine("Table = {0}, Status = {1}",
      response.Table.TableName,
      response.Table.TableStatus);

    status = response.Table.TableStatus;
  }
  catch (ResourceNotFoundException)
  {
    // DescribeTable is eventually consistent. So you might
    //   get resource not found.
  }

} while (status != TableStatus.ACTIVE);
```

Inserting an Item into a Table

In the following example, you use the low-level model to insert two items into a table in DynamoDB. Each item is inserted through the `PutItem` method of the `AmazonDynamoDBClient` class, using an instance of the `PutItemRequest` class. Each of the two instances of the `PutItemRequest` class takes the name of the table that the items will be inserted in, with a series of item attribute values.

```
// using Amazon.DynamoDBv2;
// using Amazon.DynamoDBv2.Model;

var client = new AmazonDynamoDBClient();

var request1 = new PutItemRequest
{
  TableName = "AnimalsInventory",
  Item = new Dictionary<string, AttributeValue>
  {
    { "Id", new AttributeValue { N = "1" }},
    { "Type", new AttributeValue { S = "Dog" }},
    { "Name", new AttributeValue { S = "Fido" }}
  }
};

var request2 = new PutItemRequest
{
  TableName = "AnimalsInventory",
  Item = new Dictionary<string, AttributeValue>
  {
    { "Id", new AttributeValue { N = "2" }},
    { "Type", new AttributeValue { S = "Cat" }},
    { "Name", new AttributeValue { S = "Patches" }}
  }
};
```

```
client.PutItem(request1);
client.PutItem(request2);
```

Document Model

The document programming model provides an easier way to work with data in DynamoDB. This model is specifically intended for accessing tables and items in tables. You access this model through the Amazon.DynamoDBv2.DocumentModel namespace.

Compared to the low-level programming model, the document model is easier to code against DynamoDB data. For example, you don't have to convert as many .NET data types to their equivalents in DynamoDB. However, this model doesn't provide access to as many features as the low-level programming model. For example, you can use this model to create, retrieve, update, and delete items in tables. However, to create the tables, you must use the low-level model. Compared to the object persistence model, this model requires you to write more code to store, load, and query .NET objects.

The following examples show you how to use the document model to insert items and get items in tables in DynamoDB.

Inserting an Item into a Table

In the following example, an item is inserted into the table through the `PutItem` method of the `Table` class. The `PutItem` method takes an instance of the `Document` class; the `Document` class is simply a collection of initialized attributes. To determine the table to insert the item into, call the `LoadTable` method of the `Table` class, specifying an instance of the `AmazonDynamoDBClient` class and the name of the target table in DynamoDB.

```
// using Amazon.DynamoDBv2;
// using Amazon.DynamoDBv2.DocumentModel;

var client = new AmazonDynamoDBClient();
var table = Table.LoadTable(client, "AnimalsInventory");
var item = new Document();

item["Id"] = 3;
item["Type"] = "Horse";
item["Name"] = "Shadow";

table.PutItem(item);
```

Getting an Item from a Table

In the following example, the item is retrieved through the `GetItem` method of the `Table` class. To determine the item to get, the `GetItem` method uses the hash-and-range primary key of the target item. To determine the table to get the item from, the `LoadTable` method of the `Table` class uses an instance of the `AmazonDynamoDBClient` class and the name of the target table in DynamoDB.

```
// using Amazon.DynamoDBv2;
// using Amazon.DynamoDBv2.DocumentModel;

var client = new AmazonDynamoDBClient();
var table = Table.LoadTable(client, "AnimalsInventory");
var item = table.GetItem(3, "Horse");

Console.WriteLine("Id = " + item["Id"]);
Console.WriteLine("Type = " + item["Type"]);
Console.WriteLine("Name = " + item["Name"]);
```

The preceding example implicitly converts the attribute values for `Id`, `Type`, and `Name` to strings for the `WriteLine` method. You can do explicit conversions by using the various `AsType` methods of the `DynamoDBEntry` class. For example, you could explicitly convert the attribute value for `Id` from a `Primitive` data type to an integer through the `AsInt` method:

```
int id = item["Id"].AsInt();
```

Or, you could simply perform an explicit cast here by using `(int)`:

```
int id = (int)item["Id"];
```

Object Persistence Model

The object persistence programming model is specifically designed for storing, loading, and querying .NET objects in DynamoDB. You access this model through the Amazon.DynamoDBv2.DataModel namespace.

Of the three models, the object persistence model is the easiest to code against whenever you are storing, loading, or querying DynamoDB data. For example, you work with DynamoDB data types directly. However, this model provides access only to operations that store, load, and query .NET objects in DynamoDB. For example, you can use this model to create, retrieve, update, and delete items in tables. However, you must first create your tables using the low-level model, and then use this model to map your .NET classes to the tables.

The following examples show you how to define a .NET class that represents an item, use an instance of the .NET class to insert an item, and use an instance of a .NET object to get an item from a table in DynamoDB.

Defining a .NET Class that Represents an Item in a Table

In the following example, the `DynamoDBTable` attribute specifies the table name, while the `DynamoDBHashKey` and `DynamoDBRangeKey` attributes model the table's hash-and-range primary key.

```
// using Amazon.DynamoDBv2.DataModel;

[DynamoDBTable("AnimalsInventory")]
class Item
{
  [DynamoDBHashKey]
  public int Id { get; set; }
  [DynamoDBRangeKey]
  public string Type { get; set; }
  public string Name { get; set; }
}
```

Using an Instance of the .NET Class to Insert an Item into a Table

In this example, the item is inserted through the `Save` method of the `DynamoDBContext` class, which takes an initialized instance of the .NET class that represents the item. (The instance of the `DynamoDBContext` class is initialized with an instance of the `AmazonDynamoDBClient` class.)

```
// using Amazon.DynamoDBv2;
// using Amazon.DynamoDBv2.DataModel;

var client = new AmazonDynamoDBClient();
var context = new DynamoDBContext(client);
var item = new Item
{
```

```
  Id = 4,
  Type = "Fish",
  Name = "Goldie"
};

context.Save(item);
```

Using an Instance of a .NET Object to Get an Item from a Table

In this example, the item is retrieved through the `Load` method of the `DynamoDBContext` class, which takes a partially initialized instance of the .NET class that represents the hash-and-range primary key of the item to be retrieved. (As shown previously, the instance of the `DynamoDBContext` class is initialized with an instance of the `AmazonDynamoDBClient` class.)

```
// using Amazon.DynamoDBv2;
// using Amazon.DynamoDBv2.DataModel;

var client = new AmazonDynamoDBClient();
var context = new DynamoDBContext(client);
var item = context.Load<Item>(4, "Fish");

Console.WriteLine("Id = {0}", item.Id);
Console.WriteLine("Type = {0}", item.Type);
Console.WriteLine("Name = {0}", item.Name);
```

More Info

Using the AWS SDK for .NET to program DynamoDB information and examples**

- DynamoDB APIs
- DynamoDB Series Kickoff
- DynamoDB Series - Document Model
- DynamoDB Series - Conversion Schemas
- DynamoDB Series - Object Persistence Model
- DynamoDB Series - Expressions
- Using Expressions with Amazon DynamoDB and the AWS SDK for .NET (p. 111)
- JSON Support in Amazon DynamoDB with the AWS SDK for .NET (p. 120)
- Managing ASP.NET session state with Amazon DynamoDB (p. 122)

Low-Level model information and examples

- Working with Tables Using the AWS SDK for .NET Low-Level API
- Working with Items Using the AWS SDK for .NET Low-Level API
- Querying Tables Using the AWS SDK for .NET Low-Level API
- Scanning Tables Using the AWS SDK for .NET Low-Level API
- Working with Local Secondary Indexes Using the AWS SDK for .NET Low-Level API
- Working with Global Secondary Indexes Using the AWS SDK for .NET Low-Level API

Document model information and examples

- DynamoDB Data Types
- DynamoDBEntry
- .NET: Document Model

Object persistence model information and examples

- .NET: Object Persistence Model

Using Expressions with Amazon DynamoDB and the AWS SDK for .NET

> **Note**
> The information in this topic is specific to projects based on .NET Framework and the AWS SDK for .NET version 3.3 and earlier.

The following code examples demonstrate how to use the AWS SDK for .NET to program DynamoDB with expressions. *Expressions* denote the attributes you want to read from an item in a DynamoDB table. You also use expressions when writing an item, to indicate any conditions that must be met (also known as a *conditional update*) and how the attributes are to be updated. Some update examples are replacing the attribute with a new value, or adding new data to a list or a map. For more information, see Reading and Writing Items Using Expressions.

Topics

Sample Data

The code examples in this topic rely on the following two example items in a DynamoDB table named `ProductCatalog`. These items describe information about product entries in a fictitious bicycle store catalog. These items are based on the example provided in Case Study: A ProductCatalog Item. The data type descriptors such as `BOOL`, `L`, `M`, `N`, `NS`, `S`, and `SS` correspond to those in the JSON Data Format.

```
{
  "Id": {
    "N": "205"
  },
  "Title": {
    "S": "20-Bicycle 205"
  },
  "Description": {
    "S": "205 description"
  },
  "BicycleType": {
    "S": "Hybrid"
  },
  "Brand": {
    "S": "Brand-Company C"
  },
  "Price": {
    "N": "500"
  },
  "Gender": {
```

```
    "S": "B"
  },
  "Color": {
    "SS": [
      "Red",
      "Black"
    ]
  },
  "ProductCategory": {
    "S": "Bike"
  },
  "InStock": {
    "BOOL": true
  },
  "QuantityOnHand": {
    "N": "1"
  },
  "RelatedItems": {
    "NS": [
      "341",
      "472",
      "649"
    ]
  },
  "Pictures": {
    "L": [
      {
        "M": {
          "FrontView": {
            "S": "http://example/products/205_front.jpg"
          }
        }
      },
      {
        "M": {
          "RearView": {
            "S": "http://example/products/205_rear.jpg"
          }
        }
      },
      {
        "M": {
          "SideView": {
            "S": "http://example/products/205_left_side.jpg"
          }
        }
      }
    ]
  },
  "ProductReviews": {
    "M": {
      "FiveStar": {
        "SS": [
          "Excellent! Can't recommend it highly enough! Buy it!",
          "Do yourself a favor and buy this."
        ]
      },
      "OneStar": {
        "SS": [
          "Terrible product! Do not buy this."
        ]
      }
    }
  }
},
{
```

```
"Id": {
  "N": "301"
},
"Title": {
  "S": "18-Bicycle 301"
},
"Description": {
  "S": "301 description"
},
"BicycleType": {
  "S": "Road"
},
"Brand": {
  "S": "Brand-Company C"
},
"Price": {
  "N": "185"
},
"Gender": {
  "S": "F"
},
"Color": {
  "SS": [
    "Blue",
    "Silver"
  ]
},
"ProductCategory": {
  "S": "Bike"
},
"InStock": {
  "BOOL": true
},
"QuantityOnHand": {
  "N": "3"
},
"RelatedItems": {
  "NS": [
    "801",
    "822",
    "979"
  ]
},
"Pictures": {
  "L": [
    {
      "M": {
        "FrontView": {
          "S": "http://example/products/301_front.jpg"
        }
      }
    },
    {
      "M": {
        "RearView": {
          "S": "http://example/products/301_rear.jpg"
        }
      }
    },
    {
      "M": {
        "SideView": {
          "S": "http://example/products/301_left_side.jpg"
        }
      }
    }
```

```
    ]
  },
  "ProductReviews": {
    "M": {
      "FiveStar": {
        "SS": [
          "My daughter really enjoyed this bike!"
        ]
      },
      "ThreeStar": {
        "SS": [
          "This bike was okay, but I would have preferred it in my color.",
       "Fun to ride."
        ]
      }
    }
  }
}
```

Get a Single Item by Using Expressions and the Item's Primary Key

The following example features the `Amazon.DynamoDBv2.AmazonDynamoDBClient.GetItem` method and a set of expressions to get and then print the item that has an `Id` of `205`. Only the following attributes of the item are returned: `Id`, `Title`, `Description`, `Color`, `RelatedItems`, `Pictures`, and `ProductReviews`.

```
// using Amazon.DynamoDBv2;
// using Amazon.DynamoDBv2.Model;

var client = new AmazonDynamoDBClient();
var request = new GetItemRequest
{
  TableName = "ProductCatalog",
  ProjectionExpression = "Id, Title, Description, Color, #ri, Pictures, #pr",
  ExpressionAttributeNames = new Dictionary<string, string>
  {
    { "#pr", "ProductReviews" },
    { "#ri", "RelatedItems" }
  },
  Key = new Dictionary<string, AttributeValue>
  {
    { "Id", new AttributeValue { N = "205" } }
  },
};
var response = client.GetItem(request);

// PrintItem() is a custom function.
PrintItem(response.Item);
```

In the preceding example, the `ProjectionExpression` property specifies the attributes to be returned. The `ExpressionAttributeNames` property specifies the placeholder `#pr` to represent the `ProductReviews` attribute and the placeholder `#ri` to represent the `RelatedItems` attribute. The call to `PrintItem` refers to a custom function as described in Print an Item (p. 116).

Get Multiple Items by Using Expressions and the Table's Primary Key

The following example features the `Amazon.DynamoDBv2.AmazonDynamoDBClient.Query` method and a set of expressions to get and then print the item that has an `Id` of `301`, but only if the value of `Price` is greater than `150`. Only the following attributes of the item are returned: `Id`, `Title`, and all of the `ThreeStar` attributes in `ProductReviews`.

```
// using Amazon.DynamoDBv2;
```

```
// using Amazon.DynamoDBv2.Model;

var client = new AmazonDynamoDBClient();
var request = new QueryRequest
{
  TableName = "ProductCatalog",
  KeyConditions = new Dictionary<string,Condition>
  {
    { "Id", new Condition()
      {
        ComparisonOperator = ComparisonOperator.EQ,
        AttributeValueList = new List<AttributeValue>
        {
          new AttributeValue { N = "301" }
        }
      }
    }
  },
  ProjectionExpression = "Id, Title, #pr.ThreeStar",
  ExpressionAttributeNames = new Dictionary<string, string>
  {
    { "#pr", "ProductReviews" },
    { "#p", "Price" }
  },
  ExpressionAttributeValues = new Dictionary<string,AttributeValue>
  {
    { ":val", new AttributeValue { N = "150" } }
  },
  FilterExpression = "#p > :val"
};
var response = client.Query(request);

foreach (var item in response.Items)
{
  // Write out the first page of an item's attribute keys and values.
  // PrintItem() is a custom function.
  PrintItem(item);
  Console.WriteLine("=====");
}
```

In the preceding example, the `ProjectionExpression` property specifies the attributes to be returned. The `ExpressionAttributeNames` property specifies the placeholder `#pr` to represent the `ProductReviews` attribute and the placeholder `#p` to represent the `Price` attribute. `#pr.ThreeStar` specifies to return only the `ThreeStar` attribute. The `ExpressionAttributeValues` property specifies the placeholder `:val` to represent the value `150`. The `FilterExpression` property specifies that `#p` (`Price`) must be greater than `:val` (`150`). The call to `PrintItem` refers to a custom function as described in Print an Item (p. 116).

Get Multiple Items by Using Expressions and Other Item Attributes

The following example features the `Amazon.DynamoDBv2.AmazonDynamoDBClient.Scan` method and a set of expressions to get and then print all items that have a `ProductCategory` of `Bike`. Only the following attributes of the item are returned: `Id`, `Title`, and all of the attributes in `ProductReviews`.

```
// using Amazon.DynamoDBv2;
// using Amazon.DynamoDBv2.Model;

var client = new AmazonDynamoDBClient();
var request = new ScanRequest
{
  TableName = "ProductCatalog",
  ProjectionExpression = "Id, Title, #pr",
  ExpressionAttributeValues = new Dictionary<string,AttributeValue>
```

```
  {
    { ":catg", new AttributeValue { S = "Bike" } }
  },
  ExpressionAttributeNames = new Dictionary<string, string>
  {
    { "#pr", "ProductReviews" },
    { "#pc", "ProductCategory" }
  },
  FilterExpression = "#pc = :catg",
};
var response = client.Scan(request);

foreach (var item in response.Items)
{
  // Write out the first page/scan of an item's attribute keys and values.
  // PrintItem() is a custom function.
  PrintItem(item);
  Console.WriteLine("=====");
}
```

In the preceding example, the `ProjectionExpression` property specifies the attributes to be returned. The `ExpressionAttributeNames` property specifies the placeholder `#pr` to represent the `ProductReviews` attribute and the placeholder `#pc` to represent the `ProductCategory` attribute. The `ExpressionAttributeValues` property specifies the placeholder `:catg` to represent the value `Bike`. The `FilterExpression` property specifies that `#pc` (`ProductCategory`) must be equal to `:catg` (`Bike`). The call to `PrintItem` refers to a custom function as described in Print an Item (p. 116).

Print an Item

The following example shows how to print an item's attributes and values. This example is used in the preceding examples that show how to Get a Single Item by Using Expressions and the Item's Primary Key (p. 114), Get Multiple Items by Using Expressions and the Table's Primary Key (p. 114), and Get Multiple Items by Using Expressions and Other Item Attributes (p. 115).

```
// using Amazon.DynamoDBv2.Model;

// Writes out an item's attribute keys and values.
public static void PrintItem(Dictionary<string, AttributeValue> attrs)
{
  foreach (KeyValuePair<string, AttributeValue> kvp in attrs)
  {
    Console.Write(kvp.Key + " = ");
    PrintValue(kvp.Value);
  }
}

// Writes out just an attribute's value.
public static void PrintValue(AttributeValue value)
{
  // Binary attribute value.
  if (value.B != null)
  {
    Console.Write("Binary data");
  }
  // Binary set attribute value.
  else if (value.BS.Count > 0)
  {
    foreach (var bValue in value.BS)
    {
      Console.Write("\n  Binary data");
    }
  }
  // List attribute value.
```

```
  else if (value.L.Count > 0)
  {
    foreach (AttributeValue attr in value.L)
    {
      PrintValue(attr);
    }
  }
  // Map attribute value.
  else if (value.M.Count > 0)
  {
    Console.Write("\n");
    PrintItem(value.M);
  }
  // Number attribute value.
  else if (value.N != null)
  {
    Console.Write(value.N);
  }
  // Number set attribute value.
  else if (value.NS.Count > 0)
  {
    Console.Write("{0}", string.Join("\n", value.NS.ToArray()));
  }
  // Null attribute value.
  else if (value.NULL)
  {
    Console.Write("Null");
  }
  // String attribute value.
  else if (value.S != null)
  {
    Console.Write(value.S);
  }
  // String set attribute value.
  else if (value.SS.Count > 0)
  {
    Console.Write("{0}", string.Join("\n", value.SS.ToArray()));
  }
  // Otherwise, boolean value.
  else
  {
    Console.Write(value.BOOL);
  }

  Console.Write("\n");
}
```

In the preceding example, each attribute value has several data-type-specific properties that can be evaluated to determine the correct format to print the attribute. These properties include `B`, `BOOL`, `BS`, `L`, `M`, `N`, `NS`, `NULL`, `S`, and `SS`, which correspond to those in the JSON Data Format. For properties such as `B`, `N`, `NULL`, and `S`, if the corresponding property is not `null`, then the attribute is of the corresponding non-`null` data type. For properties such as `BS`, `L`, `M`, `NS`, and `SS`, if `Count` is greater than zero, then the attribute is of the corresponding non-zero-value data type. If all of the attribute's data-type-specific properties are either `null` or the `Count` equals zero, then the attribute corresponds to the `BOOL` data type.

Create or Replace an Item by Using Expressions

The following example features the `Amazon.DynamoDBv2.AmazonDynamoDBClient.PutItem` method and a set of expressions to update the item that has a `Title` of `18-Bicycle 301`. If the item doesn't already exist, a new item is added.

```
// using Amazon.DynamoDBv2;
```

```
// using Amazon.DynamoDBv2.Model;

var client = new AmazonDynamoDBClient();
var request = new PutItemRequest
{
  TableName = "ProductCatalog",
  ExpressionAttributeNames = new Dictionary<string, string>
  {
    { "#title", "Title" }
  },
  ExpressionAttributeValues = new Dictionary<string, AttributeValue>
  {
    { ":product", new AttributeValue { S = "18-Bicycle 301" } }
  },
  ConditionExpression = "#title = :product",
  // CreateItemData() is a custom function.
  Item = CreateItemData()
};
client.PutItem(request);
```

In the preceding example, the `ExpressionAttributeNames` property specifies the placeholder `#title` to represent the `Title` attribute. The `ExpressionAttributeValues` property specifies the placeholder `:product` to represent the value `18-Bicycle 301`. The `ConditionExpression` property specifies that `#title` (`Title`) must be equal to `:product` (`18-Bicycle 301`). The call to `CreateItemData` refers to the following custom function:

```
// using Amazon.DynamoDBv2.Model;

// Provides a sample item that can be added to a table.
public static Dictionary<string, AttributeValue> CreateItemData()
{
  var itemData = new Dictionary<string, AttributeValue>
  {
    { "Id", new AttributeValue { N = "301" } },
    { "Title", new AttributeValue { S = "18\" Girl's Bike" } },
    { "BicycleType", new AttributeValue { S = "Road" } },
    { "Brand" , new AttributeValue { S = "Brand-Company C" } },
    { "Color", new AttributeValue { SS = new List<string>{ "Blue", "Silver" } } },
    { "Description", new AttributeValue { S = "301 description" } },
    { "Gender", new AttributeValue { S = "F" } },
    { "InStock", new AttributeValue { BOOL = true } },
    { "Pictures", new AttributeValue { L = new List<AttributeValue>{
      { new AttributeValue { M = new Dictionary<string,AttributeValue>{
        { "FrontView", new AttributeValue { S = "http://example/
products/301_front.jpg" } } } } },
      { new AttributeValue { M = new Dictionary<string,AttributeValue>{
        { "RearView", new AttributeValue {S = "http://example/
products/301_rear.jpg" } } } } },
      { new AttributeValue { M = new Dictionary<string,AttributeValue>{
        { "SideView", new AttributeValue { S = "http://example/
products/301_left_side.jpg" } } } } }
    } } },
    { "Price", new AttributeValue { N = "185" } },
    { "ProductCategory", new AttributeValue { S = "Bike" } },
    { "ProductReviews", new AttributeValue { M = new Dictionary<string,AttributeValue>{
      { "FiveStar", new AttributeValue { SS = new List<string>{
        "My daughter really enjoyed this bike!" } } },
      { "OneStar", new AttributeValue { SS = new List<string>{
        "Fun to ride.",
        "This bike was okay, but I would have preferred it in my color." } } }
    } } },
    { "QuantityOnHand", new AttributeValue { N = "3" } },
    { "RelatedItems", new AttributeValue { NS = new List<string>{ "979", "822", "801" } } }
  };
```

```
  return itemData;
}
```

In the preceding example, an example item with sample data is returned to the caller. A series of attributes and corresponding values are constructed, using data types such as `BOOL`, `L`, `M`, `N`, `NS`, `S`, and `SS`, which correspond to those in the JSON Data Format.

Update an Item by Using Expressions

The following example features the `Amazon.DynamoDBv2.AmazonDynamoDBClient.UpdateItem` method and a set of expressions to change the `Title` to `18" Girl's Bike` for the item with `Id` of `301`.

```
// using Amazon.DynamoDBv2;
// using Amazon.DynamoDBv2.Model;

var client = new AmazonDynamoDBClient();
var request = new UpdateItemRequest
{
  TableName = "ProductCatalog",
  Key = new Dictionary<string,AttributeValue>
  {
     { "Id", new AttributeValue { N = "301" } }
  },
  ExpressionAttributeNames = new Dictionary<string, string>
  {
    { "#title", "Title" }
  },
  ExpressionAttributeValues = new Dictionary<string, AttributeValue>
  {
    { ":newproduct", new AttributeValue { S = "18\" Girl's Bike" } }
  },
  UpdateExpression = "SET #title = :newproduct"
};
client.UpdateItem(request);
```

In the preceding example, the `ExpressionAttributeNames` property specifies the placeholder `#title` to represent the `Title` attribute. The `ExpressionAttributeValues` property specifies the placeholder `:newproduct` to represent the value `18" Girl's Bike`. The `UpdateExpression` property specifies to change `#title` (`Title`) to `:newproduct` (`18" Girl's Bike`).

Delete an Item by Using Expressions

The following example features the `Amazon.DynamoDBv2.AmazonDynamoDBClient.DeleteItem` method and a set of expressions to delete the item with `Id` of `301`, but only if the item's `Title` is `18-Bicycle 301`.

```
// using Amazon.DynamoDBv2;
// using Amazon.DynamoDBv2.Model;

var client = new AmazonDynamoDBClient();
var request = new DeleteItemRequest
{
  TableName = "ProductCatalog",
  Key = new Dictionary<string,AttributeValue>
  {
     { "Id", new AttributeValue { N = "301" } }
  },
  ExpressionAttributeNames = new Dictionary<string, string>
  {
```

```
    { "#title", "Title" }
  },
  ExpressionAttributeValues = new Dictionary<string, AttributeValue>
  {
    { ":product", new AttributeValue { S = "18-Bicycle 301" } }
  },
  ConditionExpression = "#title = :product"
};
client.DeleteItem(request);
```

In the preceding example, the `ExpressionAttributeNames` property specifies the placeholder `#title` to represent the `Title` attribute. The `ExpressionAttributeValues` property specifies the placeholder `:product` to represent the value `18-Bicycle 301`. The `ConditionExpression` property specifies that `#title` (`Title`) must equal `:product` (`18-Bicycle 301`).

More Info

For more information and code examples, see:

- DynamoDB Series - Expressions
- Accessing Item Attributes with Projection Expressions
- Using Placeholders for Attribute Names and Values
- Specifying Conditions with Condition Expressions
- Modifying Items and Attributes with Update Expressions
- Working with Items Using the AWS SDK for .NET Low-Level API
- Querying Tables Using the AWS SDK for .NET Low-Level API
- Scanning Tables Using the AWS SDK for .NET Low-Level API
- Working with Local Secondary Indexes Using the AWS SDK for .NET Low-Level API
- Working with Global Secondary Indexes Using the AWS SDK for .NET Low-Level API

JSON Support in Amazon DynamoDB with the AWS SDK for .NET

> **Note**
> The information in this topic is specific to projects based on .NET Framework and the AWS SDK for .NET version 3.3 and earlier.

The AWS SDK for .NET supports JSON data when working with Amazon DynamoDB. This enables you to more easily get JSON-formatted data from, and insert JSON documents into, DynamoDB tables.

Topics

Get Data from a DynamoDB Table in JSON Format

The following example shows how to get data from a DynamoDB table in JSON format:

```
// using Amazon.DynamoDBv2;
// using Amazon.DynamoDBv2.DocumentModel;
```

```
var client = new AmazonDynamoDBClient();
var table = Table.LoadTable(client, "AnimalsInventory");
var item = table.GetItem(3, "Horse");

var jsonText = item.ToJson();
Console.Write(jsonText);

// Output:
//   {"Name":"Shadow","Type":"Horse","Id":3}

var jsonPrettyText = item.ToJsonPretty();
Console.WriteLine(jsonPrettyText);

// Output:
//   {
//     "Name" : "Shadow",
//     "Type" : "Horse",
//     "Id"   : 3
//   }
```

In the preceding example, the `ToJson` method of the `Document` class converts an item from the table into a JSON-formatted string. The item is retrieved through the `GetItem` method of the `Table` class. To determine the item to get, in this example, the `GetItem` method uses the hash-and-range primary key of the target item. To determine the table to get the item from, the `LoadTable` method of the `Table` class uses an instance of the `AmazonDynamoDBClient` class and the name of the target table in DynamoDB.

Insert JSON Format Data into a DynamoDB Table

The following example shows how to use JSON format to insert an item into a DynamoDB table:

```
// using Amazon.DynamoDBv2;
// using Amazon.DynamoDBv2.DocumentModel;

var client = new AmazonDynamoDBClient();
var table = Table.LoadTable(client, "AnimalsInventory");
var jsonText = "{\"Id\":6,\"Type\":\"Bird\",\"Name\":\"Tweety\"}";
var item = Document.FromJson(jsonText);

table.PutItem(item);
```

In the preceding example, the `FromJson` method of the `Document` class converts a JSON-formatted string into an item. The item is inserted into the table through the `PutItem` method of the `Table` class, which uses the instance of the `Document` class that contains the item. To determine the table to insert the item into, the `LoadTable` method of the `Table` class is called, specifying an instance of the `AmazonDynamoDBClient` class and the name of the target table in DynamoDB.

DynamoDB Data Type Conversions to JSON

Whenever you call the `ToJson` method of the `Document` class, and then on the resulting JSON data you call the `FromJson` method to convert the JSON data back into an instance of a `Document` class, some DynamoDB data types will not be converted as expected. Specifically:

- DynamoDB sets (the `SS`, `NS`, and `BS` types) will be converted to JSON arrays.
- DynamoDB binary scalars and sets (the `B` and `BS` types) will be converted to base64-encoded JSON strings or lists of strings.

  In this scenario, you must call the `DecodeBase64Attributes` method of the `Document` class to replace the base64-encoded JSON data with the correct binary representation. The following example replaces a base64-encoded binary scalar item attribute in an instance of a `Document`

class, named `Picture`, with the correct binary representation. This example also does the same for a base64-encoded binary set item attribute in the same instance of the `Document` class, named `RelatedPictures`:

```
item.DecodeBase64Attributes("Picture", "RelatedPictures");
```

More Info

For more information and examples of programming JSON with DynamoDB with the AWS SDK for .NET, see:

- DynamoDB JSON Support
- Amazon DynamoDB Update - JSON, Expanded Free Tier, Flexible Scaling, Larger Items

Managing ASP.NET session state with Amazon DynamoDB

Note
The information in this topic is specific to projects based on .NET Framework and the AWS SDK for .NET version 3.3 and earlier.

Warning
This topic is specific to ASP.NET; the information in this topic isn't necessarily applicable to ASP.NET Core.

ASP.NET applications often store session state data in memory. However, this approach doesn't scale well. After the application grows beyond a single web server, the session state must be shared between servers. A common solution is to set up a dedicated session-state server with Microsoft SQL Server, but this approach also has drawbacks: you must administer another machine; the session-state server is a single point of failure; and the session-state server itself can become a performance bottleneck.

DynamoDB, a NoSQL database store from AWS, provides an effective solution for sharing session state across web servers without incurring any of these drawbacks.

Note
Regardless of the solution you choose, be aware that Amazon DynamoDB enforces limits on the size of an item. None of the records you store in DynamoDB can exceed this limit. For more information, see Limits in DynamoDB in the Amazon DynamoDB Developer Guide.

The AWS SDK for .NET includes `AWS.SessionProvider.dll`, which contains an ASP.NET session state provider. It also includes the *AmazonDynamoDBSessionProviderSample* sample, which demonstrates how to use Amazon DynamoDB as a session state provider.

For more information about using session state with ASP.NET applications, go to the Microsoft documentation.

Create the *ASP.NET_SessionState* Table

When your application starts, it looks for an Amazon DynamoDB table named, by default, `ASP.NET_SessionState`. We recommend you create this table before you run your application for the first time.

To create the ASP.NET_SessionState table

1. Choose **Create Table**. The **Create Table** wizard opens.
2. In the **Table name** text box, enter `ASP.NET_SessionState`.
3. In the **Primary key** field, enter `SessionId` and set the type to `String`.

4. When all your options are entered as you want them, choose **Create**.

The `ASP.NET_SessionState` table is ready for use when its status changes from `CREATING` to `ACTIVE`.

> **Note**
> If you decide not to create the table beforehand, the session state provider will create the table during its initialization. See the `web.config` options below for a list of attributes that act as configuration parameters for the session state table. If the provider creates the table, it will use these parameters.

Configure the Session State Provider

To configure an ASP.NET application to use DynamoDB as the session-state server

1. Add references to both `AWSSDK.dll` and `AWS.SessionProvider.dll` to your Visual Studio ASP.NET project. These assemblies are available through NuGet packages (p. 28) or by installing assemblies manually (p. 30).

   In earlier versions of the SDK, the functionality for the session state provider was contained in `AWS.Extension.dll`. To improve usability, the functionality was moved to `AWS.SessionProvider.dll`. For more information, see the blog post `AWS.Extension` renaming.
2. Edit your application's `Web.config` file. In the `system.web` element, replace the existing `sessionState` element with the following XML fragment:

```
<sessionState timeout="20" mode="Custom" customProvider="DynamoDBSessionStoreProvider">
  <providers>
    <add name="DynamoDBSessionStoreProvider"
         type="Amazon.SessionProvider.DynamoDBSessionStateStore"
         AWSProfileName="{profile_name}"
         Region="us-west-2" />
  </providers>
</sessionState>
```

   The profile represents the AWS credentials that are used to communicate with DynamoDB to store and retrieve the session state. If you are using the AWS SDK for .NET and are specifying a profile in the `appSettings` section of your application's `Web.config` file, you do not need to specify a profile in the `providers` section; the AWS .NET client code will discover it at run time. For more information, see Configuring Your AWS SDK for .NET Application (p. 17).

   If the web server is running on an Amazon EC2 instance configured to use IAM roles for EC2 instances, then you do not need to specify any credentials in the `Web.config` file. In this case, the AWS .NET client will use the IAM role credentials. For more information, see Granting Access Using an IAM Role (p. 183) and Security Considerations (p. 124).

Web.config Options

You can use the following configuration attributes in the `providers` section of your `Web.config` file:

AWSAccessKey

Access key ID to use. This can be set either in the `providers` or `appSettings` section. We recommend not using this setting. Instead, specify credentials by using `AWSProfileName` to specify a profile.

AWSSecretKey

Secret key to use. This can be set either in the `providers` or `appSettings` section. We recommend not using this setting. Instead, specify credentials by using `AWSProfileName` to specify a profile.

AWSProfileName

The profile name associated with the credentials you want to use. For more information, see Configuring Your AWS SDK for .NET Application (p. 17).

Region

Required `string` attribute. The AWS region in which to use Amazon DynamoDB. For a list of AWS regions, see Regions and Endpoints: DynamoDB.

Application

Optional `string` attribute. The value of the `Application` attribute is used to partition the session data in the table so that the table can be used for more than one application.

Table

Optional `string` attribute. The name of the table used to store session data. The default is `ASP.NET_SessionState`.

ReadCapacityUnits

Optional `int` attribute. The read capacity units to use if the provider creates the table. The default is 10.

WriteCapacityUnits

Optional `int` attribute. The write capacity units to use if the provider creates the table. The default is 5.

CreateIfNotExist

Optional `boolean` attribute. The `CreateIfNotExist` attribute controls whether the provider will auto-create the table if it doesn't exist. The default is true. If this flag is set to false and the table doesn't exist, an exception will be thrown.

Security Considerations

After the DynamoDB table is created and the application is configured, sessions can be used as with any other session provider.

As a security best practice, we recommend you run your applications with the credentials of an IAM User Guide user. You can use either the IAM Management Console or the AWS Toolkit for Visual Studio to create IAM users and define access policies.

The session state provider needs to be able to call the DeleteItem, DescribeTable, GetItem, PutItem, and UpdateItem operations for the table that stores the session data. The sample policy below can be used to restrict the IAM user to only the operations needed by the provider for an instance of DynamoDB running in us-west-2:

```
{ "Version" : "2012-10-17",
"Statement" : [
  {
    "Sid" : "1",
    "Effect" : "Allow",
    "Action" : [
        "dynamodb:DeleteItem",
        "dynamodb:DescribeTable",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:UpdateItem"
    ],
    "Resource" : "arn:aws:dynamodb:us-west-2:{<YOUR-AWS-ACCOUNT-ID>}:table/
ASP.NET_SessionState"
    }
```

```
  ]
}
```

Working with Amazon EC2

The AWS SDK for .NET supports Amazon EC2, which is a web service that provides resizable computing capacity. You use this computing capacity to build and host your software systems.

APIs

The AWS SDK for .NET provides APIs for Amazon EC2 clients. The APIs enable you to work with EC2 features such as security groups and key pairs. The APIs also enable you to control Amazon EC2 instances. This section contains a small number of examples that show you the patterns you can follow when working with these APIs. To view the full set of APIs, see the AWS SDK for .NET API Reference (and scroll to "Amazon.EC2").

The Amazon EC2 APIs are provided by the AWSSDK.EC2 NuGet package.

Prerequisites

Before you begin, be sure you have set up your environment (p. 15). Also review the information in Setting up your project (p. 17) and SDK features (p. 49).

About the examples

The examples in this section show you how to work with Amazon EC2 clients and manage Amazon EC2 instances.

The EC2 Spot Instance tutorial (p. 158) shows you how to request Amazon EC2 Spot Instances. Spot Instances enable you to access unused EC2 capacity for less than the On-Demand price.

Topics

Working with security groups in Amazon EC2

In Amazon EC2, a *security group* acts as a virtual firewall that controls the network traffic for one or more EC2 instances. By default, EC2 associates your instances with a security group that allows no inbound traffic. You can create a security group that allows your EC2 instances to accept certain traffic. For example, if you need to connect to an EC2 Windows instance, you must configure the security group to allow RDP traffic.

Read more about security groups in the Amazon EC2 User Guide for Linux Instances and the Amazon EC2 User Guide for Windows Instances.

When using the AWS SDK for .NET, you can create a security group for use in EC2 in a VPC or EC2-Classic. For more information about EC2 in a VPC versus EC2-Classic, see the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances.

> **Warning**
> We are retiring EC2-Classic on August 15, 2022. We recommend that you migrate from EC2-Classic to a VPC. For more information, see **Migrate from EC2-Classic to a VPC** in the Amazon

EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances. Also see the blog post EC2-Classic Networking is Retiring – Here's How to Prepare.

For information about the APIs and prerequisites, see the parent section (Working with Amazon EC2 (p. 125)).

Topics

Enumerating security groups

This example shows you how to use the AWS SDK for .NET to enumerate security groups. If you supply an Amazon Virtual Private Cloud ID, the application enumerates the security groups for that particular VPC. Otherwise, the application simply displays a list of all available security groups.

The following sections provide snippets of this example. The complete code for the example (p. 127) is shown after that, and can be built and run as is.

Topics

Enumerate security groups

The following snippet enumerates your security groups. It enumerates all groups or the groups for a particular VPC if one is given.

The example at the end of this topic (p. 127) shows this snippet in use.

```
//
// Method to enumerate the security groups
private static async Task EnumerateGroups(IAmazonEC2 ec2Client, string vpcID)
{
  // A request object, in case we need it.
  var request = new DescribeSecurityGroupsRequest();

  // Put together the properties, if needed
  if(!string.IsNullOrEmpty(vpcID))
  {
    // We have a VPC ID. Find the security groups for just that VPC.
    Console.WriteLine($"\nGetting security groups for VPC {vpcID}...\n");
    request.Filters.Add(new Filter
    {
      Name = "vpc-id",
      Values = new List<string>() { vpcID }
    });
  }

  // Get the list of security groups
  DescribeSecurityGroupsResponse response =
    await ec2Client.DescribeSecurityGroupsAsync(request);

  // Display the list of security groups.
  foreach (SecurityGroup item in response.SecurityGroups)
  {
    Console.WriteLine("Security group: " + item.GroupId);
```

```
        Console.WriteLine("\tGroupId: " + item.GroupId);
        Console.WriteLine("\tGroupName: " + item.GroupName);
        Console.WriteLine("\tVpcId: " + item.VpcId);
        Console.WriteLine();
      }
    }
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.EC2

Programming elements:

- Namespace Amazon.EC2

  Class AmazonEC2Client
- Namespace Amazon.EC2.Model

  Class DescribeSecurityGroupsRequest

  Class DescribeSecurityGroupsResponse

  Class Filter

  Class SecurityGroup

The Code

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
using Amazon.EC2;
using Amazon.EC2.Model;

namespace EC2EnumerateSecGroups
{
  class Program
  {
    static async Task Main(string[] args)
    {
      // Parse the command line
       string vpcID = string.Empty;
      if(args.Length == 0)
      {
        Console.WriteLine("\nEC2EnumerateSecGroups [vpc_id]");
        Console.WriteLine("  vpc_id - The ID of the VPC for which you want to see security
 groups.");
        Console.WriteLine("\nSince you specified no arguments, showing all available
 security groups.");
      }
      else
      {
        vpcID = args[0];
      }
```

```
      if(vpcID.StartsWith("vpc-") || string.IsNullOrEmpty(vpcID))
      {
        // Create an EC2 client object
        var ec2Client = new AmazonEC2Client();

        // Enumerate the security groups
        await EnumerateGroups(ec2Client, vpcID);
      }
      else
      {
        Console.WriteLine("Could not find a valid VPC ID in the command-line arguments:");
        Console.WriteLine($"{args[0]}");
      }
    }


    //
    // Method to enumerate the security groups
    private static async Task EnumerateGroups(IAmazonEC2 ec2Client, string vpcID)
    {
      // A request object, in case we need it.
      var request = new DescribeSecurityGroupsRequest();

      // Put together the properties, if needed
      if(!string.IsNullOrEmpty(vpcID))
      {
        // We have a VPC ID. Find the security groups for just that VPC.
        Console.WriteLine($"\nGetting security groups for VPC {vpcID}...\n");
        request.Filters.Add(new Filter
        {
          Name = "vpc-id",
          Values = new List<string>() { vpcID }
        });
      }

      // Get the list of security groups
      DescribeSecurityGroupsResponse response =
        await ec2Client.DescribeSecurityGroupsAsync(request);

      // Display the list of security groups.
      foreach (SecurityGroup item in response.SecurityGroups)
      {
        Console.WriteLine("Security group: " + item.GroupId);
        Console.WriteLine("\tGroupId: " + item.GroupId);
        Console.WriteLine("\tGroupName: " + item.GroupName);
        Console.WriteLine("\tVpcId: " + item.VpcId);
        Console.WriteLine();
      }
    }
  }
}
```

Additional considerations

- Notice for the VPC case that the filter is constructed with the `Name` part of the name-value pair set to "vpc-id". This name comes from the description for the `Filters` property of the DescribeSecurityGroupsRequest class.

- To get the complete list of your security groups, you can also use DescribeSecurityGroupsAsync with no parameters.

- You can verify the results by checking the list of security groups in the Amazon EC2 console.

Creating security groups

This example shows you how to use the AWS SDK for .NET to create a security group. You can provide the ID of an existing VPC to create a security group for EC2 in a VPC. If you don't supply such an ID, the new security group will be for EC2-Classic if your AWS account supports this.

If you don't supply a VPC ID and your AWS account doesn't support EC2-Classic, the new security group will belong to the default VPC of your account. For more information, see the references for EC2 in a VPC versus EC2-Classic in the parent section (Working with security groups in Amazon EC2 (p. 125)).

The following sections provide snippets of this example. The complete code for the example (p. 130) is shown after that, and can be built and run as is.

Topics

Find existing security groups

The following snippet searches for existing security groups with the given name in the given VPC.

The example at the end of this topic (p. 130) shows this snippet in use.

```
//
// Method to determine if a security group with the specified name
// already exists in the VPC
private static async Task<List<SecurityGroup>> FindSecurityGroups(
  IAmazonEC2 ec2Client, string groupName, string vpcID)
{
  var request = new DescribeSecurityGroupsRequest();
  request.Filters.Add(new Filter{
    Name = "group-name",
    Values = new List<string>() { groupName }
  });
  if(!string.IsNullOrEmpty(vpcID))
    request.Filters.Add(new Filter{
      Name = "vpc-id",
      Values = new List<string>() { vpcID }
    });

  var response = await ec2Client.DescribeSecurityGroupsAsync(request);
  return response.SecurityGroups;
}
```

Create a security group

The following snippet creates a new security group if a group with that name doesn't exist in the given VPC. If no VPC is given and one or more groups with that name exist, the snippet simply returns the list of groups.

The example at the end of this topic (p. 130) shows this snippet in use.

```
//
// Method to create a new security group (either EC2-Classic or EC2-VPC)
// If vpcID is empty, the security group will be for EC2-Classic
private static async Task<List<SecurityGroup>> CreateSecurityGroup(
  IAmazonEC2 ec2Client, string groupName, string vpcID)
{
```

```
    // See if one or more security groups with that name
    // already exist in the given VPC. If so, return the list of them.
    var securityGroups = await FindSecurityGroups(ec2Client, groupName, vpcID);
    if (securityGroups.Count > 0)
    {
      Console.WriteLine(
        $"\nOne or more security groups with name {groupName} already exist.\n");
      return securityGroups;
    }

    // If the security group doesn't already exists, create it.
    var createRequest = new CreateSecurityGroupRequest{
      GroupName = groupName
    };
    if(string.IsNullOrEmpty(vpcID))
    {
      createRequest.Description = "My .NET example security group for EC2-Classic";
    }
    else
    {
      createRequest.VpcId = vpcID;
      createRequest.Description = "My .NET example security group for EC2-VPC";
    }
    CreateSecurityGroupResponse createResponse =
      await ec2Client.CreateSecurityGroupAsync(createRequest);

    // Return the new security group
    DescribeSecurityGroupsResponse describeResponse =
      await ec2Client.DescribeSecurityGroupsAsync(new DescribeSecurityGroupsRequest{
        GroupIds = new List<string>() { createResponse.GroupId }
      });
    return describeResponse.SecurityGroups;
  }
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.EC2

Programming elements:

- Namespace Amazon.EC2

  Class AmazonEC2Client
- Namespace Amazon.EC2.Model

  Class CreateSecurityGroupRequest

  Class CreateSecurityGroupResponse

  Class DescribeSecurityGroupsRequest

  Class DescribeSecurityGroupsResponse

  Class Filter

  Class SecurityGroup

The code

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
using Amazon.EC2;
using Amazon.EC2.Model;

namespace EC2CreateSecGroup
{
  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 =
  // Class to create a security group
  class Program
  {
    private const int MaxArgs = 2;

    static async Task Main(string[] args)
    {
      // Parse the command line and show help if necessary
      var parsedArgs = CommandLine.Parse(args);
      if(parsedArgs.Count == 0)
      {
        PrintHelp();
        return;
      }
      if(parsedArgs.Count > MaxArgs)
        CommandLine.ErrorExit("\nThe number of command-line arguments is incorrect." +
          "\nRun the command with no arguments to see help.");

      // Get the application arguments from the parsed list
      var groupName = CommandLine.GetArgument(parsedArgs, null, "-g", "--group-name");
      var vpcID = CommandLine.GetArgument(parsedArgs, null, "-v", "--vpc-id");
      if(string.IsNullOrEmpty(groupName))
        CommandLine.ErrorExit("\nYou must supply a name for the new group." +
          "\nRun the command with no arguments to see help.");
      if(!string.IsNullOrEmpty(vpcID) && !vpcID.StartsWith("vpc-"))
        CommandLine.ErrorExit($"\nNot a valid VPC ID: {vpcID}");

      // groupName has a value and vpcID either has a value or is null (which is fine)
      // Create the new security group and display information about it
      var securityGroups =
        await CreateSecurityGroup(new AmazonEC2Client(), groupName, vpcID);
      Console.WriteLine("Information about the security group(s):");
      foreach(var group in securityGroups)
      {
        Console.WriteLine($"\nGroupName: {group.GroupName}");
        Console.WriteLine($"GroupId: {group.GroupId}");
        Console.WriteLine($"Description: {group.Description}");
        Console.WriteLine($"VpcId (if any): {group.VpcId}");
      }
    }


    //
    // Method to create a new security group (either EC2-Classic or EC2-VPC)
    // If vpcID is empty, the security group will be for EC2-Classic
    private static async Task<List<SecurityGroup>> CreateSecurityGroup(
      IAmazonEC2 ec2Client, string groupName, string vpcID)
    {
      // See if one or more security groups with that name
      // already exist in the given VPC. If so, return the list of them.
      var securityGroups = await FindSecurityGroups(ec2Client, groupName, vpcID);
      if (securityGroups.Count > 0)
      {
```

```
        Console.WriteLine(
          $"\nOne or more security groups with name {groupName} already exist.\n");
        return securityGroups;
      }

      // If the security group doesn't already exists, create it.
      var createRequest = new CreateSecurityGroupRequest{
        GroupName = groupName
      };
      if(string.IsNullOrEmpty(vpcID))
      {
        createRequest.Description = "Security group for .NET code example (no VPC
specified)";
      }
      else
      {
        createRequest.VpcId = vpcID;
        createRequest.Description = "Security group for .NET code example (VPC: " + vpcID +
")";
      }
      CreateSecurityGroupResponse createResponse =
        await ec2Client.CreateSecurityGroupAsync(createRequest);

      // Return the new security group
      DescribeSecurityGroupsResponse describeResponse =
        await ec2Client.DescribeSecurityGroupsAsync(new DescribeSecurityGroupsRequest{
          GroupIds = new List<string>() { createResponse.GroupId }
        });
      return describeResponse.SecurityGroups;
    }


    //
    // Method to determine if a security group with the specified name
    // already exists in the VPC
    private static async Task<List<SecurityGroup>> FindSecurityGroups(
      IAmazonEC2 ec2Client, string groupName, string vpcID)
    {
      var request = new DescribeSecurityGroupsRequest();
      request.Filters.Add(new Filter{
        Name = "group-name",
        Values = new List<string>() { groupName }
      });
      if(!string.IsNullOrEmpty(vpcID))
        request.Filters.Add(new Filter{
          Name = "vpc-id",
          Values = new List<string>() { vpcID }
        });

      var response = await ec2Client.DescribeSecurityGroupsAsync(request);
      return response.SecurityGroups;
    }


    //
    // Command-line help
    private static void PrintHelp()
    {
      Console.WriteLine(
        "\nUsage: EC2CreateSecGroup -g <group-name> [-v <vpc-id>]" +
        "\n  -g, --group-name: The name you would like the new security group to have." +
        "\n  -v, --vpc-id: The ID of a VPC to which the new security group will belong." +
        "\n     If vpc-id isn't present, the security group will be" +
        "\n     for EC2-Classic (if your AWS account supports this)" +
        "\n     or will use the default VCP for EC2-VPC.");
    }
```

```
 }


 // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
=
 // Class that represents a command line on the console or terminal.
 // (This is the same for all examples. When you have seen it once, you can ignore it.)
 static class CommandLine
 {
   //
   // Method to parse a command line of the form: "--key value" or "-k value".
   //
   // Parameters:
   // - args: The command-line arguments passed into the application by the system.
   //
   // Returns:
   // A Dictionary with string Keys and Values.
   //
   // If a key is found without a matching value, Dictionary.Value is set to the key
   //  (including the dashes).
   // If a value is found without a matching key, Dictionary.Key is set to "--NoKeyN",
   //  where "N" represents sequential numbers.
   public static Dictionary<string,string> Parse(string[] args)
   {
     var parsedArgs = new Dictionary<string,string>();
     int i = 0, n = 0;
     while(i < args.Length)
     {
       // If the first argument in this iteration starts with a dash it's an option.
       if(args[i].StartsWith("-"))
       {
         var key = args[i++];
         var value = key;

         // Check to see if there's a value that goes with this option?
         if((i < args.Length) && (!args[i].StartsWith("-"))) value = args[i++];
         parsedArgs.Add(key, value);
       }

       // If the first argument in this iteration doesn't start with a dash, it's a value
       else
       {
         parsedArgs.Add("--NoKey" + n.ToString(), args[i++]);
         n++;
       }
     }

     return parsedArgs;
   }

   //
   // Method to get an argument from the parsed command-line arguments
   //
   // Parameters:
   // - parsedArgs: The Dictionary object returned from the Parse() method (shown above).
   // - defaultValue: The default string to return if the specified key isn't in
parsedArgs.
   // - keys: An array of keys to look for in parsedArgs.
   public static string GetArgument(
     Dictionary<string,string> parsedArgs, string defaultReturn, params string[] keys)
   {
     string retval = null;
     foreach(var key in keys)
       if(parsedArgs.TryGetValue(key, out retval)) break;
     return retval ?? defaultReturn;
   }
```

```
    //
    // Method to exit the application with an error.
    public static void ErrorExit(string msg, int code=1)
    {
      Console.WriteLine("\nError");
      Console.WriteLine(msg);
      Environment.Exit(code);
    }
  }

}
```

Updating security groups

This example shows you how to use the AWS SDK for .NET to add a rule to a security group. In particular, the example adds a rule to allow inbound traffic on a given TCP port, which can be used, for example, for remote connections to an EC2 instance. The application takes the ID of an existing security group, an IP address (or address range) in CIDR format, and optionally a TCP port number. It then adds an inbound rule to the given security group.

> **Note**
> To use this example, you need an IP address (or address range) in CIDR format. See **Additional considerations** at this end of this topic for methods to obtain the IP address of your local computer.

The following sections provide snippets of this example. The complete code for the example (p. 135) is shown after that, and can be built and run as is.

Topics

Add an inbound rule

The following snippet adds an inbound rule to a security group for a particular IP address (or range) and TCP port.

The example at the end of this topic (p. 135) shows this snippet in use.

```
    //
    // Method that adds a TCP ingress rule to a security group
    private static async Task AddIngressRule(
      IAmazonEC2 eC2Client, string groupID, string ipAddress, int port)
    {
      // Create an object to hold the request information for the rule.
      // It uses an IpPermission object to hold the IP information for the rule.
      var ingressRequest = new AuthorizeSecurityGroupIngressRequest{
        GroupId = groupID};
      ingressRequest.IpPermissions.Add(new IpPermission{
        IpProtocol = "tcp",
        FromPort = port,
        ToPort = port,
        Ipv4Ranges = new List<IpRange>() { new IpRange { CidrIp = ipAddress } }
      });

      // Create the inbound rule for the security group
      AuthorizeSecurityGroupIngressResponse responseIngress =
        await eC2Client.AuthorizeSecurityGroupIngressAsync(ingressRequest);
      Console.WriteLine($"\nNew RDP rule was written in {groupID} for {ipAddress}.");
```

```
      Console.WriteLine($"Result: {responseIngress.HttpStatusCode}");
    }
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.EC2

Programming elements:

- Namespace Amazon.EC2

  Class AmazonEC2Client
- Namespace Amazon.EC2.Model

  Class AuthorizeSecurityGroupIngressRequest

  Class AuthorizeSecurityGroupIngressResponse

  Class IpPermission

  Class IpRange

The code

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
using Amazon.EC2;
using Amazon.EC2.Model;

namespace EC2AddRuleForRDP
{
  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 =
  // Class to add a rule that allows inbound traffic on TCP a port
  class Program
  {
    private const int DefaultPort = 3389;

    static async Task Main(string[] args)
    {
      // Parse the command line and show help if necessary
      var parsedArgs = CommandLine.Parse(args);
      if(parsedArgs.Count == 0)
      {
        PrintHelp();
        return;
      }

      // Get the application arguments from the parsed list
      var groupID = CommandLine.GetArgument(parsedArgs, null, "-g", "--group-id");
      var ipAddress = CommandLine.GetArgument(parsedArgs, null, "-i", "--ip-address");
      var portStr = CommandLine.GetArgument(parsedArgs, DefaultPort.ToString(), "-p", "--
port");
      if(string.IsNullOrEmpty(ipAddress))
```

```
        CommandLine.ErrorExit("\nYou must supply an IP address in CIDR format.");
      if(string.IsNullOrEmpty(groupID) || !groupID.StartsWith("sg-"))
        CommandLine.ErrorExit("\nThe ID for a security group is missing or incorrect.");
      if(int.Parse(portStr) == 0)
        CommandLine.ErrorExit($"\nThe given TCP port number, {portStr}, isn't allowed.");

      // Add a rule to the given security group that allows
      // inbound traffic on a TCP port
      await AddIngressRule(
        new AmazonEC2Client(), groupID, ipAddress, int.Parse(portStr));
    }


    //
    // Method that adds a TCP ingress rule to a security group
    private static async Task AddIngressRule(
      IAmazonEC2 eC2Client, string groupID, string ipAddress, int port)
    {
      // Create an object to hold the request information for the rule.
      // It uses an IpPermission object to hold the IP information for the rule.
      var ingressRequest = new AuthorizeSecurityGroupIngressRequest{
        GroupId = groupID};
      ingressRequest.IpPermissions.Add(new IpPermission{
        IpProtocol = "tcp",
        FromPort = port,
        ToPort = port,
        Ipv4Ranges = new List<IpRange>() { new IpRange { CidrIp = ipAddress } }
      });

      // Create the inbound rule for the security group
      AuthorizeSecurityGroupIngressResponse responseIngress =
        await eC2Client.AuthorizeSecurityGroupIngressAsync(ingressRequest);
      Console.WriteLine($"\nNew RDP rule was written in {groupID} for {ipAddress}.");
      Console.WriteLine($"Result: {responseIngress.HttpStatusCode}");
    }


    //
    // Command-line help
    private static void PrintHelp()
    {
      Console.WriteLine(
        "\nUsage: EC2AddRuleForRDP -g <group-id> -i <ip-address> [-p <port>]" +
        "\n  -g, --group-id: The ID of the security group to which you want to add the
inbound rule." +
        "\n  -i, --ip-address: An IP address or address range in CIDR format." +
        "\n  -p, --port: The TCP port number. Defaults to 3389.");
    }
  }


  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
=
  // Class that represents a command line on the console or terminal.
  // (This is the same for all examples. When you have seen it once, you can ignore it.)
  static class CommandLine
  {
    //
    // Method to parse a command line of the form: "--key value" or "-k value".
    //
    // Parameters:
    // - args: The command-line arguments passed into the application by the system.
    //
    // Returns:
    // A Dictionary with string Keys and Values.
    //
```

```
    // If a key is found without a matching value, Dictionary.Value is set to the key
    //  (including the dashes).
    // If a value is found without a matching key, Dictionary.Key is set to "--NoKeyN",
    //  where "N" represents sequential numbers.
    public static Dictionary<string,string> Parse(string[] args)
    {
      var parsedArgs = new Dictionary<string,string>();
      int i = 0, n = 0;
      while(i < args.Length)
      {
        // If the first argument in this iteration starts with a dash it's an option.
        if(args[i].StartsWith("-"))
        {
          var key = args[i++];
          var value = key;

          // Check to see if there's a value that goes with this option?
          if((i < args.Length) && (!args[i].StartsWith("-"))) value = args[i++];
          parsedArgs.Add(key, value);
        }

        // If the first argument in this iteration doesn't start with a dash, it's a value
        else
        {
          parsedArgs.Add("--NoKey" + n.ToString(), args[i++]);
          n++;
        }
      }

      return parsedArgs;
    }

    //
    // Method to get an argument from the parsed command-line arguments
    //
    // Parameters:
    // - parsedArgs: The Dictionary object returned from the Parse() method (shown above).
    // - defaultValue: The default string to return if the specified key isn't in
 parsedArgs.
    // - keys: An array of keys to look for in parsedArgs.
    public static string GetArgument(
      Dictionary<string,string> parsedArgs, string defaultReturn, params string[] keys)
    {
      string retval = null;
      foreach(var key in keys)
        if(parsedArgs.TryGetValue(key, out retval)) break;
      return retval ?? defaultReturn;
    }

    //
    // Method to exit the application with an error.
    public static void ErrorExit(string msg, int code=1)
    {
      Console.WriteLine("\nError");
      Console.WriteLine(msg);
      Environment.Exit(code);
    }
  }

}
```

Additional considerations

- If you don't supply a port number, the application defaults to port 3389. This is the port for Windows RDP, which enables you to connect to an EC2 instance running Windows. If you're launching an EC2 instance running Linux, you can use TCP port 22 (SSH) instead.

- Notice that the example sets `IpProtocol` to "tcp". The values for `IpProtocol` can be found in the description for the `IpProtocol` property of the IpPermission class.

- You might want the IP address of your local computer when you use this example. The following are some of the ways in which you can obtain the address.
  - If your local computer (from which you will connect to your EC2 instance) has a static public IP address, you can use a service to get that address. One such service is http://checkip.amazonaws.com/. Read more about authorizing inbound traffic in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances.
  - Another way to obtain the IP address of your local computer is to use the Amazon EC2 console.

    Select one of your security groups, select the **Inbound rules** tab, and choose **Edit inbound rules**. In an inbound rule, open the drop-down menu in the **Source** column and choose **My IP** to see the IP address of your local computer in CIDR format. Be sure to **Cancel** the operation.

- You can verify the results of this example by examining the list of security groups in the Amazon EC2 console.

Working with Amazon EC2 key pairs

Amazon EC2 uses public–key cryptography to encrypt and decrypt login information. Public–key cryptography uses a public key to encrypt data, and then the recipient uses the private key to decrypt the data. The public and private keys are known as a key pair. If you want to log into an EC2 instance, you must specify a key pair when you launch it, and then provide the private key of the pair when you connect to it.

When you launch an EC2 instance, you can create a key pair for it or use one that you've already used when launching other instances. Read more about Amazon EC2 key pairs in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances.

For information about the APIs and prerequisites, see the parent section (Working with Amazon EC2 (p. 125)).

Topics

Creating and displaying key pairs

This example shows you how to use the AWS SDK for .NET to create a key pair. The application takes the name for the new key pair and the name of a PEM file (with a ".pem" extension). It creates the keypair, writes the private key to the PEM file, and then displays all available key pairs. If you provide no command-line arguments, the application simply displays all available key pairs.

The following sections provide snippets of this example. The complete code for the example (p. 139) is shown after that, and can be built and run as is.

Topics

Create the key pair

The following snippet creates a key pair and then stores the private key to the given PEM file.

The example at the end of this topic (p. 139) shows this snippet in use.

```
//
// Method to create a key pair and save the key material in a PEM file
private static async Task CreateKeyPair(
  IAmazonEC2 ec2Client, string keyPairName, string pemFileName)
{
  // Create the key pair
  CreateKeyPairResponse response =
    await ec2Client.CreateKeyPairAsync(new CreateKeyPairRequest{
      KeyName = keyPairName
    });
  Console.WriteLine($"\nCreated new key pair: {response.KeyPair.KeyName}");

  // Save the private key in a PEM file
  using (var s = new FileStream(pemFileName, FileMode.Create))
  using (var writer = new StreamWriter(s))
  {
    writer.WriteLine(response.KeyPair.KeyMaterial);
  }
}
```

Display available key pairs

The following snippet displays a list of the available key pairs.

The example at the end of this topic (p. 139) shows this snippet in use.

```
//
// Method to show the key pairs that are available
private static async Task EnumerateKeyPairs(IAmazonEC2 ec2Client)
{
  DescribeKeyPairsResponse response = await ec2Client.DescribeKeyPairsAsync();
  Console.WriteLine("Available key pairs:");
  foreach (KeyPairInfo item in response.KeyPairs)
    Console.WriteLine($"  {item.KeyName}");
}
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.EC2

Programming elements:

- Namespace Amazon.EC2

  Class AmazonEC2Client
- Namespace Amazon.EC2.Model

  Class CreateKeyPairRequest

  Class CreateKeyPairResponse

  Class DescribeKeyPairsResponse

  Class KeyPairInfo

The code

```
using System;
using System.Threading.Tasks;
using System.IO;
using Amazon.EC2;
using Amazon.EC2.Model;
using System.Collections.Generic;

namespace EC2CreateKeyPair
{
  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 =
  // Class to create and store a key pair
  class Program
  {
    static async Task Main(string[] args)
    {
      // Create the EC2 client
      var ec2Client = new AmazonEC2Client();

      // Parse the command line and show help if necessary
      var parsedArgs = CommandLine.Parse(args);
      if(parsedArgs.Count == 0)
      {
        // In the case of no command-line arguments,
        // just show help and the existing key pairs
        PrintHelp();
        Console.WriteLine("\nNo arguments specified.");
        Console.Write(
          "Do you want to see a list of the existing key pairs? ((y) or n): ");
        string response = Console.ReadLine();
        if((string.IsNullOrEmpty(response)) || (response.ToLower() == "y"))
          await EnumerateKeyPairs(ec2Client);
        return;
      }

      // Get the application arguments from the parsed list
      string keyPairName =
        CommandLine.GetArgument(parsedArgs, null, "-k", "--keypair-name");
      string pemFileName =
        CommandLine.GetArgument(parsedArgs, null, "-p", "--pem-filename");
      if(string.IsNullOrEmpty(keyPairName))
        CommandLine.ErrorExit("\nNo key pair name specified." +
          "\nRun the command with no arguments to see help.");
      if(string.IsNullOrEmpty(pemFileName) || !pemFileName.EndsWith(".pem"))
        CommandLine.ErrorExit("\nThe PEM filename is missing or incorrect." +
          "\nRun the command with no arguments to see help.");

      // Create the key pair
```

```
      await CreateKeyPair(ec2Client, keyPairName, pemFileName);
      await EnumerateKeyPairs(ec2Client);
    }


    //
    // Method to create a key pair and save the key material in a PEM file
    private static async Task CreateKeyPair(
      IAmazonEC2 ec2Client, string keyPairName, string pemFileName)
    {
      // Create the key pair
      CreateKeyPairResponse response =
        await ec2Client.CreateKeyPairAsync(new CreateKeyPairRequest{
          KeyName = keyPairName
        });
      Console.WriteLine($"\nCreated new key pair: {response.KeyPair.KeyName}");

      // Save the private key in a PEM file
      using (var s = new FileStream(pemFileName, FileMode.Create))
      using (var writer = new StreamWriter(s))
      {
        writer.WriteLine(response.KeyPair.KeyMaterial);
      }
    }


    //
    // Method to show the key pairs that are available
    private static async Task EnumerateKeyPairs(IAmazonEC2 ec2Client)
    {
      DescribeKeyPairsResponse response = await ec2Client.DescribeKeyPairsAsync();
      Console.WriteLine("Available key pairs:");
      foreach (KeyPairInfo item in response.KeyPairs)
        Console.WriteLine($"  {item.KeyName}");
    }


    //
    // Command-line help
    private static void PrintHelp()
    {
      Console.WriteLine(
        "\nUsage: EC2CreateKeyPair -k <keypair-name> -p <pem-filename>" +
        "\n  -k, --keypair-name: The name you want to assign to the key pair." +
        "\n  -p, --pem-filename: The name of the PEM file to create, with a \".pem\"
extension.");
    }
  }


  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
=
  // Class that represents a command line on the console or terminal.
  // (This is the same for all examples. When you have seen it once, you can ignore it.)
  static class CommandLine
  {
    //
    // Method to parse a command line of the form: "--key value" or "-k value".
    //
    // Parameters:
    // - args: The command-line arguments passed into the application by the system.
    //
    // Returns:
    // A Dictionary with string Keys and Values.
    //
    // If a key is found without a matching value, Dictionary.Value is set to the key
```

```
    //  (including the dashes).
    // If a value is found without a matching key, Dictionary.Key is set to "--NoKeyN",
    //  where "N" represents sequential numbers.
    public static Dictionary<string,string> Parse(string[] args)
    {
      var parsedArgs = new Dictionary<string,string>();
      int i = 0, n = 0;
      while(i < args.Length)
      {
        // If the first argument in this iteration starts with a dash it's an option.
        if(args[i].StartsWith("-"))
        {
          var key = args[i++];
          var value = key;

          // Check to see if there's a value that goes with this option?
          if((i < args.Length) && (!args[i].StartsWith("-"))) value = args[i++];
          parsedArgs.Add(key, value);
        }

        // If the first argument in this iteration doesn't start with a dash, it's a value
        else
        {
          parsedArgs.Add("--NoKey" + n.ToString(), args[i++]);
          n++;
        }
      }

      return parsedArgs;
    }

    //
    // Method to get an argument from the parsed command-line arguments
    //
    // Parameters:
    // - parsedArgs: The Dictionary object returned from the Parse() method (shown above).
    // - defaultValue: The default string to return if the specified key isn't in
 parsedArgs.
    // - keys: An array of keys to look for in parsedArgs.
    public static string GetArgument(
      Dictionary<string,string> parsedArgs, string defaultReturn, params string[] keys)
    {
      string retval = null;
      foreach(var key in keys)
        if(parsedArgs.TryGetValue(key, out retval)) break;
      return retval ?? defaultReturn;
    }

    //
    // Method to exit the application with an error.
    public static void ErrorExit(string msg, int code=1)
    {
      Console.WriteLine("\nError");
      Console.WriteLine(msg);
      Environment.Exit(code);
    }
  }

}
```

Additional considerations

- After you run the example, you can see the new key pair in the Amazon EC2 console.

- When you create a key pair, you must save the private key that is returned because you can't retrieve the private key later.

Deleting key pairs

This example shows you how to use the AWS SDK for .NET to delete a key pair. The application takes the name of a key pair. It deletes the key pair and then displays all available key pairs. If you provide no command-line arguments, the application simply displays all available key pairs.

The following sections provide snippets of this example. The complete code for the example (p. 143) is shown after that, and can be built and run as is.

Topics

Delete the key pair

The following snippet deletes a key pair.

The example at the end of this topic (p. 143) shows this snippet in use.

```
//
// Method to delete a key pair
private static async Task DeleteKeyPair(IAmazonEC2 ec2Client, string keyName)
{
  await ec2Client.DeleteKeyPairAsync(new DeleteKeyPairRequest{
    KeyName = keyName});
  Console.WriteLine($"\nKey pair {keyName} has been deleted (if it existed).");
}
```

Display available key pairs

The following snippet displays a list of the available key pairs.

The example at the end of this topic (p. 143) shows this snippet in use.

```
//
// Method to show the key pairs that are available
private static async Task EnumerateKeyPairs(IAmazonEC2 ec2Client)
{
  DescribeKeyPairsResponse response = await ec2Client.DescribeKeyPairsAsync();
  Console.WriteLine("Available key pairs:");
  foreach (KeyPairInfo item in response.KeyPairs)
    Console.WriteLine($"  {item.KeyName}");
}
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.EC2

Programming elements:

- Namespace Amazon.EC2

  Class AmazonEC2Client
- Namespace Amazon.EC2.Model

  Class DeleteKeyPairRequest

  Class DescribeKeyPairsResponse

  Class KeyPairInfo

The code

```
using System;
using System.Threading.Tasks;
using Amazon.EC2;
using Amazon.EC2.Model;

namespace EC2DeleteKeyPair
{
  class Program
  {
    static async Task Main(string[] args)
    {
      // Create the EC2 client
      var ec2Client = new AmazonEC2Client();

      if(args.Length == 1)
      {
        // Delete a key pair (if it exists)
        await DeleteKeyPair(ec2Client, args[0]);

        // Display the key pairs that are left
        await EnumerateKeyPairs(ec2Client);
      }
      else
      {
        Console.WriteLine("\nUsage: EC2DeleteKeyPair keypair-name");
        Console.WriteLine("  keypair-name - The name of the key pair you want to delete.");
        Console.WriteLine("\nNo arguments specified.");
        Console.Write(
          "Do you want to see a list of the existing key pairs? ((y) or n): ");
        string response = Console.ReadLine();
        if((string.IsNullOrEmpty(response)) || (response.ToLower() == "y"))
          await EnumerateKeyPairs(ec2Client);
      }
    }


    //
    // Method to delete a key pair
    private static async Task DeleteKeyPair(IAmazonEC2 ec2Client, string keyName)
    {
      await ec2Client.DeleteKeyPairAsync(new DeleteKeyPairRequest{
        KeyName = keyName});
      Console.WriteLine($"\nKey pair {keyName} has been deleted (if it existed).");
    }


    //
    // Method to show the key pairs that are available
```

```
    private static async Task EnumerateKeyPairs(IAmazonEC2 ec2Client)
    {
      DescribeKeyPairsResponse response = await ec2Client.DescribeKeyPairsAsync();
      Console.WriteLine("Available key pairs:");
      foreach (KeyPairInfo item in response.KeyPairs)
        Console.WriteLine($"  {item.KeyName}");
    }
  }
}
```

Seeing your Amazon EC2 Regions and Availability Zones

Amazon EC2 is hosted in multiple locations worldwide. These locations are composed of Regions and Availability Zones. Each Region is a separate geographic area that has multiple, isolated locations known as Availability Zones.

Read more about Regions and Availability Zones in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances.

This example shows you how to use the AWS SDK for .NET to get details about the Regions and Availability Zones related to an EC2 client. The application displays lists of the Regions and Availability Zones available to an EC2 client.

SDK references

NuGet packages:

- AWSSDK.EC2

Programming elements:

- Namespace Amazon.EC2

  Class AmazonEC2Client
- Namespace Amazon.EC2.Model

  Class DescribeAvailabilityZonesResponse

  Class DescribeRegionsResponse

  Class AvailabilityZone

  Class Region

```
using System;
using System.Threading.Tasks;
using Amazon.EC2;
using Amazon.EC2.Model;

namespace EC2RegionsAndZones
{
  class Program
  {
    static async Task Main(string[] args)
    {
      Console.WriteLine(
        "Finding the Regions and Availability Zones available to an EC2 client...");

      // Create the EC2 client
```

```
      var ec2Client = new AmazonEC2Client();

      // Display the Regions and Availability Zones
      await DescribeRegions(ec2Client);
      await DescribeAvailabilityZones(ec2Client);
    }


    //
    // Method to display Regions
    private static async Task DescribeRegions(IAmazonEC2 ec2Client)
    {
      Console.WriteLine("\nRegions that are enabled for the EC2 client:");
      DescribeRegionsResponse response = await ec2Client.DescribeRegionsAsync();
      foreach (Region region in response.Regions)
        Console.WriteLine(region.RegionName);
    }


    //
    // Method to display Availability Zones
    private static async Task DescribeAvailabilityZones(IAmazonEC2 ec2Client)
    {
      Console.WriteLine("\nAvailability Zones for the EC2 client's region:");
      DescribeAvailabilityZonesResponse response =
        await ec2Client.DescribeAvailabilityZonesAsync();
      foreach (AvailabilityZone az in response.AvailabilityZones)
        Console.WriteLine(az.ZoneName);
    }
  }
}
```

Working with Amazon EC2 instances

You can use the AWS SDK for .NET to control Amazon EC2 instances with operations such as create, start, and terminate. The topics in this section provide some examples of how to do this. Read more about EC2 instances in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances

For information about the APIs and prerequisites, see the parent section (Working with Amazon EC2 (p. 125)).

Topics

Launching an Amazon EC2 instance

This example shows you how to use the AWS SDK for .NET to launch one or more identically configured Amazon EC2 instances from the same Amazon Machine Image (AMI). Using several inputs (p. 147) that you supply, the application launches an EC2 instance and then monitors the instance until it's out of the "Pending" state.

When your EC2 instance is running, you can connect to it remotely, as described in (optional) Connect to the instance (p. 154).

You can launch an EC2 instance in a VPC or in EC2-Classic (if your AWS account supports this). For more information about EC2 in a VPC versus EC2-Classic, see the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances.

Warning
We are retiring EC2-Classic on August 15, 2022. We recommend that you migrate from EC2-Classic to a VPC. For more information, see **Migrate from EC2-Classic to a VPC** in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances. Also see the blog post EC2-Classic Networking is Retiring – Here's How to Prepare.

The following sections provide snippets and other information for this example. The complete code for the example (p. 149) is shown after the snippets, and can be built and run as is.

Topics

Gather what you need

To launch an EC2 instance, you'll need several things.

- A VPC where the instance will be launched. If it'll be a Windows instance and you'll be connecting to it through RDP, the VPC will most likely need to have an internet gateway attached to it, as well as an entry for the internet gateway in the route table. For more information, see Internet gateways in the *Amazon VPC User Guide*.

- The ID of an existing subnet in the VPC where the instance will be launched. An easy way to find or create this is to sign in to the Amazon VPC console, but you can also obtain it programmatically by using the CreateSubnetAsync and DescribeSubnetsAsync methods.

  Note
  If your AWS account supports EC2-Classic and that's the type of instance you want to launch, this parameter isn't required. However, if your account doesn't support EC2-Classic and you don't supply this parameter, the new instance is launched in the default VPC for your account.

- The ID of an existing security group that belongs to the VPC where the instance will be launched. For more information, see Working with security groups in Amazon EC2 (p. 125).

- If you want to connect to the new instance, the security group mentioned earlier must have an appropriate inbound rule that allows SSH traffic on port 22 (Linux instance) or RDP traffic on port 3389 (Windows instance). For information about how to do this see Updating security groups (p. 134), including the Additional considerations (p. 138) near the end of that topic.

- The Amazon Machine Image (AMI) that will be used to create the instance. See the information about AMIs in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances. For example, read about shared AMIs in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances.

- The name of an existing EC2 key pair, which is used to connect to the new instance. For more information, see Working with Amazon EC2 key pairs (p. 138).

- The name of the PEM file that contains the private key of the EC2 key pair mentioned earlier. The PEM file is used when you connect remotely (p. 154) to the instance.

Launch an instance

The following snippet launches an EC2 instance.

The example near the end of this topic (p. 149) shows this snippet in use.

```
//
// Method to launch the instances
// Returns a list with the launched instance IDs
private static async Task<List<string>> LaunchInstances(
  IAmazonEC2 ec2Client, RunInstancesRequest requestLaunch)
{
  var instanceIds = new List<string>();
  RunInstancesResponse responseLaunch =
    await ec2Client.RunInstancesAsync(requestLaunch);

  Console.WriteLine("\nNew instances have been created.");
  foreach (Instance item in responseLaunch.Reservation.Instances)
  {
    instanceIds.Add(item.InstanceId);
    Console.WriteLine($"  New instance: {item.InstanceId}");
  }

  return instanceIds;
}
```

Monitor the instance

The following snippet monitors the instance until it's out of the "Pending" state.

The example near the end of this topic (p. 149) shows this snippet in use.

See the InstanceState class for the valid values of the `Instance.State.Code` property.

```
//
// Method to wait until the instances are running (or at least not pending)
private static async Task CheckState(IAmazonEC2 ec2Client, List<string> instanceIds)
{
  Console.WriteLine(
    "\nWaiting for the instances to start." +
    "\nPress any key to stop waiting. (Response might be slightly delayed.)");

  int numberRunning;
  DescribeInstancesResponse responseDescribe;
  var requestDescribe = new DescribeInstancesRequest{
    InstanceIds = instanceIds};

  // Check every couple of seconds
  int wait = 2000;
  while(true)
  {
    // Get and check the status for each of the instances to see if it's past pending.
    // Once all instances are past pending, break out.
    // (For this example, we are assuming that there is only one reservation.)
    Console.Write(".");
    numberRunning = 0;
    responseDescribe = await ec2Client.DescribeInstancesAsync(requestDescribe);
    foreach(Instance i in responseDescribe.Reservations[0].Instances)
    {
```

```
        // Check the lower byte of State.Code property
        // Code == 0 is the pending state
        if((i.State.Code & 255) > 0) numberRunning++;
      }
      if(numberRunning == responseDescribe.Reservations[0].Instances.Count)
        break;

      // Wait a bit and try again (unless the user wants to stop waiting)
      Thread.Sleep(wait);
      if(Console.KeyAvailable)
        break;
    }

    Console.WriteLine("\nNo more instances are pending.");
    foreach(Instance i in responseDescribe.Reservations[0].Instances)
    {
      Console.WriteLine($"For {i.InstanceId}:");
      Console.WriteLine($"  VPC ID: {i.VpcId}");
      Console.WriteLine($"  Instance state: {i.State.Name}");
      Console.WriteLine($"  Public IP address: {i.PublicIpAddress}");
      Console.WriteLine($"  Public DNS name: {i.PublicDnsName}");
      Console.WriteLine($"  Key pair name: {i.KeyName}");
    }
  }
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.EC2

Programming elements:

- Namespace Amazon.EC2

  Class AmazonEC2Client

  Class InstanceType
- Namespace Amazon.EC2.Model

  Class DescribeInstancesRequest

  Class DescribeInstancesResponse

  Class Instance

  Class InstanceNetworkInterfaceSpecification

  Class RunInstancesRequest

  Class RunInstancesResponse

The code

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

```
using System.Collections.Generic;
using Amazon.EC2;
using Amazon.EC2.Model;

namespace EC2LaunchInstance
{
  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 =
  // Class to launch an EC2 instance
  class Program
  {
    static async Task Main(string[] args)
    {
      // Parse the command line and show help if necessary
      var parsedArgs = CommandLine.Parse(args);
      if(parsedArgs.Count == 0)
      {
        PrintHelp();
        return;
      }

      // Get the application arguments from the parsed list
      string groupID =
        CommandLine.GetArgument(parsedArgs, null, "-g", "--group-id");
      string ami =
        CommandLine.GetArgument(parsedArgs, null, "-a", "--ami-id");
      string keyPairName =
        CommandLine.GetArgument(parsedArgs, null, "-k", "--keypair-name");
      string subnetID =
        CommandLine.GetArgument(parsedArgs, null, "-s", "--subnet-id");
      if(   (string.IsNullOrEmpty(groupID) || !groupID.StartsWith("sg-"))
         || (string.IsNullOrEmpty(ami) || !ami.StartsWith("ami-"))
         || (string.IsNullOrEmpty(keyPairName))
         || (!string.IsNullOrEmpty(subnetID) && !subnetID.StartsWith("subnet-")))
        CommandLine.ErrorExit(
          "\nOne or more of the required arguments is missing or incorrect." +
          "\nRun the command with no arguments to see help.");

      // Create an EC2 client
      var ec2Client = new AmazonEC2Client();

      // Create an object with the necessary properties
      RunInstancesRequest request = GetRequestData(groupID, ami, keyPairName, subnetID);

      // Launch the instances and wait for them to start running
      var instanceIds = await LaunchInstances(ec2Client, request);
      await CheckState(ec2Client, instanceIds);
    }


    //
    // Method to put together the properties needed to launch the instance.
    private static RunInstancesRequest GetRequestData(
      string groupID, string ami, string keyPairName, string subnetID)
    {
      // Common properties
      var groupIDs = new List<string>() { groupID };
      var request = new RunInstancesRequest()
      {
        // The first three of these would be additional command-line arguments or similar.
        InstanceType = InstanceType.T1Micro,
        MinCount = 1,
        MaxCount = 1,
        ImageId = ami,
        KeyName = keyPairName
      };
```

```
  // Properties specifically for EC2 in a VPC.
  if(!string.IsNullOrEmpty(subnetID))
  {
    request.NetworkInterfaces =
      new List<InstanceNetworkInterfaceSpecification>() {
        new InstanceNetworkInterfaceSpecification() {
          DeviceIndex = 0,
          SubnetId = subnetID,
          Groups = groupIDs,
          AssociatePublicIpAddress = true
        }
      };
  }

  // Properties specifically for EC2-Classic
  else
  {
    request.SecurityGroupIds = groupIDs;
  }
  return request;
}


//
// Method to launch the instances
// Returns a list with the launched instance IDs
private static async Task<List<string>> LaunchInstances(
  IAmazonEC2 ec2Client, RunInstancesRequest requestLaunch)
{
  var instanceIds = new List<string>();
  RunInstancesResponse responseLaunch =
    await ec2Client.RunInstancesAsync(requestLaunch);

  Console.WriteLine("\nNew instances have been created.");
  foreach (Instance item in responseLaunch.Reservation.Instances)
  {
    instanceIds.Add(item.InstanceId);
    Console.WriteLine($"  New instance: {item.InstanceId}");
  }

  return instanceIds;
}


//
// Method to wait until the instances are running (or at least not pending)
private static async Task CheckState(IAmazonEC2 ec2Client, List<string> instanceIds)
{
  Console.WriteLine(
    "\nWaiting for the instances to start." +
    "\nPress any key to stop waiting. (Response might be slightly delayed.)");

  int numberRunning;
  DescribeInstancesResponse responseDescribe;
  var requestDescribe = new DescribeInstancesRequest{
    InstanceIds = instanceIds};

  // Check every couple of seconds
  int wait = 2000;
  while(true)
  {
    // Get and check the status for each of the instances to see if it's past pending.
    // Once all instances are past pending, break out.
    // (For this example, we are assuming that there is only one reservation.)
    Console.Write(".");
```

```
        numberRunning = 0;
        responseDescribe = await ec2Client.DescribeInstancesAsync(requestDescribe);
        foreach(Instance i in responseDescribe.Reservations[0].Instances)
        {
          // Check the lower byte of State.Code property
          // Code == 0 is the pending state
          if((i.State.Code & 255) > 0) numberRunning++;
        }
        if(numberRunning == responseDescribe.Reservations[0].Instances.Count)
          break;

        // Wait a bit and try again (unless the user wants to stop waiting)
        Thread.Sleep(wait);
        if(Console.KeyAvailable)
          break;
      }

      Console.WriteLine("\nNo more instances are pending.");
      foreach(Instance i in responseDescribe.Reservations[0].Instances)
      {
        Console.WriteLine($"For {i.InstanceId}:");
        Console.WriteLine($"  VPC ID: {i.VpcId}");
        Console.WriteLine($"  Instance state: {i.State.Name}");
        Console.WriteLine($"  Public IP address: {i.PublicIpAddress}");
        Console.WriteLine($"  Public DNS name: {i.PublicDnsName}");
        Console.WriteLine($"  Key pair name: {i.KeyName}");
      }
    }


    //
    // Command-line help
    private static void PrintHelp()
    {
      Console.WriteLine(
        "\nUsage: EC2LaunchInstance -g <group-id> -a <ami-id> -k <keypair-name> [-s
<subnet-id>]" +
        "\n  -g, --group-id: The ID of the security group." +
        "\n  -a, --ami-id: The ID of an Amazon Machine Image." +
        "\n  -k, --keypair-name - The name of a key pair." +
        "\n  -s, --subnet-id: The ID of a subnet. Required only for EC2 in a VPC.");
    }
  }


  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
=
  // Class that represents a command line on the console or terminal.
  // (This is the same for all examples. When you have seen it once, you can ignore it.)
  static class CommandLine
  {
    //
    // Method to parse a command line of the form: "--key value" or "-k value".
    //
    // Parameters:
    // - args: The command-line arguments passed into the application by the system.
    //
    // Returns:
    // A Dictionary with string Keys and Values.
    //
    // If a key is found without a matching value, Dictionary.Value is set to the key
    //  (including the dashes).
    // If a value is found without a matching key, Dictionary.Key is set to "--NoKeyN",
    //  where "N" represents sequential numbers.
    public static Dictionary<string,string> Parse(string[] args)
    {
```

```
      var parsedArgs = new Dictionary<string,string>();
      int i = 0, n = 0;
      while(i < args.Length)
      {
        // If the first argument in this iteration starts with a dash it's an option.
        if(args[i].StartsWith("-"))
        {
          var key = args[i++];
          var value = key;

          // Check to see if there's a value that goes with this option?
          if((i < args.Length) && (!args[i].StartsWith("-"))) value = args[i++];
          parsedArgs.Add(key, value);
        }

        // If the first argument in this iteration doesn't start with a dash, it's a value
        else
        {
          parsedArgs.Add("--NoKey" + n.ToString(), args[i++]);
          n++;
        }
      }

      return parsedArgs;
    }

    //
    // Method to get an argument from the parsed command-line arguments
    //
    // Parameters:
    // - parsedArgs: The Dictionary object returned from the Parse() method (shown above).
    // - defaultValue: The default string to return if the specified key isn't in
 parsedArgs.
    // - keys: An array of keys to look for in parsedArgs.
    public static string GetArgument(
      Dictionary<string,string> parsedArgs, string defaultReturn, params string[] keys)
    {
      string retval = null;
      foreach(var key in keys)
        if(parsedArgs.TryGetValue(key, out retval)) break;
      return retval ?? defaultReturn;
    }

    //
    // Method to exit the application with an error.
    public static void ErrorExit(string msg, int code=1)
    {
      Console.WriteLine("\nError");
      Console.WriteLine(msg);
      Environment.Exit(code);
    }
  }

}
```

Additional considerations

- When checking the state of an EC2 instance, you can add a filter to the `Filter` property of the DescribeInstancesRequest object. Using this technique, you can limit the request to certain instances; for example, instances with a particular user-specified tag.

- For brevity, some properties were given typical values. Any or all of these properties can instead be determined programmatically or by user input.

- The values you can use for the `MinCount` and `MaxCount` properties of the RunInstancesRequest object are determined by the target Availability Zone and the maximum number of instances you're allowed for the instance type. For more information, see How many instances can I run in Amazon EC2 in the Amazon EC2 General FAQ.

- If you want to use a different instance type than this example, there are several instance types to choose from, which you can see in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances.

- You can also attach an IAM role (p. 183) to an instance when you launch it. To do so, create an IamInstanceProfileSpecification object whose `Name` property is set to the name of an IAM role. Then add that object to the `IamInstanceProfile` property of the RunInstancesRequest object.

  Note
  To launch an EC2 instance that has an IAM role attached, an IAM user's configuration must include certain permissions. For more information about the required permissions, see the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances.

(optional) Connect to the instance

After an instance is running, you can connect to it remotely by using the appropriate remote client. For both Linux and Windows instances, you need the instance's public IP address or public DNS name. You also need the following.

For Linux instances

You can use an SSH client to connect to your Linux instance. Make sure that the security group you used when you launched the instance allows SSH traffic on port 22, as described in Updating security groups (p. 134).

You also need the private portion of the key pair you used to launch the instance; that is, the PEM file.

For more information, see Connect to your Linux instance in the Amazon EC2 User Guide for Linux Instances.

For Windows instances

You can use an RDP client to connect to your instance. Make sure that the security group you used when you launched the instance allows RDP traffic on port 3389, as described in Updating security groups (p. 134).

You also need the Administrator password. You can obtain this by using the following example code, which requires the instance ID and the private portion of the key pair used to launch the instance; that is, the PEM file.

For more information, see Connecting to your Windows instance in the Amazon EC2 User Guide for Windows Instances.

Warning
This example code returns the plaintext Administrator password for your instance.

SDK references

NuGet packages:

- AWSSDK.EC2

Programming elements:

- Namespace Amazon.EC2

  Class AmazonEC2Client
- Namespace Amazon.EC2.Model

  Class GetPasswordDataRequest

  Class GetPasswordDataResponse

The code

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;
using Amazon.EC2;
using Amazon.EC2.Model;

namespace EC2GetWindowsPassword
{
  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 =
  // Class to get the Administrator password of a Windows EC2 instance
  class Program
  {
    static async Task Main(string[] args)
    {
      // Parse the command line and show help if necessary
      var parsedArgs = CommandLine.Parse(args);
      if(parsedArgs.Count == 0)
      {
        PrintHelp();
        return;
      }

      // Get the application arguments from the parsed list
      string instanceID =
        CommandLine.GetArgument(parsedArgs, null, "-i", "--instance-id");
      string pemFileName =
        CommandLine.GetArgument(parsedArgs, null, "-p", "--pem-filename");
      if(   (string.IsNullOrEmpty(instanceID) || !instanceID.StartsWith("i-"))
         || (string.IsNullOrEmpty(pemFileName) || !pemFileName.EndsWith(".pem")))
        CommandLine.ErrorExit(
          "\nOne or more of the required arguments is missing or incorrect." +
          "\nRun the command with no arguments to see help.");

      // Create the EC2 client
      var ec2Client = new AmazonEC2Client();

      // Get and display the password
      string password = await GetPassword(ec2Client, instanceID, pemFileName);
      Console.WriteLine($"\nPassword: {password}");
    }


    //
    // Method to get the administrator password of a Windows EC2 instance
    private static async Task<string> GetPassword(
      IAmazonEC2 ec2Client, string instanceID, string pemFilename)
    {
      string password = string.Empty;
```

```
      GetPasswordDataResponse response =
        await ec2Client.GetPasswordDataAsync(new GetPasswordDataRequest{
          InstanceId = instanceID});
      if(response.PasswordData != null)
      {
        password = response.GetDecryptedPassword(File.ReadAllText(pemFilename));
      }
      else
      {
        Console.WriteLine($"\nThe password is not available for instance {instanceID}.");
        Console.WriteLine($"If this is a Windows instance, the password might not be
ready.");
      }
      return password;
    }


    //
    // Command-line help
    private static void PrintHelp()
    {
      Console.WriteLine(
        "\nUsage: EC2GetWindowsPassword -i <instance-id> -p pem-filename" +
        "\n  -i, --instance-id: The name of the EC2 instance." +
        "\n  -p, --pem-filename: The name of the PEM file with the private key.");
    }
  }

  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
=
  // Class that represents a command line on the console or terminal.
  // (This is the same for all examples. When you have seen it once, you can ignore it.)
  static class CommandLine
  {
    //
    // Method to parse a command line of the form: "--key value" or "-k value".
    //
    // Parameters:
    // - args: The command-line arguments passed into the application by the system.
    //
    // Returns:
    // A Dictionary with string Keys and Values.
    //
    // If a key is found without a matching value, Dictionary.Value is set to the key
    //  (including the dashes).
    // If a value is found without a matching key, Dictionary.Key is set to "--NoKeyN",
    //  where "N" represents sequential numbers.
    public static Dictionary<string,string> Parse(string[] args)
    {
      var parsedArgs = new Dictionary<string,string>();
      int i = 0, n = 0;
      while(i < args.Length)
      {
        // If the first argument in this iteration starts with a dash it's an option.
        if(args[i].StartsWith("-"))
        {
          var key = args[i++];
          var value = key;

          // Check to see if there's a value that goes with this option?
          if((i < args.Length) && (!args[i].StartsWith("-"))) value = args[i++];
          parsedArgs.Add(key, value);
        }

        // If the first argument in this iteration doesn't start with a dash, it's a value
        else
```

```
        {
          parsedArgs.Add("--NoKey" + n.ToString(), args[i++]);
          n++;
        }
      }

      return parsedArgs;
    }

    //
    // Method to get an argument from the parsed command-line arguments
    //
    // Parameters:
    // - parsedArgs: The Dictionary object returned from the Parse() method (shown above).
    // - defaultValue: The default string to return if the specified key isn't in
 parsedArgs.
    // - keys: An array of keys to look for in parsedArgs.
    public static string GetArgument(
      Dictionary<string,string> parsedArgs, string defaultReturn, params string[] keys)
    {
      string retval = null;
      foreach(var key in keys)
        if(parsedArgs.TryGetValue(key, out retval)) break;
      return retval ?? defaultReturn;
    }

    //
    // Method to exit the application with an error.
    public static void ErrorExit(string msg, int code=1)
    {
      Console.WriteLine("\nError");
      Console.WriteLine(msg);
      Environment.Exit(code);
    }
  }

}
```

Clean up

When you no longer need your EC2 instance, be sure to terminate it, as described in Terminating an Amazon EC2 instance (p. 157).

Terminating an Amazon EC2 instance

When you no longer need one or more of your Amazon EC2 instances, you can terminate them.

This example shows you how to use the AWS SDK for .NET to terminate EC2 instances. It takes an instance ID as input.

SDK references

NuGet packages:

- AWSSDK.EC2

Programming elements:

- Namespace Amazon.EC2

  Class AmazonEC2Client
- Namespace Amazon.EC2.Model

Class TerminateInstancesRequest

Class TerminateInstancesResponse

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
using Amazon.EC2;
using Amazon.EC2.Model;

namespace EC2TerminateInstance
{
  class Program
  {
    static async Task Main(string[] args)
    {
      if((args.Length == 1) && (args[0].StartsWith("i-")))
      {
        // Terminate the instance
        var ec2Client = new AmazonEC2Client();
        await TerminateInstance(ec2Client, args[0]);
      }
      else
      {
        Console.WriteLine("\nCommand-line argument missing or incorrect.");
        Console.WriteLine("\nUsage: EC2TerminateInstance instance-ID");
        Console.WriteLine("  instance-ID - The EC2 instance you want to terminate.");
        return;
      }
    }

    //
    // Method to terminate an EC2 instance
    private static async Task TerminateInstance(IAmazonEC2 ec2Client, string instanceID)
    {
      var request = new TerminateInstancesRequest{
        InstanceIds = new List<string>() { instanceID }};
      TerminateInstancesResponse response =
        await ec2Client.TerminateInstancesAsync(new TerminateInstancesRequest{
          InstanceIds = new List<string>() { instanceID }
        });
      foreach (InstanceStateChange item in response.TerminatingInstances)
      {
        Console.WriteLine("Terminated instance: " + item.InstanceId);
        Console.WriteLine("Instance state: " + item.CurrentState.Name);
      }
    }
  }
}
```

After you run the example, it's a good idea to sign in to the Amazon EC2 console to verify that the EC2 instance has been terminated.

Amazon EC2 Spot Instance tutorial

This tutorial shows you how to use the AWS SDK for .NET to manage Amazon EC2 Spot Instances.

Overview

Spot Instances enable you to request unused Amazon EC2 capacity for less than the On-Demand price. This can significantly lower your EC2 costs for applications that can be interrupted.

The following is a high-level summary of how Spot Instances are requested and used.

1. Create a Spot Instance request, specifying the maximum price you are willing to pay.
2. When the request is fulfilled, run the instance as you would any other Amazon EC2 instance.
3. Run the instance as long as you want and then terminate it, unless the *Spot Price* changes such that the instance is terminated for you.
4. Clean up the Spot Instance request when you no longer need it so that Spot Instances are no longer created.

This has been a very high level overview of Spot Instances. You can gain a better understanding of Spot Instances by reading about them in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances.

About this tutorial

As you follow this tutorial, you use the AWS SDK for .NET to do the following:

- Create a Spot Instance request
- Determine when the Spot Instance request has been fulfilled
- Cancel the Spot Instance request
- Terminate associated instances

The following sections provide snippets and other information for this example. The complete code for the example (p. 163) is shown after the snippets, and can be built and run as is.

Topics

Prerequisites

For information about the APIs and prerequisites, see the parent section (Working with Amazon EC2 (p. 125)).

Gather what you need

To create a Spot Instance request, you'll need several things.

- The number of instances and their instance type. There are several instance types to choose from, which you can see in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances. The default number for this tutorial is 1.

- The Amazon Machine Image (AMI) that will be used to create the instance. See the information about AMIs in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows

Instances. For example, read about shared AMIs in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances.

- The maximum price that you're willing to pay per instance hour. You can see the prices for all instance types (for both On-Demand Instances and Spot Instances) on the Amazon EC2 pricing page. The default price for this tutorial is explained later.

- If you want to connect remotely to an instance, a security group with the appropriate configuration and resources. This is described in Working with security groups in Amazon EC2 (p. 125) and the information about gathering what you need (p. 147) and connecting to an instance (p. 154) in Launching an Amazon EC2 instance (p. 146). For simplicity, this tutorial uses the security group named **default** that all newer AWS accounts have.

There are many ways to approach requesting Spot Instances. The following are common strategies:

- Make requests that are sure to cost less than on-demand pricing.
- Make requests based on the value of the resulting computation.
- Make requests so as to acquire computing capacity as quickly as possible.

The following explanations refer to the Spot Price history in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances.

Reduce cost below On-Demand

You have a batch processing job that will take a number of hours or days to run. However, you are flexible with respect to when it starts and ends. You want to see if you can complete it for less than the cost of On-Demand Instances.

You examine the Spot Price history for instance types by using either the Amazon EC2 console or the Amazon EC2 API. After you've analyzed the price history for your desired instance type in a given Availability Zone, you have two alternative approaches for your request:

- Specify a request at the upper end of the range of Spot Prices, which are still below the On-Demand price, anticipating that your one-time Spot Instance request would most likely be fulfilled and run for enough consecutive compute time to complete the job.
- Specify a request at the lower end of the price range, and plan to combine many instances launched over time through a persistent request. The instances would run long enough, in aggregate, to complete the job at an even lower total cost.

Pay no more than the value of the result

You have a data processing job to run. You understand the value of the job's results well enough to know how much they're worth in terms of computing costs.

After you've analyzed the Spot Price history for your instance type, you choose a price at which the cost of the computing time is no more than the value of the job's results. You create a persistent request and allow it to run intermittently as the Spot Price fluctuates at or below your request.

Acquire computing capacity quickly

You have an unanticipated, short-term need for additional capacity that's not available through On-Demand Instances. After you've analyzed the Spot Price history for your instance type, you choose a price above the highest historical price to greatly improve the likelihood that your request will be fulfilled quickly and continue computing until it's complete.

After you have gathered what you need and chosen a strategy, you are ready to request a Spot Instance. For this tutorial the default maximum spot-instance price is set to be the same as the On-Demand price (which is $0.003 for this tutorial). Setting the price in this way maximizes the chances that the request will be fulfilled.

Creating a Spot Instance request

The following snippet shows you how to create a Spot Instance request with the elements you gathered earlier.

The example at the end of this topic (p. 163) shows this snippet in use.

```
//
// Method to create a Spot Instance request
private static async Task<SpotInstanceRequest> CreateSpotInstanceRequest(
  IAmazonEC2 ec2Client, string amiId, string securityGroupName,
  InstanceType instanceType, string spotPrice, int instanceCount)
{
  var launchSpecification = new LaunchSpecification{
    ImageId = amiId,
    InstanceType = instanceType
  };
  launchSpecification.SecurityGroups.Add(securityGroupName);
  var request = new RequestSpotInstancesRequest{
    SpotPrice = spotPrice,
    InstanceCount = instanceCount,
    LaunchSpecification = launchSpecification
  };

  RequestSpotInstancesResponse result =
    await ec2Client.RequestSpotInstancesAsync(request);
  return result.SpotInstanceRequests[0];
}
```

The important value returned from this method is the Spot Instance request ID, which is contained in the `SpotInstanceRequestId` member of the returned SpotInstanceRequest object.

Note
You will be charged for any Spot Instances that are launched. To avoid unnecessary costs be sure to cancel any requests (p. 162) and terminate any instances (p. 162).

Determine the state of your Spot Instance request

The following snippet shows you how to get information about your Spot Instance request. You can use that information to make certain decisions in your code, such as whether to continue waiting for a Spot Instance request to be fulfilled.

The example at the end of this topic (p. 163) shows this snippet in use.

```
//
// Method to get information about a Spot Instance request, including the status,
// instance ID, etc.
// It gets the information for a specific request (as opposed to all requests).
private static async Task<SpotInstanceRequest> GetSpotInstanceRequestInfo(
  IAmazonEC2 ec2Client, string requestId)
{
  var describeRequest = new DescribeSpotInstanceRequestsRequest();
  describeRequest.SpotInstanceRequestIds.Add(requestId);

  DescribeSpotInstanceRequestsResponse describeResponse =
    await ec2Client.DescribeSpotInstanceRequestsAsync(describeRequest);
```

```
    return describeResponse.SpotInstanceRequests[0];
}
```

The method returns information about the Spot Instance request such as the instance ID, it's state, and the status code. You can see the status codes for Spot Instance requests in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances.

Clean up your Spot Instance requests

When you no longer need to request Spot Instances, it's important to cancel any outstanding requests to prevent those requests from being re-fulfilled. The following snippet shows you how to cancel a Spot Instance request.

The example at the end of this topic (p. 163) shows this snippet in use.

```
//
// Method to cancel a Spot Instance request
private static async Task CancelSpotInstanceRequest(
  IAmazonEC2 ec2Client, string requestId)
{
  var cancelRequest = new CancelSpotInstanceRequestsRequest();
  cancelRequest.SpotInstanceRequestIds.Add(requestId);

  await ec2Client.CancelSpotInstanceRequestsAsync(cancelRequest);
}
```

Clean up your Spot Instances

To avoid unnecessary costs, it's important that you terminate any instances that were started from Spot Instance requests; simply canceling Spot Instance requests will not terminate your instances, which means that you'll continue to be charged for them. The following snippet shows you how to terminate an instance after you obtain the instance identifier for an active Spot Instance.

The example at the end of this topic (p. 163) shows this snippet in use.

```
//
// Method to terminate a Spot Instance
private static async Task TerminateSpotInstance(
  IAmazonEC2 ec2Client, string requestId)
{
  var describeRequest = new DescribeSpotInstanceRequestsRequest();
  describeRequest.SpotInstanceRequestIds.Add(requestId);

  // Retrieve the Spot Instance request to check for running instances.
  DescribeSpotInstanceRequestsResponse describeResponse =
    await ec2Client.DescribeSpotInstanceRequestsAsync(describeRequest);

  // If there are any running instances, terminate them
  if(   (describeResponse.SpotInstanceRequests[0].Status.Code
          == "request-canceled-and-instance-running")
     || (describeResponse.SpotInstanceRequests[0].State == SpotInstanceState.Active))
  {
    TerminateInstancesResponse response =
      await ec2Client.TerminateInstancesAsync(new TerminateInstancesRequest{
        InstanceIds = new List<string>(){
          describeResponse.SpotInstanceRequests[0].InstanceId } });
    foreach (InstanceStateChange item in response.TerminatingInstances)
    {
      Console.WriteLine($"\n  Terminated instance: {item.InstanceId}");
      Console.WriteLine($"  Instance state: {item.CurrentState.Name}\n");
```

```
        }
      }
    }
```

Complete code

The following code example calls the methods described earlier to create and cancel a Spot Instance request and terminate a Spot Instance.

SDK references

NuGet packages:

- AWSSDK.EC2

Programming elements:

- Namespace Amazon.EC2

  Class AmazonEC2Client

  Class InstanceType

- Namespace Amazon.EC2.Model

  Class CancelSpotInstanceRequestsRequest

  Class DescribeSpotInstanceRequestsRequest

  Class DescribeSpotInstanceRequestsResponse

  Class InstanceStateChange

  Class LaunchSpecification

  Class RequestSpotInstancesRequest

  Class RequestSpotInstancesResponse

  Class SpotInstanceRequest

  Class TerminateInstancesRequest

  Class TerminateInstancesResponse

The code

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Collections.Generic;
using Amazon.EC2;
using Amazon.EC2.Model;

namespace EC2SpotInstanceRequests
{
  class Program
  {
    static async Task Main(string[] args)
```

```
    {
      // Some default values.
      // These could be made into command-line arguments instead.
      var instanceType = InstanceType.T1Micro;
      string securityGroupName = "default";
      string spotPrice = "0.003";
      int instanceCount = 1;

      // Parse the command line arguments
      if((args.Length != 1) || (!args[0].StartsWith("ami-")))
      {
        Console.WriteLine("\nUsage: EC2SpotInstanceRequests ami");
        Console.WriteLine("  ami: the Amazon Machine Image to use for the Spot
Instances.");
        return;
      }

      // Create the Amazon EC2 client.
      var ec2Client = new AmazonEC2Client();

      // Create the Spot Instance request and record its ID
      Console.WriteLine("\nCreating spot instance request...");
      var req = await CreateSpotInstanceRequest(
        ec2Client, args[0], securityGroupName, instanceType, spotPrice, instanceCount);
      string requestId = req.SpotInstanceRequestId;

      // Wait for an EC2 Spot Instance to become active
      Console.WriteLine(
        $"Waiting for Spot Instance request with ID {requestId} to become active...");
      int wait = 1;
      var start = DateTime.Now;
      while(true)
      {
        Console.Write(".");

        // Get and check the status to see if the request has been fulfilled.
        var requestInfo = await GetSpotInstanceRequestInfo(ec2Client, requestId);
        if(requestInfo.Status.Code == "fulfilled")
        {
          Console.WriteLine($"\nSpot Instance request {requestId} " +
            $"has been fulfilled by instance {requestInfo.InstanceId}.\n");
          break;
        }

        // Wait a bit and try again, longer each time (1, 2, 4, ...)
        Thread.Sleep(wait);
        wait = wait * 2;
      }

      // Show the user how long it took to fulfill the Spot Instance request.
      TimeSpan span = DateTime.Now.Subtract(start);
      Console.WriteLine($"That took {span.TotalMilliseconds} milliseconds");

      // Perform actions here as needed.
      // For this example, simply wait for the user to hit a key.
      // That gives them a chance to look at the EC2 console to see
      // the running instance if they want to.
      Console.WriteLine("Press any key to start the cleanup...");
      Console.ReadKey(true);

      // Cancel the request.
      // Do this first to make sure that the request can't be re-fulfilled
      // once the Spot Instance has been terminated.
      Console.WriteLine("Canceling Spot Instance request...");
      await CancelSpotInstanceRequest(ec2Client, requestId);
```

```
  // Terminate the Spot Instance that's running.
  Console.WriteLine("Terminating the running Spot Instance...");
  await TerminateSpotInstance(ec2Client, requestId);

  Console.WriteLine("Done. Press any key to exit...");
  Console.ReadKey(true);
}


//
// Method to create a Spot Instance request
private static async Task<SpotInstanceRequest> CreateSpotInstanceRequest(
  IAmazonEC2 ec2Client, string amiId, string securityGroupName,
  InstanceType instanceType, string spotPrice, int instanceCount)
{
  var launchSpecification = new LaunchSpecification{
    ImageId = amiId,
    InstanceType = instanceType
  };
  launchSpecification.SecurityGroups.Add(securityGroupName);
  var request = new RequestSpotInstancesRequest{
    SpotPrice = spotPrice,
    InstanceCount = instanceCount,
    LaunchSpecification = launchSpecification
  };

  RequestSpotInstancesResponse result =
    await ec2Client.RequestSpotInstancesAsync(request);
  return result.SpotInstanceRequests[0];
}


//
// Method to get information about a Spot Instance request, including the status,
// instance ID, etc.
// It gets the information for a specific request (as opposed to all requests).
private static async Task<SpotInstanceRequest> GetSpotInstanceRequestInfo(
  IAmazonEC2 ec2Client, string requestId)
{
  var describeRequest = new DescribeSpotInstanceRequestsRequest();
  describeRequest.SpotInstanceRequestIds.Add(requestId);

  DescribeSpotInstanceRequestsResponse describeResponse =
    await ec2Client.DescribeSpotInstanceRequestsAsync(describeRequest);
  return describeResponse.SpotInstanceRequests[0];
}


//
// Method to cancel a Spot Instance request
private static async Task CancelSpotInstanceRequest(
  IAmazonEC2 ec2Client, string requestId)
{
  var cancelRequest = new CancelSpotInstanceRequestsRequest();
  cancelRequest.SpotInstanceRequestIds.Add(requestId);

  await ec2Client.CancelSpotInstanceRequestsAsync(cancelRequest);
}


//
// Method to terminate a Spot Instance
private static async Task TerminateSpotInstance(
  IAmazonEC2 ec2Client, string requestId)
{
  var describeRequest = new DescribeSpotInstanceRequestsRequest();
```

```
      describeRequest.SpotInstanceRequestIds.Add(requestId);

      // Retrieve the Spot Instance request to check for running instances.
      DescribeSpotInstanceRequestsResponse describeResponse =
        await ec2Client.DescribeSpotInstanceRequestsAsync(describeRequest);

      // If there are any running instances, terminate them
      if(   (describeResponse.SpotInstanceRequests[0].Status.Code
              == "request-canceled-and-instance-running")
         || (describeResponse.SpotInstanceRequests[0].State == SpotInstanceState.Active))
      {
        TerminateInstancesResponse response =
          await ec2Client.TerminateInstancesAsync(new TerminateInstancesRequest{
            InstanceIds = new List<string>(){
              describeResponse.SpotInstanceRequests[0].InstanceId } });
        foreach (InstanceStateChange item in response.TerminatingInstances)
        {
          Console.WriteLine($"\n  Terminated instance: {item.InstanceId}");
          Console.WriteLine($"  Instance state: {item.CurrentState.Name}\n");
        }
      }
    }
  }
}
```

Additional considerations

- After you run the tutorial, it's a good idea to sign in to the Amazon EC2 console to verify that the Spot Instance request has been canceled and that the Spot Instance has been terminated.

Accessing AWS Identity and Access Management (IAM) with the AWS SDK for .NET

The AWS SDK for .NET supports AWS Identity and Access Management, which is a web service that enables AWS customers to manage users and user permissions in AWS.

An AWS Identity and Access Management (IAM) *user* is an entity that you create in AWS. The entity represents a person or application that interacts with AWS. For more information about IAM users, see IAM Users and IAM and STS Limits in the IAM User Guide.

You grant permissions to a user by creating a IAM *policy*. The policy contains a *policy document* that lists the actions that a user can perform and the resources those actions can affect. For more information about IAM policies, see Policies and Permissions in the *IAM User Guide*.

APIs

The AWS SDK for .NET provides APIs for IAM clients. The APIs enable you to work with IAM features such as users, roles, and access keys.

This section contains a small number of examples that show you the patterns you can follow when working with these APIs. To view the full set of APIs, see the AWS SDK for .NET API Reference (and scroll to "Amazon.IdentityManagement").

This section also contains an example (p. 183) that shows you how to attach an IAM role to Amazon EC2 instances to make managing credentials easier.

The IAM APIs are provided by the AWSSDK.IdentityManagement NuGet package.

Prerequisites

Before you begin, be sure you have set up your environment (p. 15). Also review the information in Setting up your project (p. 17) and SDK features (p. 49).

Topics

Topics

Creating and listing users for your AWS account

This example shows you how to use the AWS SDK for .NET to create a new IAM user. With the information you supply to the application, it creates a user, attaches the given managed policy, obtains credentials for the user, and then displays a list of all the users in your AWS account.

In you don't supply any command-line arguments, the application simply displays a list of all the users in your AWS account.

One of the inputs you supply is the Amazon Resource Name (ARN) for an existing managed policy. You can find the available policies and their ARNs in the IAM console.

The following sections provide snippets of this example. The complete code for the example (p. 169) is shown after that, and can be built and run as is.

Topics

Create a user

The following snippet creates an IAM user, adds the given managed security policy, and then creates and stores credentials for the user.

The example at the end of this topic (p. 169) shows this snippet in use.

```
//
// Method to create the user
private static async Task<CreateUserResponse> CreateUser(
  IAmazonIdentityManagementService iamClient, string userName,
  string policyArn, string csvFilename)
{
  // Create the user
  // Could also create a login profile for the user by using CreateLoginProfileAsync
  CreateUserResponse responseCreate =
    await iamClient.CreateUserAsync(new CreateUserRequest(userName));
```

```
  // Attach an existing managed policy
  await iamClient.AttachUserPolicyAsync(new AttachUserPolicyRequest{
    UserName = responseCreate.User.UserName,
    PolicyArn = policyArn});

  // Create credentials and write them to a CSV file.
  CreateAccessKeyResponse responseCreds =
    await iamClient.CreateAccessKeyAsync(new CreateAccessKeyRequest{
      UserName = responseCreate.User.UserName});
  using (FileStream s = new FileStream(csvFilename, FileMode.Create))
  using (StreamWriter writer = new StreamWriter(s))
  {
    writer.WriteLine("User name,Access key ID,Secret access key");
    writer.WriteLine("{0},{1},{2}", responseCreds.AccessKey.UserName,
      responseCreds.AccessKey.AccessKeyId,
      responseCreds.AccessKey.SecretAccessKey);
  }

  return responseCreate;
}
```

Display a list of users

The following snippet displays a list of existing users, as well as information about each user such as access key IDs and attached policies.

The example at the end of this topic (p. 169) shows this snippet in use.

```
//
// Method to print out a list of the existing users and information about them
private static async Task ListUsers(IAmazonIdentityManagementService iamClient)
{
  // Get the list of users
  ListUsersResponse responseUsers = await iamClient.ListUsersAsync();
  Console.WriteLine("\nFull list of users...");
  foreach (var user in responseUsers.Users)
  {
    Console.WriteLine($"User {user.UserName}:");
    Console.WriteLine($"\tCreated: {user.CreateDate.ToShortDateString()}");

    // Show the list of groups this user is part of
    ListGroupsForUserResponse responseGroups =
      await iamClient.ListGroupsForUserAsync(
        new ListGroupsForUserRequest(user.UserName));
    foreach (var group in responseGroups.Groups)
      Console.WriteLine($"\tGroup: {group.GroupName}");

    // Show the list of access keys for this user
    ListAccessKeysResponse responseAccessKeys =
      await iamClient.ListAccessKeysAsync(
        new ListAccessKeysRequest{UserName = user.UserName});
    foreach(AccessKeyMetadata accessKey in responseAccessKeys.AccessKeyMetadata)
      Console.WriteLine($"\tAccess key ID: {accessKey.AccessKeyId}");

    // Show the list of managed policies attached to this user
    var requestManagedPolicies = new ListAttachedUserPoliciesRequest{
      UserName = user.UserName};
    ListAttachedUserPoliciesResponse responseManagedPolicies =
      await iamClient.ListAttachedUserPoliciesAsync(
        new ListAttachedUserPoliciesRequest{UserName = user.UserName});
    foreach(var policy in responseManagedPolicies.AttachedPolicies)
      Console.WriteLine($"\tManaged policy name: {policy.PolicyName}");
```

```
        // Show the list of inline policies attached to this user
        ListUserPoliciesResponse responseInlinePolicies =
          await iamClient.ListUserPoliciesAsync(
            new ListUserPoliciesRequest(user.UserName));
        foreach(var policy in responseInlinePolicies.PolicyNames)
          Console.WriteLine($"\tInline policy name: {policy}");
      }
    }
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.IdentityManagement

Programming elements:

- Namespace Amazon.IdentityManagement

  Class AmazonIdentityManagementServiceClient
- Namespace Amazon.IdentityManagement.Model

  Class AttachUserPolicyRequest

  Class CreateAccessKeyRequest

  Class CreateAccessKeyResponse

  Class CreateUserRequest

  Class CreateUserResponse

  Class ListAccessKeysRequest

  Class ListAccessKeysResponse

  Class ListAttachedUserPoliciesRequest

  Class ListAttachedUserPoliciesResponse

  Class ListGroupsForUserRequest

  Class ListGroupsForUserResponse

  Class ListUserPoliciesRequest

  Class ListUserPoliciesResponse

  Class ListUsersResponse

The code

```
using System;
using System.Collections.Generic;
using System.IO;
```

```
using System.Threading.Tasks;
using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

namespace IamCreateUser
{
  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 =
  // Class to create a user
  class Program
  {
    private const int MaxArgs = 3;

    static async Task Main(string[] args)
    {
      // Create an IAM service client
      var iamClient = new AmazonIdentityManagementServiceClient();

      // Parse the command line and show help if necessary
      var parsedArgs = CommandLine.Parse(args);
      if((parsedArgs.Count == 0) || (parsedArgs.Count > MaxArgs))
      {
        PrintHelp();
        Console.WriteLine("\nIncorrect number of arguments specified.");
        Console.Write("Do you want to see a list of the existing users? ((y) or n): ");
        string response = Console.ReadLine();
        if((string.IsNullOrEmpty(response)) || (response.ToLower() == "y"))
          await ListUsers(iamClient);
        return;
      }

      // Get the application arguments from the parsed list
      string userName =
        CommandLine.GetArgument(parsedArgs, null, "-u", "--user-name");
      string policyArn =
        CommandLine.GetArgument(parsedArgs, null, "-p", "--policy-arn");
      string csvFilename =
        CommandLine.GetArgument(parsedArgs, null, "-c", "--csv-filename");
      if(   (string.IsNullOrEmpty(policyArn) || !policyArn.StartsWith("arn:"))
         || (string.IsNullOrEmpty(csvFilename) || !csvFilename.EndsWith(".csv"))
         || (string.IsNullOrEmpty(userName)))
        CommandLine.ErrorExit(
          "\nOne or more of the required arguments is missing or incorrect." +
          "\nRun the command with no arguments to see help.");

      // Create a user, attach a managed policy, and obtain credentials
      var responseCreate =
        await CreateUser(iamClient, userName, policyArn, csvFilename);
      Console.WriteLine($"\nUser {responseCreate.User.UserName} was created.");
      Console.WriteLine($"User ID: {responseCreate.User.UserId}");

      // Output a list of the existing users
      await ListUsers(iamClient);
    }


    //
    // Method to create the user
    private static async Task<CreateUserResponse> CreateUser(
      IAmazonIdentityManagementService iamClient, string userName,
      string policyArn, string csvFilename)
    {
      // Create the user
      // Could also create a login profile for the user by using CreateLoginProfileAsync
      CreateUserResponse responseCreate =
        await iamClient.CreateUserAsync(new CreateUserRequest(userName));
```

```
  // Attach an existing managed policy
  await iamClient.AttachUserPolicyAsync(new AttachUserPolicyRequest{
    UserName = responseCreate.User.UserName,
    PolicyArn = policyArn});

  // Create credentials and write them to a CSV file.
  CreateAccessKeyResponse responseCreds =
    await iamClient.CreateAccessKeyAsync(new CreateAccessKeyRequest{
      UserName = responseCreate.User.UserName});
  using (FileStream s = new FileStream(csvFilename, FileMode.Create))
  using (StreamWriter writer = new StreamWriter(s))
  {
    writer.WriteLine("User name,Access key ID,Secret access key");
    writer.WriteLine("{0},{1},{2}", responseCreds.AccessKey.UserName,
      responseCreds.AccessKey.AccessKeyId,
      responseCreds.AccessKey.SecretAccessKey);
  }

  return responseCreate;
}


//
// Method to print out a list of the existing users and information about them
private static async Task ListUsers(IAmazonIdentityManagementService iamClient)
{
  // Get the list of users
  ListUsersResponse responseUsers = await iamClient.ListUsersAsync();
  Console.WriteLine("\nFull list of users...");
  foreach (var user in responseUsers.Users)
  {
    Console.WriteLine($"User {user.UserName}:");
    Console.WriteLine($"\tCreated: {user.CreateDate.ToShortDateString()}");

    // Show the list of groups this user is part of
    ListGroupsForUserResponse responseGroups =
      await iamClient.ListGroupsForUserAsync(
        new ListGroupsForUserRequest(user.UserName));
    foreach (var group in responseGroups.Groups)
      Console.WriteLine($"\tGroup: {group.GroupName}");

    // Show the list of access keys for this user
    ListAccessKeysResponse responseAccessKeys =
      await iamClient.ListAccessKeysAsync(
        new ListAccessKeysRequest{UserName = user.UserName});
    foreach(AccessKeyMetadata accessKey in responseAccessKeys.AccessKeyMetadata)
      Console.WriteLine($"\tAccess key ID: {accessKey.AccessKeyId}");

    // Show the list of managed policies attached to this user
    var requestManagedPolicies = new ListAttachedUserPoliciesRequest{
      UserName = user.UserName};
    ListAttachedUserPoliciesResponse responseManagedPolicies =
      await iamClient.ListAttachedUserPoliciesAsync(
        new ListAttachedUserPoliciesRequest{UserName = user.UserName});
    foreach(var policy in responseManagedPolicies.AttachedPolicies)
      Console.WriteLine($"\tManaged policy name: {policy.PolicyName}");

    // Show the list of inline policies attached to this user
    ListUserPoliciesResponse responseInlinePolicies =
      await iamClient.ListUserPoliciesAsync(
        new ListUserPoliciesRequest(user.UserName));
    foreach(var policy in responseInlinePolicies.PolicyNames)
      Console.WriteLine($"\tInline policy name: {policy}");
  }
}
```

```
  //
  // Command-line help
  private static void PrintHelp()
  {
    Console.WriteLine(
      "\nUsage: IamCreateUser -u <user-name> -p <policy-arn> -c <csv-filename>" +
      "\n  -u, --user-name: The name of the user you want to create." +
      "\n  -p, --policy-arn: The ARN of an existing managed policy." +
      "\n  -c, --csv-filename: The name of a .csv file to write the credentials to.");
  }
}


// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
=
// Class that represents a command line on the console or terminal.
// (This is the same for all examples. When you have seen it once, you can ignore it.)
static class CommandLine
{
  //
  // Method to parse a command line of the form: "--key value" or "-k value".
  //
  // Parameters:
  // - args: The command-line arguments passed into the application by the system.
  //
  // Returns:
  // A Dictionary with string Keys and Values.
  //
  // If a key is found without a matching value, Dictionary.Value is set to the key
  //  (including the dashes).
  // If a value is found without a matching key, Dictionary.Key is set to "--NoKeyN",
  //  where "N" represents sequential numbers.
  public static Dictionary<string,string> Parse(string[] args)
  {
    var parsedArgs = new Dictionary<string,string>();
    int i = 0, n = 0;
    while(i < args.Length)
    {
      // If the first argument in this iteration starts with a dash it's an option.
      if(args[i].StartsWith("-"))
      {
        var key = args[i++];
        var value = key;

        // Check to see if there's a value that goes with this option?
        if((i < args.Length) && (!args[i].StartsWith("-"))) value = args[i++];
        parsedArgs.Add(key, value);
      }

      // If the first argument in this iteration doesn't start with a dash, it's a value
      else
      {
        parsedArgs.Add("--NoKey" + n.ToString(), args[i++]);
        n++;
      }
    }

    return parsedArgs;
  }

  //
  // Method to get an argument from the parsed command-line arguments
  //
  // Parameters:
```

```
    // - parsedArgs: The Dictionary object returned from the Parse() method (shown above).
    // - defaultValue: The default string to return if the specified key isn't in
 parsedArgs.
    // - keys: An array of keys to look for in parsedArgs.
    public static string GetArgument(
      Dictionary<string,string> parsedArgs, string defaultReturn, params string[] keys)
    {
      string retval = null;
      foreach(var key in keys)
        if(parsedArgs.TryGetValue(key, out retval)) break;
      return retval ?? defaultReturn;
    }

    //
    // Method to exit the application with an error.
    public static void ErrorExit(string msg, int code=1)
    {
      Console.WriteLine("\nError");
      Console.WriteLine(msg);
      Environment.Exit(code);
    }
  }

}
```

Additional considerations

- You can also see the list of users and the results of this example in the IAM console.

Deleting IAM users

This example shows you how use the AWS SDK for .NET to delete an IAM user. It first removes resources such as access keys, attached policies, etc., and then deletes the user.

The following sections provide snippets of this example. The complete code for the example (p. 174) is shown after that, and can be built and run as is.

Topics

Remove items from the user

The following snippets show examples of items that must be removed from a user before the user can be deleted, items such as managed policies and access keys.

The example at the end of this topic (p. 174) shows this snippet in use.

```
    //
    // Method to detach managed policies from a user
    private static async Task DetachPolicies(
      IAmazonIdentityManagementService iamClient, string userName)
    {
      ListAttachedUserPoliciesResponse responseManagedPolicies =
        await iamClient.ListAttachedUserPoliciesAsync(
```

```
        new ListAttachedUserPoliciesRequest{UserName = userName});
    foreach(AttachedPolicyType policy in responseManagedPolicies.AttachedPolicies)
    {
      Console.WriteLine($"\tDetaching policy {policy.PolicyName}");
      await iamClient.DetachUserPolicyAsync(new DetachUserPolicyRequest{
        PolicyArn = policy.PolicyArn,
        UserName = userName});
    }
  }


  //
  // Method to delete access keys from a user
  private static async Task DeleteAccessKeys(
    IAmazonIdentityManagementService iamClient, string userName)
  {
    ListAccessKeysResponse responseAccessKeys =
      await iamClient.ListAccessKeysAsync(
        new ListAccessKeysRequest{UserName = userName});
    foreach(AccessKeyMetadata accessKey in responseAccessKeys.AccessKeyMetadata)
    {
      Console.WriteLine($"\tDeleting Access key {accessKey.AccessKeyId}");
      await iamClient.DeleteAccessKeyAsync(new DeleteAccessKeyRequest{
        UserName = userName,
        AccessKeyId = accessKey.AccessKeyId});
    }
  }
```

Delete the user

The following snippet calls methods to remove items from a user and then deletes the user.

The example at the end of this topic (p. 174) shows this snippet in use.

```
  //
  // Method to delete a user
  private static async Task DeleteUser(
    IAmazonIdentityManagementService iamClient, string userName)
  {
    Console.WriteLine($"\nDeleting user {userName}...");
    //
    // Remove items from the user
    //
    // Detach any managed policies
    await DetachPolicies(iamClient, userName);

    // Delete any access keys
    await DeleteAccessKeys(iamClient, userName);

    // DeleteLoginProfileAsycn(), DeleteUserPolicyAsync(), etc.
    // See the description of DeleteUserAsync for a full list.

    //
    // Delete the user
    //
    await iamClient.DeleteUserAsync(new DeleteUserRequest(userName));
    Console.WriteLine("Done");
  }
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.IdentityManagement

Programming elements:

- Namespace Amazon.IdentityManagement

  Class AmazonIdentityManagementServiceClient
- Namespace Amazon.IdentityManagement.Model

  Class AccessKeyMetadata

  Class AttachedPolicyType

  Class DeleteAccessKeyRequest

  Class DeleteUserRequest

  Class DetachUserPolicyRequest

  Class ListAccessKeysRequest

  Class ListAccessKeysResponse

  Class ListAttachedUserPoliciesRequest

  Class ListAttachedUserPoliciesResponse

The code

```
using System;
using System.Threading.Tasks;
using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

namespace IamDeleteUser
{
  class Program
  {
    static async Task Main(string[] args)
    {
      if(args.Length != 1)
      {
        Console.WriteLine("\nUsage: IamDeleteUser user-name");
        Console.WriteLine("   user-name - The name of the user you want to delete.");
        return;
      }

      // Create an IAM service client
      var iamClient = new AmazonIdentityManagementServiceClient();

      // Delete the given user
      await DeleteUser(iamClient, args[0]);

      // Could display a list of the users that are left.
    }

    //
```

```
    // Method to delete a user
    private static async Task DeleteUser(
      IAmazonIdentityManagementService iamClient, string userName)
    {
      Console.WriteLine($"\nDeleting user {userName}...");
      //
      // Remove items from the user
      //
      // Detach any managed policies
      await DetachPolicies(iamClient, userName);

      // Delete any access keys
      await DeleteAccessKeys(iamClient, userName);

      // DeleteLoginProfileAsycn(), DeleteUserPolicyAsync(), etc.
      // See the description of DeleteUserAsync for a full list.

      //
      // Delete the user
      //
      await iamClient.DeleteUserAsync(new DeleteUserRequest(userName));
      Console.WriteLine("Done");
    }


    //
    // Method to detach managed policies from a user
    private static async Task DetachPolicies(
      IAmazonIdentityManagementService iamClient, string userName)
    {
      ListAttachedUserPoliciesResponse responseManagedPolicies =
        await iamClient.ListAttachedUserPoliciesAsync(
          new ListAttachedUserPoliciesRequest{UserName = userName});
      foreach(AttachedPolicyType policy in responseManagedPolicies.AttachedPolicies)
      {
        Console.WriteLine($"\tDetaching policy {policy.PolicyName}");
        await iamClient.DetachUserPolicyAsync(new DetachUserPolicyRequest{
          PolicyArn = policy.PolicyArn,
          UserName = userName});
      }
    }


    //
    // Method to delete access keys from a user
    private static async Task DeleteAccessKeys(
      IAmazonIdentityManagementService iamClient, string userName)
    {
      ListAccessKeysResponse responseAccessKeys =
        await iamClient.ListAccessKeysAsync(
          new ListAccessKeysRequest{UserName = userName});
      foreach(AccessKeyMetadata accessKey in responseAccessKeys.AccessKeyMetadata)
      {
        Console.WriteLine($"\tDeleting Access key {accessKey.AccessKeyId}");
        await iamClient.DeleteAccessKeyAsync(new DeleteAccessKeyRequest{
          UserName = userName,
          AccessKeyId = accessKey.AccessKeyId});
      }
    }
  }
}
```

Additional considerations

- For information about the resources that must be removed from the user, see the description of the DeleteUserAsync method, but be sure to use the Async versions of the referenced methods.

- You can also see the list of users and the results of this example in the IAM console.

Creating IAM managed policies from JSON

This example shows you how to use the AWS SDK for .NET to create an IAM managed policy from a given policy document in JSON. The application creates an IAM client object, reads the policy document from a file, and then creates the policy.

> **Note**
> For an example policy document in JSON, see the additional considerations (p. 180) at the end of this topic.

The following sections provide snippets of this example. The complete code for the example (p. 177) is shown after that, and can be built and run as is.

Topics

Create the policy

The following snippet creates an IAM managed policy with the given name and policy document.

The example at the end of this topic (p. 177) shows this snippet in use.

```
//
// Method to create an IAM policy from a JSON file
private static async Task<CreatePolicyResponse> CreateManagedPolicy(
  IAmazonIdentityManagementService iamClient, string policyName, string jsonFilename)
{
  return await iamClient.CreatePolicyAsync(new CreatePolicyRequest{
    PolicyName = policyName,
    PolicyDocument = File.ReadAllText(jsonFilename)});
}
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.IdentityManagement

Programming elements:

- Namespace Amazon.IdentityManagement

  Class AmazonIdentityManagementServiceClient

- Namespace Amazon.IdentityManagement.Model

  Class CreatePolicyRequest

  Class CreatePolicyResponse

The code

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;
using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

namespace IamCreatePolicyFromJson
{
  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 =
  // Class to create an IAM policy with a given policy document
  class Program
  {
    private const int MaxArgs = 2;

    static async Task Main(string[] args)
    {
      // Parse the command line and show help if necessary
      var parsedArgs = CommandLine.Parse(args);
      if((parsedArgs.Count == 0) || (parsedArgs.Count > MaxArgs))
      {
        PrintHelp();
        return;
      }

      // Get the application arguments from the parsed list
      string policyName =
        CommandLine.GetArgument(parsedArgs, null, "-p", "--policy-name");
      string policyFilename =
        CommandLine.GetArgument(parsedArgs, null, "-j", "--json-filename");
      if(   string.IsNullOrEmpty(policyName)
         || (string.IsNullOrEmpty(policyFilename) || !policyFilename.EndsWith(".json")))
        CommandLine.ErrorExit(
          "\nOne or more of the required arguments is missing or incorrect." +
          "\nRun the command with no arguments to see help.");

      // Create an IAM service client
      var iamClient = new AmazonIdentityManagementServiceClient();

      // Create the new policy
      var response = await CreateManagedPolicy(iamClient, policyName, policyFilename);
      Console.WriteLine($"\nPolicy {response.Policy.PolicyName} has been created.");
      Console.WriteLine($"  Arn: {response.Policy.Arn}");
    }


    //
    // Method to create an IAM policy from a JSON file
    private static async Task<CreatePolicyResponse> CreateManagedPolicy(
      IAmazonIdentityManagementService iamClient, string policyName, string jsonFilename)
    {
      return await iamClient.CreatePolicyAsync(new CreatePolicyRequest{
        PolicyName = policyName,
        PolicyDocument = File.ReadAllText(jsonFilename)});
    }
```

```
    //
    // Command-line help
    private static void PrintHelp()
    {
      Console.WriteLine(
        "\nUsage: IamCreatePolicyFromJson -p <policy-name> -j <json-filename>" +
        "\n  -p, --policy-name: The name you want the new policy to have." +
        "\n  -j, --json-filename: The name of the JSON file with the policy document.");
    }
  }


  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
=
  // Class that represents a command line on the console or terminal.
  // (This is the same for all examples. When you have seen it once, you can ignore it.)
  static class CommandLine
  {
    //
    // Method to parse a command line of the form: "--key value" or "-k value".
    //
    // Parameters:
    // - args: The command-line arguments passed into the application by the system.
    //
    // Returns:
    // A Dictionary with string Keys and Values.
    //
    // If a key is found without a matching value, Dictionary.Value is set to the key
    //  (including the dashes).
    // If a value is found without a matching key, Dictionary.Key is set to "--NoKeyN",
    //  where "N" represents sequential numbers.
    public static Dictionary<string,string> Parse(string[] args)
    {
      var parsedArgs = new Dictionary<string,string>();
      int i = 0, n = 0;
      while(i < args.Length)
      {
        // If the first argument in this iteration starts with a dash it's an option.
        if(args[i].StartsWith("-"))
        {
          var key = args[i++];
          var value = key;

          // Check to see if there's a value that goes with this option?
          if((i < args.Length) && (!args[i].StartsWith("-"))) value = args[i++];
          parsedArgs.Add(key, value);
        }

        // If the first argument in this iteration doesn't start with a dash, it's a value
        else
        {
          parsedArgs.Add("--NoKey" + n.ToString(), args[i++]);
          n++;
        }
      }

      return parsedArgs;
    }

    //
    // Method to get an argument from the parsed command-line arguments
    //
    // Parameters:
    // - parsedArgs: The Dictionary object returned from the Parse() method (shown above).
```

```
    // - defaultValue: The default string to return if the specified key isn't in
 parsedArgs.
    // - keys: An array of keys to look for in parsedArgs.
    public static string GetArgument(
      Dictionary<string,string> parsedArgs, string defaultReturn, params string[] keys)
    {
      string retval = null;
      foreach(var key in keys)
        if(parsedArgs.TryGetValue(key, out retval)) break;
      return retval ?? defaultReturn;
    }

    //
    // Method to exit the application with an error.
    public static void ErrorExit(string msg, int code=1)
    {
      Console.WriteLine("\nError");
      Console.WriteLine(msg);
      Environment.Exit(code);
    }
  }

}
```

Additional considerations

- The following is an example policy document that you can copy into a JSON file and use as input for this application:

```
{
  "Version" : "2012-10-17",
  "Id"  : "DotnetTutorialPolicy",
  "Statement" : [
    {
      "Sid" : "DotnetTutorialPolicyS3",
      "Effect" : "Allow",
      "Action" : [
        "s3:Get*",
        "s3:List*"
      ],
      "Resource" : "*"
    },
    {
      "Sid" : "DotnetTutorialPolicyPolly",
      "Effect": "Allow",
      "Action": [
        "polly:DescribeVoices",
        "polly:SynthesizeSpeech"
      ],
      "Resource": "*"
    }
  ]
}
```

- You can verify that the policy was created by looking in the IAM console. In the **Filter policies** drop-down list, select **Customer managed**. Delete the policy when you no longer need it.

- For more information about policy creation, see Creating IAM policies and the IAM JSON policy reference in the IAM User Guide

Display the policy document of an IAM managed policy

This example shows you how to use the AWS SDK for .NET to display a policy document. The application creates an IAM client object, finds the default version of the given IAM managed policy, and then displays the policy document in JSON.

The following sections provide snippets of this example. The complete code for the example (p. 182) is shown after that, and can be built and run as is.

Topics

Find the default version

The following snippet finds the default version of the given IAM policy.

The example at the end of this topic (p. 182) shows this snippet in use.

```
//
// Method to determine the default version of an IAM policy
// Returns a string with the version
private static async Task<string> GetDefaultVersion(
  IAmazonIdentityManagementService iamClient, string policyArn)
{
  // Retrieve all the versions of this policy
  string defaultVersion = string.Empty;
  ListPolicyVersionsResponse reponseVersions =
    await iamClient.ListPolicyVersionsAsync(new ListPolicyVersionsRequest{
      PolicyArn = policyArn});

  // Find the default version
  foreach(PolicyVersion version in reponseVersions.Versions)
  {
    if(version.IsDefaultVersion)
    {
      defaultVersion = version.VersionId;
      break;
    }
  }

  return defaultVersion;
}
```

Display the policy document

The following snippet displays the policy document in JSON of the given IAM policy.

The example at the end of this topic (p. 182) shows this snippet in use.

```
//
// Method to retrieve and display the policy document of an IAM policy
private static async Task ShowPolicyDocument(
  IAmazonIdentityManagementService iamClient, string policyArn, string defaultVersion)
{
  // Retrieve the policy document of the default version
  GetPolicyVersionResponse responsePolicy =
    await iamClient.GetPolicyVersionAsync(new GetPolicyVersionRequest{
```

```
          PolicyArn = policyArn,
          VersionId = defaultVersion});

      // Display the policy document (in JSON)
      Console.WriteLine($"Version {defaultVersion} of the policy (in JSON format):");
      Console.WriteLine(
        $"{HttpUtility.UrlDecode(responsePolicy.PolicyVersion.Document)}");
    }
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.IdentityManagement

Programming elements:

- Namespace Amazon.IdentityManagement

  Class AmazonIdentityManagementServiceClient
- Namespace Amazon.IdentityManagement.Model

  Class GetPolicyVersionRequest

  Class GetPolicyVersionResponse

  Class ListPolicyVersionsRequest

  Class ListPolicyVersionsResponse

  Class PolicyVersion

The code

```
using System;
using System.Web;
using System.Threading.Tasks;
using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

namespace IamDisplayPolicyJson
{
  class Program
  {
    static async Task Main(string[] args)
    {
      // Parse the command line and show help if necessary
      if(args.Length != 1)
      {
        Console.WriteLine("\nUsage: IamDisplayPolicyJson policy-arn");
        Console.WriteLine("   policy-arn: The ARN of the policy to retrieve.");
        return;
      }
      if(!args[0].StartsWith("arn:"))
      {
        Console.WriteLine("\nCould not find policy ARN in the command-line arguments:");
```

```
        Console.WriteLine($"{args[0]}");
        return;
      }

      // Create an IAM service client
      var iamClient = new AmazonIdentityManagementServiceClient();

      // Retrieve and display the policy document of the given policy
      string defaultVersion = await GetDefaultVersion(iamClient, args[0]);
      if(string.IsNullOrEmpty(defaultVersion))
        Console.WriteLine($"Could not find the default version for policy {args[0]}.");
      else
        await ShowPolicyDocument(iamClient, args[0], defaultVersion);
    }


    //
    // Method to determine the default version of an IAM policy
    // Returns a string with the version
    private static async Task<string> GetDefaultVersion(
      IAmazonIdentityManagementService iamClient, string policyArn)
    {
      // Retrieve all the versions of this policy
      string defaultVersion = string.Empty;
      ListPolicyVersionsResponse reponseVersions =
        await iamClient.ListPolicyVersionsAsync(new ListPolicyVersionsRequest{
          PolicyArn = policyArn});

      // Find the default version
      foreach(PolicyVersion version in reponseVersions.Versions)
      {
        if(version.IsDefaultVersion)
        {
          defaultVersion = version.VersionId;
          break;
        }
      }

      return defaultVersion;
    }


    //
    // Method to retrieve and display the policy document of an IAM policy
    private static async Task ShowPolicyDocument(
      IAmazonIdentityManagementService iamClient, string policyArn, string defaultVersion)
    {
      // Retrieve the policy document of the default version
      GetPolicyVersionResponse responsePolicy =
        await iamClient.GetPolicyVersionAsync(new GetPolicyVersionRequest{
          PolicyArn = policyArn,
          VersionId = defaultVersion});

      // Display the policy document (in JSON)
      Console.WriteLine($"Version {defaultVersion} of the policy (in JSON format):");
      Console.WriteLine(
        $"{HttpUtility.UrlDecode(responsePolicy.PolicyVersion.Document)}");
    }
  }
}
```

Granting access by using an IAM role

This tutorial shows you how to use the AWS SDK for .NET to enable IAM roles on Amazon EC2 instances.

Overview

All requests to AWS must be cryptographically signed by using credentials issued by AWS. Therefore, you need a strategy to manage credentials for applications that run on Amazon EC2 instances. You have to distribute, store, and rotate these credentials securely, but also keep them accessible to the applications.

With IAM roles, you can effectively manage these credentials. You create an IAM role and configure it with the permissions that an application requires, and then attach that role to an EC2 instance. Read more about the benefits of using IAM roles in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances. Also see the information about IAM Roles in the IAM User Guide.

For an application that is built using the AWS SDK for .NET, when the application constructs a client object for an AWS service, the object searches for credentials from several potential sources. The order in which it searches is shown in Credential and profile resolution (p. 24).

If the client object doesn't find credentials from any other source, it retrieves temporary credentials that have the same permissions as those that have been configured into the IAM role and are in the metadata of the EC2 instance. These credentials are used to make calls to AWS from the client object.

About this tutorial

As you follow this tutorial, you use the AWS SDK for .NET (and other tools) to launch an Amazon EC2 instance with an IAM role attached, and then see an application on the instance using the permissions of the IAM role.

Topics

Create a sample Amazon S3 application

This sample application retrieves an object from Amazon S3. To run the application, you need the following:

- An Amazon S3 bucket that contains a text file.
- AWS credentials on your development machine that allow you to access to the bucket.

For information about creating an Amazon S3 bucket and uploading an object, see the Amazon S3 Getting Started Guide. For information about AWS credentials, see Configure AWS credentials (p. 19).

Create a .NET Core project with the following code. Then test the application on your development machine.

> **Note**
> On your development machine, the .NET Core Runtime is installed, which enables you to run the application without publishing it. When you create an EC2 instance later in this tutorial, you can choose to install the .NET Core Runtime on the instance. This gives you a similar experience and a smaller file transfer.
> However, you can also choose not to install the .NET Core Runtime on the instance. If you choose this course of action, you must publish the application so that all dependencies are included when you transfer it to the instance.

SDK references

NuGet packages:

- AWSSDK.S3

Programming elements:

- Namespace Amazon.S3

  Class AmazonS3Client
- Namespace Amazon.S3.Model

  Class GetObjectResponse

The code

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;
using Amazon.S3;
using Amazon.S3.Model;

namespace S3GetTextItem
{
  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 =
  // Class to retrieve a text file from an S3 bucket and write it to a local file
  class Program
  {
    static async Task Main(string[] args)
    {
      // Parse the command line and show help if necessary
      var parsedArgs = CommandLine.Parse(args);
      if(parsedArgs.Count == 0)
      {
        PrintHelp();
        return;
      }

      // Get the application arguments from the parsed list
      string bucket =
        CommandLine.GetArgument(parsedArgs, null, "-b", "--bucket-name");
      string item =
        CommandLine.GetArgument(parsedArgs, null, "-t", "--text-object");
      string outFile =
        CommandLine.GetArgument(parsedArgs, null, "-o", "--output-filename");
      if(   string.IsNullOrEmpty(bucket)
         || string.IsNullOrEmpty(item)
         || string.IsNullOrEmpty(outFile))
        CommandLine.ErrorExit(
          "\nOne or more of the required arguments is missing or incorrect." +
          "\nRun the command with no arguments to see help.");

      // Create the S3 client object and get the file object from the bucket.
      var response = await GetObject(new AmazonS3Client(), bucket, item);

      // Write the contents of the file object to the given output file.
      var reader = new StreamReader(response.ResponseStream);
      string contents = reader.ReadToEnd();
      using (var s = new FileStream(outFile, FileMode.Create))
```

```
      using (var writer = new StreamWriter(s))
        writer.WriteLine(contents);
    }


    //
    // Method to get an object from an S3 bucket.
    private static async Task<GetObjectResponse> GetObject(
      IAmazonS3 s3Client, string bucket, string item)
    {
        Console.WriteLine($"Retrieving {item} from bucket {bucket}.");
        return await s3Client.GetObjectAsync(bucket, item);
    }


    //
    // Command-line help
    private static void PrintHelp()
    {
      Console.WriteLine(
        "\nUsage: S3GetTextItem -b <bucket-name> -t <text-object> -o <output-filename>" +
        "\n  -b, --bucket-name: The name of the S3 bucket." +
        "\n  -t, --text-object: The name of the text object in the bucket." +
        "\n  -o, --output-filename: The name of the file to write the text to.");
    }
  }


  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
=
  // Class that represents a command line on the console or terminal.
  // (This is the same for all examples. When you have seen it once, you can ignore it.)
  static class CommandLine
  {
    //
    // Method to parse a command line of the form: "--key value" or "-k value".
    //
    // Parameters:
    // - args: The command-line arguments passed into the application by the system.
    //
    // Returns:
    // A Dictionary with string Keys and Values.
    //
    // If a key is found without a matching value, Dictionary.Value is set to the key
    //  (including the dashes).
    // If a value is found without a matching key, Dictionary.Key is set to "--NoKeyN",
    //  where "N" represents sequential numbers.
    public static Dictionary<string,string> Parse(string[] args)
    {
      var parsedArgs = new Dictionary<string,string>();
      int i = 0, n = 0;
      while(i < args.Length)
      {
        // If the first argument in this iteration starts with a dash it's an option.
        if(args[i].StartsWith("-"))
        {
          var key = args[i++];
          var value = key;

          // Check to see if there's a value that goes with this option?
          if((i < args.Length) && (!args[i].StartsWith("-"))) value = args[i++];
          parsedArgs.Add(key, value);
        }

        // If the first argument in this iteration doesn't start with a dash, it's a value
        else
```

```
        {
          parsedArgs.Add("--NoKey" + n.ToString(), args[i++]);
          n++;
        }
      }

      return parsedArgs;
    }

    //
    // Method to get an argument from the parsed command-line arguments
    //
    // Parameters:
    // - parsedArgs: The Dictionary object returned from the Parse() method (shown above).
    // - defaultValue: The default string to return if the specified key isn't in
 parsedArgs.
    // - keys: An array of keys to look for in parsedArgs.
    public static string GetArgument(
      Dictionary<string,string> parsedArgs, string defaultReturn, params string[] keys)
    {
      string retval = null;
      foreach(var key in keys)
        if(parsedArgs.TryGetValue(key, out retval)) break;
      return retval ?? defaultReturn;
    }

    //
    // Method to exit the application with an error.
    public static void ErrorExit(string msg, int code=1)
    {
      Console.WriteLine("\nError");
      Console.WriteLine(msg);
      Environment.Exit(code);
    }
  }

}
```

If you want, you can temporarily remove the credentials you use on your development machine to see how the application responds. (But be sure to restore the credentials when you're finished.)

Create an IAM role

Create an IAM role that has the appropriate permissions to access Amazon S3.

1. Open the IAM console.
2. In the navigation pane, choose **Roles**, and then choose **Create Role**.
3. Select **AWS service**, find and choose **EC2**, and choose **Next: Permissions**.
4. Under **Attach permissions policies**, find and select **AmazonS3ReadOnlyAccess**. Review the policy if you want to, and then choose **Next: Tags**.
5. Add tags if you want and then choose **Next: Review**.
6. Type a name and description for the role, and then choose **Create role**. Remember this name because you'll need it when you launch your EC2 instance.

Launch an EC2 instance and attach the IAM role

Launch an EC2 instance with the IAM role you created previously. You can do so in the following ways.

- **Using the EC2 console**

Follow the directions to launch an instance in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances.

As you go through the wizard you should at least visit the **Configure Instance Details** page so that you can select the **IAM role** you created earlier.

- **Using the AWS SDK for .NET**

  For information about this, see Launching an Amazon EC2 instance (p. 146), including the Additional considerations (p. 153) near the end of that topic.

To launch an EC2 instance that has an IAM role attached, an IAM user's configuration must include certain permissions. For more information about the required permissions, see the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances.

Connect to the EC2 instance

Connect to the EC2 instance so that you can transfer the sample application to it and then run the application. You'll need the file that contains the private portion of the key pair you used to launch the instance; that is, the PEM file.

You can do this by following the connect procedure in the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances. When you connect, do so in such a way that you can transfer files from your development machine to your instance.

If you're using Visual Studio on Windows, you can also connect to the instance by using the Toolkit for Visual Studio. For more information, see Connecting to an Amazon EC2 Instance in the AWS Toolkit for Visual Studio User Guide.

Run the sample application on the EC2 instance

1. Copy the application files from your local drive to your instance.

   Which files you transfer depends on how you built the application and whether your instance has the .NET Core Runtime installed. For information about how to transfer files to your instance see the Amazon EC2 User Guide for Linux Instances or the Amazon EC2 User Guide for Windows Instances.
2. Start the application and verify that it runs with the same results as on your development machine.
3. Verify that the application uses the credentials provided by the IAM role.

   a. Open the Amazon EC2 console.

   b. Select the instance and detach the IAM role through **Actions**, **Instance Settings**, **Attach/Replace IAM Role**.

   c. Run the application again and see that it returns an authorization error.

Clean up

When you are finished with this tutorial, and if you no longer want the EC2 instance you created, be sure to terminate the instance to avoid unwanted cost. You can do so in the Amazon EC2 console or programmatically, as described in Terminating an Amazon EC2 instance (p. 157). If you want to, you can also delete other resources that you created for this tutorial. These might include an IAM role, an EC2 keypair and PEM file, a security group, etc.

Using Amazon Simple Storage Service Internet storage

The AWS SDK for .NET supports Amazon S3, which is storage for the Internet. It is designed to make web-scale computing easier for developers.

APIs

The AWS SDK for .NET provides APIs for Amazon S3 clients. The APIs enable you to work with Amazon S3 resources such as buckets and items. To view the full set of APIs for Amazon S3, see the following:

- AWS SDK for .NET API Reference (and scroll to "Amazon.S3").
- Amazon.Extensions.S3.Encryption documentation

The Amazon S3 APIs are provided by the following NuGet packages:

- AWSSDK.S3
- Amazon.Extensions.S3.Encryption

Prerequisites

Before you begin, be sure you have set up your environment (p. 15). Also review the information in Setting up your project (p. 17) and SDK features (p. 49).

Examples in this document

The following topics in this document show you how to use the AWS SDK for .NET to work with Amazon S3.

- Using KMS keys for S3 encryption (p. 190)

Examples in other documents

The following links to the Amazon S3 Developer Guide provide additional examples of how to use the AWS SDK for .NET to work with Amazon S3.

> **Note**
> Although these examples and additional programming considerations were created for Version 3 of the AWS SDK for .NET using .NET Framework, they are also viable for later versions of the AWS SDK for .NET using .NET Core. Small adjustments in the code are sometimes necessary.

Amazon S3 programming examples

- Managing ACLs
- Creating a Bucket
- Upload an Object
- Multipart Upload with the High-Level API (Amazon.S3.Transfer.TransferUtility)
- Multipart Upload with the Low-Level API
- Listing Objects
- Listing Keys
- Get an Object

- Copy an Object
- Copy an Object with the Multipart Upload API
- Deleting an Object
- Deleting Multiple Objects
- Restore an Object
- Configure a Bucket for Notifications
- Manage an Object's Lifecycle
- Generate a Pre-signed Object URL
- Managing Websites
- Enabling Cross-Origin Resource Sharing (CORS)

Additional programming considerations

- Using the AWS SDK for .NET for Amazon S3 Programming
- Making Requests Using IAM User Temporary Credentials
- Making Requests Using Federated User Temporary Credentials
- Specifying Server-Side Encryption
- Specifying Server-Side Encryption with Customer-Provided Encryption Keys

Using AWS KMS keys for Amazon S3 encryption in the AWS SDK for .NET

This example shows you how to use AWS Key Management Service keys to encrypt Amazon S3 objects. The application creates a customer master key (CMK) and uses it to create an AmazonS3EncryptionClientV2 object for client-side encryption. The application uses that client to create an encrypted object from a given text file in an existing Amazon S3 bucket. It then decrypts the object and displays its contents.

> **Warning**
> A similar class called `AmazonS3EncryptionClient` is deprecated and is less secure than the `AmazonS3EncryptionClientV2` class. To migrate existing code that uses `AmazonS3EncryptionClient`, see S3 Encryption Client Migration (p. 320).

Topics

Create encryption materials

The following snippet creates an `EncryptionMaterials` object that contains a KMS key ID.

The example at the end of this topic (p. 191) shows this snippet in use.

```
     // Create a customer master key (CMK) and store the result
     CreateKeyResponse createKeyResponse =
       await new AmazonKeyManagementServiceClient().CreateKeyAsync(new
CreateKeyRequest());
     var kmsEncryptionContext = new Dictionary<string, string>();
     var kmsEncryptionMaterials = new EncryptionMaterialsV2(
```

```
        createKeyResponse.KeyMetadata.KeyId, KmsType.KmsContext, kmsEncryptionContext);
```

Create and encrypt an Amazon S3 object

The following snippet creates an `AmazonS3EncryptionClientV2` object that uses the encryption materials created earlier. It then uses the client to create and encrypt a new Amazon S3 object.

The example at the end of this topic (p. 191) shows this snippet in use.

```
//
// Method to create and encrypt an object in an S3 bucket
static async Task<GetObjectResponse> CreateAndRetrieveObjectAsync(
  EncryptionMaterialsV2 materials, string bucketName,
  string fileName, string itemName)
{
  // CryptoStorageMode.ObjectMetadata is required for KMS EncryptionMaterials
  var config = new AmazonS3CryptoConfigurationV2(SecurityProfile.V2AndLegacy)
  {
    StorageMode = CryptoStorageMode.ObjectMetadata
  };
  var s3EncClient = new AmazonS3EncryptionClientV2(config, materials);

  // Create, encrypt, and put the object
  await s3EncClient.PutObjectAsync(new PutObjectRequest
  {
    BucketName = bucketName,
    Key = itemName,
    ContentBody = File.ReadAllText(fileName)
  });

  // Get, decrypt, and return the object
  return await s3EncClient.GetObjectAsync(new GetObjectRequest
  {
    BucketName = bucketName,
    Key = itemName
  });
}
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- Amazon.Extensions.S3.Encryption

Programming elements:

- Namespace Amazon.Extensions.S3.Encryption

  Class AmazonS3EncryptionClientV2

  Class AmazonS3CryptoConfigurationV2

  Class CryptoStorageMode

  Class EncryptionMaterialsV2
- Namespace Amazon.Extensions.S3.Encryption.Primitives

Class KmsType

- Namespace Amazon.S3.Model

  Class GetObjectRequest

  Class GetObjectResponse

  Class PutObjectRequest
- Namespace Amazon.KeyManagementService

  Class AmazonKeyManagementServiceClient
- Namespace Amazon.KeyManagementService.Model

  Class CreateKeyRequest

  Class CreateKeyResponse

The code

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;
using Amazon.Extensions.S3.Encryption;
using Amazon.Extensions.S3.Encryption.Primitives;
using Amazon.S3.Model;
using Amazon.KeyManagementService;
using Amazon.KeyManagementService.Model;

namespace KmsS3Encryption
{
  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 =
  // Class to store text in an encrypted S3 object.
  class Program
  {
    private const int MaxArgs = 3;

    public static async Task Main(string[] args)
    {
      // Parse the command line and show help if necessary
      var parsedArgs = CommandLine.Parse(args);
      if((parsedArgs.Count == 0) || (parsedArgs.Count > MaxArgs))
      {
        PrintHelp();
        return;
      }

      // Get the application arguments from the parsed list
      string bucketName =
        CommandLine.GetArgument(parsedArgs, null, "-b", "--bucket-name");
      string fileName =
        CommandLine.GetArgument(parsedArgs, null, "-f", "--file-name");
      string itemName =
        CommandLine.GetArgument(parsedArgs, null, "-i", "--item-name");
      if(string.IsNullOrEmpty(bucketName) || (string.IsNullOrEmpty(fileName)))
        CommandLine.ErrorExit(
          "\nOne or more of the required arguments is missing or incorrect." +
          "\nRun the command with no arguments to see help.");
      if(!File.Exists(fileName))
        CommandLine.ErrorExit($"\nThe given file {fileName} doesn't exist.");
```

```
      if(string.IsNullOrEmpty(itemName))
        itemName = Path.GetFileName(fileName);

      // Create a customer master key (CMK) and store the result
      CreateKeyResponse createKeyResponse =
        await new AmazonKeyManagementServiceClient().CreateKeyAsync(new
CreateKeyRequest());
      var kmsEncryptionContext = new Dictionary<string, string>();
      var kmsEncryptionMaterials = new EncryptionMaterialsV2(
        createKeyResponse.KeyMetadata.KeyId, KmsType.KmsContext, kmsEncryptionContext);

      // Create the object in the bucket, then display the content of the object
      var putObjectResponse =
        await CreateAndRetrieveObjectAsync(kmsEncryptionMaterials, bucketName, fileName,
itemName);
      Stream stream = putObjectResponse.ResponseStream;
      StreamReader reader = new StreamReader(stream);
      Console.WriteLine(reader.ReadToEnd());
    }


    //
    // Method to create and encrypt an object in an S3 bucket
    static async Task<GetObjectResponse> CreateAndRetrieveObjectAsync(
      EncryptionMaterialsV2 materials, string bucketName,
      string fileName, string itemName)
    {
      // CryptoStorageMode.ObjectMetadata is required for KMS EncryptionMaterials
      var config = new AmazonS3CryptoConfigurationV2(SecurityProfile.V2AndLegacy)
      {
        StorageMode = CryptoStorageMode.ObjectMetadata
      };
      var s3EncClient = new AmazonS3EncryptionClientV2(config, materials);

      // Create, encrypt, and put the object
      await s3EncClient.PutObjectAsync(new PutObjectRequest
      {
        BucketName = bucketName,
        Key = itemName,
        ContentBody = File.ReadAllText(fileName)
      });

      // Get, decrypt, and return the object
      return await s3EncClient.GetObjectAsync(new GetObjectRequest
      {
        BucketName = bucketName,
        Key = itemName
      });
    }


    //
    // Command-line help
    private static void PrintHelp()
    {
      Console.WriteLine(
        "\nUsage: KmsS3Encryption -b <bucket-name> -f <file-name> [-i <item-name>]" +
        "\n  -b, --bucket-name: The name of an existing S3 bucket." +
        "\n  -f, --file-name: The name of a text file with content to encrypt and store in
S3." +
        "\n  -i, --item-name: The name you want to use for the item." +
        "\n      If item-name isn't given, file-name will be used.");
    }

  }
```

```
 // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
=
 // Class that represents a command line on the console or terminal.
 // (This is the same for all examples. When you have seen it once, you can ignore it.)
 static class CommandLine
 {
   //
   // Method to parse a command line of the form: "--key value" or "-k value".
   //
   // Parameters:
   // - args: The command-line arguments passed into the application by the system.
   //
   // Returns:
   // A Dictionary with string Keys and Values.
   //
   // If a key is found without a matching value, Dictionary.Value is set to the key
   //  (including the dashes).
   // If a value is found without a matching key, Dictionary.Key is set to "--NoKeyN",
   //  where "N" represents sequential numbers.
   public static Dictionary<string,string> Parse(string[] args)
   {
     var parsedArgs = new Dictionary<string,string>();
     int i = 0, n = 0;
     while(i < args.Length)
     {
       // If the first argument in this iteration starts with a dash it's an option.
       if(args[i].StartsWith("-"))
       {
         var key = args[i++];
         var value = key;

         // Check to see if there's a value that goes with this option?
         if((i < args.Length) && (!args[i].StartsWith("-"))) value = args[i++];
         parsedArgs.Add(key, value);
       }

       // If the first argument in this iteration doesn't start with a dash, it's a value
       else
       {
         parsedArgs.Add("--NoKey" + n.ToString(), args[i++]);
         n++;
       }
     }

     return parsedArgs;
   }

   //
   // Method to get an argument from the parsed command-line arguments
   //
   // Parameters:
   // - parsedArgs: The Dictionary object returned from the Parse() method (shown above).
   // - defaultValue: The default string to return if the specified key isn't in
parsedArgs.
   // - keys: An array of keys to look for in parsedArgs.
   public static string GetArgument(
     Dictionary<string,string> parsedArgs, string defaultReturn, params string[] keys)
   {
     string retval = null;
     foreach(var key in keys)
       if(parsedArgs.TryGetValue(key, out retval)) break;
     return retval ?? defaultReturn;
   }

   //
   // Method to exit the application with an error.
```

```
    public static void ErrorExit(string msg, int code=1)
    {
      Console.WriteLine("\nError");
      Console.WriteLine(msg);
      Environment.Exit(code);
    }
  }

}
```

Additional considerations

- You can check the results of this example. To do so, go to the Amazon S3 console and open the bucket you provided to the application. Then find the new object, download it, and open it in a text editor.

- The AmazonS3EncryptionClientV2 class implements the same interface as the standard `AmazonS3Client` class. This makes it easier to port your code to the `AmazonS3EncryptionClientV2` class so that encryption and decryption happen automatically and transparently in the client.

- One advantage of using an AWS KMS key as your master key is that you don't need to store and manage your own master keys; this is done by AWS. A second advantage is that the `AmazonS3EncryptionClientV2` class of the AWS SDK for .NET is interoperable with the `AmazonS3EncryptionClientV2` class of the AWS SDK for Java. This means you can encrypt with the AWS SDK for Java and decrypt with the AWS SDK for .NET, and vice versa.

  Note
  The `AmazonS3EncryptionClientV2` class of the AWS SDK for .NET supports KMS master keys only when run in metadata mode. The instruction file mode of the `AmazonS3EncryptionClientV2` class of the AWS SDK for .NET is incompatible with the `AmazonS3EncryptionClientV2` class of the AWS SDK for Java.

- For more information about client-side encryption with the `AmazonS3EncryptionClientV2` class, and how envelope encryption works, see Client Side Data Encryption with AWS SDK for .NET and Amazon S3.

Sending Notifications From the Cloud Using Amazon Simple Notification Service

Note
The information in this topic is specific to projects based on .NET Framework and the AWS SDK for .NET version 3.3 and earlier.

The AWS SDK for .NET supports Amazon Simple Notification Service (Amazon SNS), which is a web service that enables applications, end users, and devices to instantly send notifications from the cloud. For more information, see Amazon SNS.

Listing Your Amazon SNS Topics

The following example shows how to list your Amazon SNS topics, the subscriptions to each topic, and the attributes for each topic. This example uses the default AmazonSimpleNotificationServiceClient, which loads credentials from your default configuration.

```
// using Amazon.SimpleNotificationService;
```

```
// using Amazon.SimpleNotificationService.Model;

var client = new AmazonSimpleNotificationServiceClient();
var request = new ListTopicsRequest();
var response = new ListTopicsResponse();

do
{
  response = client.ListTopics(request);

  foreach (var topic in response.Topics)
  {
    Console.WriteLine("Topic: {0}", topic.TopicArn);

    var subs = client.ListSubscriptionsByTopic(
      new ListSubscriptionsByTopicRequest
      {
        TopicArn = topic.TopicArn
      });

    var ss = subs.Subscriptions;

    if (ss.Any())
    {
      Console.WriteLine("  Subscriptions:");

      foreach (var sub in ss)
      {
        Console.WriteLine("    {0}", sub.SubscriptionArn);
      }
    }

    var attrs = client.GetTopicAttributes(
      new GetTopicAttributesRequest
      {
        TopicArn = topic.TopicArn
      }).Attributes;

    if (attrs.Any())
    {
      Console.WriteLine("  Attributes:");

      foreach (var attr in attrs)
      {
        Console.WriteLine("    {0} = {1}", attr.Key, attr.Value);
      }
    }

    Console.WriteLine();
  }

  request.NextToken = response.NextToken;

} while (!string.IsNullOrEmpty(response.NextToken));
```

Sending a Message to an Amazon SNS Topic

The following example shows how to send a message to an Amazon SNS topic. The example takes one argument, the ARN of the Amazon SNS topic.

```
using System;
using System.Linq;
using System.Threading.Tasks;
```

```
using Amazon;
using Amazon.SimpleNotificationService;
using Amazon.SimpleNotificationService.Model;

namespace SnsSendMessage
{
    class Program
    {
        static void Main(string[] args)
        {
            /* Topic ARNs must be in the correct format:
             *   arn:aws:sns:REGION:ACCOUNT_ID:NAME
             *
             *  where:
             *  REGION     is the region in which the topic is created, such as us-west-2
             *  ACCOUNT_ID is your (typically) 12-character account ID
             *  NAME       is the name of the topic
             */
            string topicArn = args[0];
            string message = "Hello at " + DateTime.Now.ToShortTimeString();

            var client = new AmazonSimpleNotificationServiceClient(region:
 Amazon.RegionEndpoint.USWest2);

            var request = new PublishRequest
            {
                Message = message,
                TopicArn = topicArn
            };

            try
            {
                var response = client.Publish(request);

                Console.WriteLine("Message sent to topic:");
                Console.WriteLine(message);
            }
            catch (Exception ex)
            {
                Console.WriteLine("Caught exception publishing request:");
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

See the complete example, including information on how to build and run the example from the command line, on GitHub.

Sending an SMS Message to a Phone Number

The following example shows how to send an SMS message to a telephone number. The example takes one argument, the telephone number, which must be in either of the two formats described in the comments.

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Amazon;
using Amazon.SimpleNotificationService;
using Amazon.SimpleNotificationService.Model;

namespace SnsPublish
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            // US phone numbers must be in the correct format:
            // +1 (nnn) nnn-nnnn OR +1nnnnnnnnnn
            string number = args[0];
            string message = "Hello at " + DateTime.Now.ToShortTimeString();

            var client = new AmazonSimpleNotificationServiceClient(region:
 Amazon.RegionEndpoint.USWest2);
            var request = new PublishRequest
            {
                Message = message,
                PhoneNumber = number
            };

            try
            {
                var response = client.Publish(request);

                Console.WriteLine("Message sent to " + number + ":");
                Console.WriteLine(message);
            }
            catch (Exception ex)
            {
                Console.WriteLine("Caught exception publishing request:");
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

See the complete example, including information on how to build and run the example from the command line, on GitHub.

Messaging using Amazon SQS

The AWS SDK for .NET supports Amazon Simple Queue Service (Amazon SQS), which is a message queuing service that handles messages or workflows between components in a system.

Amazon SQS queues provide a mechanism that enables you to send, store, and receive messages between software components such as microservices, distributed systems, and serverless applications. This enables you to decouple such components and frees you from the need to design and operate your own messaging system. For information about how queues and messages work in Amazon SQS, see Amazon SQS tutorials and How Amazon SQS works in the Amazon Simple Queue Service Developer Guide.

> **Important**
> Due to the distributed nature of queues, Amazon SQS can't guarantee that you'll receive messages in the precise order they're sent. If you need to preserve message order, use an Amazon SQS FIFO queue.

APIs

The AWS SDK for .NET provides APIs for Amazon SQS clients. The APIs enable you to work with Amazon SQS features such as queues and messages. This section contains a small number of examples that show you the patterns you can follow when working with these APIs. To view the full set of APIs, see the AWS SDK for .NET API Reference (and scroll to "Amazon.SQS").

The Amazon SQS APIs are provided by the AWSSDK.SQS NuGet package.

Prerequisites

Before you begin, be sure you have set up your environment (p. 15). Also review the information in Setting up your project (p. 17) and SDK features (p. 49).

Topics

Topics

Creating Amazon SQS queues

This example shows you how to use the AWS SDK for .NET to create an Amazon SQS queue. The application creates a dead-letter queue if you don't supply the ARN for one. It then creates a standard message queue, which includes a dead-letter queue (the one you supplied or the one that was created).

If you don't supply any command-line arguments, the application simply shows information about all existing queues.

The following sections provide snippets of this example. The complete code for the example (p. 201) is shown after that, and can be built and run as is.

Topics

Show existing queues

The following snippet shows a list of the existing queues in the SQS client's region and the attributes of each queue.

The example at the end of this topic (p. 201) shows this snippet in use.

```
//
// Method to show a list of the existing queues
private static async Task ShowQueues(IAmazonSQS sqsClient)
{
  ListQueuesResponse responseList = await sqsClient.ListQueuesAsync("");
  Console.WriteLine();
  foreach(string qUrl in responseList.QueueUrls)
  {
    // Get and show all attributes. Could also get a subset.
    await ShowAllAttributes(sqsClient, qUrl);
  }
}

//
```

```
    // Method to show all attributes of a queue
    private static async Task ShowAllAttributes(IAmazonSQS sqsClient, string qUrl)
    {
      var attributes = new List<string>{ QueueAttributeName.All };
      GetQueueAttributesResponse responseGetAtt =
        await sqsClient.GetQueueAttributesAsync(qUrl, attributes);
      Console.WriteLine($"Queue: {qUrl}");
      foreach(var att in responseGetAtt.Attributes)
        Console.WriteLine($"\t{att.Key}: {att.Value}");
    }
```

Create the queue

The following snippet creates a queue. The snippet includes the use of a dead-letter queue, but a dead-letter queue isn't necessarily required for your queues.

The example at the end of this topic (p. 201) shows this snippet in use.

```
    //
    // Method to create a queue. Returns the queue URL.
    private static async Task<string> CreateQueue(
      IAmazonSQS sqsClient, string qName, string deadLetterQueueUrl=null,
      string maxReceiveCount=null, string receiveWaitTime=null)
    {
      var attrs = new Dictionary<string, string>();

      // If a dead-letter queue is given, create a message queue
      if(!string.IsNullOrEmpty(deadLetterQueueUrl))
      {
        attrs.Add(QueueAttributeName.ReceiveMessageWaitTimeSeconds, receiveWaitTime);
        attrs.Add(QueueAttributeName.RedrivePolicy,
          $"{{\"deadLetterTargetArn\":\"{await GetQueueArn(sqsClient,
deadLetterQueueUrl)}\"," +
          $"\"maxReceiveCount\":\"{maxReceiveCount}\"}}");
        // Add other attributes for the message queue such as VisibilityTimeout
      }

      // If no dead-letter queue is given, create one of those instead
      //else
      //{
      //  // Add attributes for the dead-letter queue as needed
      //  attrs.Add();
      //}

      // Create the queue
      CreateQueueResponse responseCreate = await sqsClient.CreateQueueAsync(
          new CreateQueueRequest{QueueName = qName, Attributes = attrs});
      return responseCreate.QueueUrl;
    }
```

Get a queue's ARN

The following snippet gets the ARN of the queue identified by the given queue URL.

The example at the end of this topic (p. 201) shows this snippet in use.

```
    //
    // Method to get the ARN of a queue
    private static async Task<string> GetQueueArn(IAmazonSQS sqsClient, string qUrl)
    {
      GetQueueAttributesResponse responseGetAtt = await sqsClient.GetQueueAttributesAsync(
        qUrl, new List<string>{QueueAttributeName.QueueArn});
```

```
      return responseGetAtt.QueueARN;
    }
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.SQS

Programming elements:

- Namespace Amazon.SQS

  Class AmazonSQSClient

  Class QueueAttributeName
- Namespace Amazon.SQS.Model

  Class CreateQueueRequest

  Class CreateQueueResponse

  Class GetQueueAttributesResponse

  Class ListQueuesResponse

The code

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
using Amazon.SQS;
using Amazon.SQS.Model;

namespace SQSCreateQueue
{
  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 =
  // Class to create a queue
  class Program
  {
    private const string MaxReceiveCount = "10";
    private const string ReceiveMessageWaitTime = "2";
    private const int MaxArgs = 3;

    static async Task Main(string[] args)
    {
      // Parse the command line and show help if necessary
      var parsedArgs = CommandLine.Parse(args);
      if(parsedArgs.Count > MaxArgs)
        CommandLine.ErrorExit(
          "\nToo many command-line arguments.\nRun the command with no arguments to see
 help.");

      // Create the Amazon SQS client
      var sqsClient = new AmazonSQSClient();
```

```
    // In the case of no command-line arguments, just show help and the existing queues
    if(parsedArgs.Count == 0)
    {
      PrintHelp();
      Console.WriteLine("\nNo arguments specified.");
      Console.Write("Do you want to see a list of the existing queues? ((y) or n): ");
      string response = Console.ReadLine();
      if((string.IsNullOrEmpty(response)) || (response.ToLower() == "y"))
        await ShowQueues(sqsClient);
      return;
    }

    // Get the application arguments from the parsed list
    string queueName =
      CommandLine.GetArgument(parsedArgs, null, "-q", "--queue-name");
    string deadLetterQueueUrl =
      CommandLine.GetArgument(parsedArgs, null, "-d", "--dead-letter-queue");
    string maxReceiveCount =
      CommandLine.GetArgument(parsedArgs, MaxReceiveCount, "-m", "--max-receive-count");
    string receiveWaitTime =
      CommandLine.GetArgument(parsedArgs, ReceiveMessageWaitTime, "-w", "--wait-time");

    if(string.IsNullOrEmpty(queueName))
      CommandLine.ErrorExit(
        "\nYou must supply a queue name.\nRun the command with no arguments to see
help.");

    // If a dead-letter queue wasn't given, create one
    if(string.IsNullOrEmpty(deadLetterQueueUrl))
    {
      Console.WriteLine("\nNo dead-letter queue was specified. Creating one...");
      deadLetterQueueUrl = await CreateQueue(sqsClient, queueName + "__dlq");
      Console.WriteLine($"Your new dead-letter queue:");
      await ShowAllAttributes(sqsClient, deadLetterQueueUrl);
    }

    // Create the message queue
    string messageQueueUrl = await CreateQueue(
      sqsClient, queueName, deadLetterQueueUrl, maxReceiveCount, receiveWaitTime);
    Console.WriteLine($"Your new message queue:");
    await ShowAllAttributes(sqsClient, messageQueueUrl);
  }


  //
  // Method to show a list of the existing queues
  private static async Task ShowQueues(IAmazonSQS sqsClient)
  {
    ListQueuesResponse responseList = await sqsClient.ListQueuesAsync("");
    Console.WriteLine();
    foreach(string qUrl in responseList.QueueUrls)
    {
      // Get and show all attributes. Could also get a subset.
      await ShowAllAttributes(sqsClient, qUrl);
    }
  }


  //
  // Method to create a queue. Returns the queue URL.
  private static async Task<string> CreateQueue(
    IAmazonSQS sqsClient, string qName, string deadLetterQueueUrl=null,
    string maxReceiveCount=null, string receiveWaitTime=null)
  {
    var attrs = new Dictionary<string, string>();
```

```
      // If a dead-letter queue is given, create a message queue
      if(!string.IsNullOrEmpty(deadLetterQueueUrl))
      {
        attrs.Add(QueueAttributeName.ReceiveMessageWaitTimeSeconds, receiveWaitTime);
        attrs.Add(QueueAttributeName.RedrivePolicy,
          $"{{\"deadLetterTargetArn\":\"{await GetQueueArn(sqsClient,
 deadLetterQueueUrl)}\"," +
          $"\"maxReceiveCount\":\"{maxReceiveCount}\"}}");
        // Add other attributes for the message queue such as VisibilityTimeout
      }

      // If no dead-letter queue is given, create one of those instead
      //else
      //{
      //  // Add attributes for the dead-letter queue as needed
      //  attrs.Add();
      //}

      // Create the queue
      CreateQueueResponse responseCreate = await sqsClient.CreateQueueAsync(
          new CreateQueueRequest{QueueName = qName, Attributes = attrs});
      return responseCreate.QueueUrl;
    }


    //
    // Method to get the ARN of a queue
    private static async Task<string> GetQueueArn(IAmazonSQS sqsClient, string qUrl)
    {
      GetQueueAttributesResponse responseGetAtt = await sqsClient.GetQueueAttributesAsync(
        qUrl, new List<string>{QueueAttributeName.QueueArn});
      return responseGetAtt.QueueARN;
    }


    //
    // Method to show all attributes of a queue
    private static async Task ShowAllAttributes(IAmazonSQS sqsClient, string qUrl)
    {
      var attributes = new List<string>{ QueueAttributeName.All };
      GetQueueAttributesResponse responseGetAtt =
        await sqsClient.GetQueueAttributesAsync(qUrl, attributes);
      Console.WriteLine($"Queue: {qUrl}");
      foreach(var att in responseGetAtt.Attributes)
        Console.WriteLine($"\t{att.Key}: {att.Value}");
    }


    //
    // Command-line help
    private static void PrintHelp()
    {
      Console.WriteLine(
      "\nUsage: SQSCreateQueue -q <queue-name> [-d <dead-letter-queue>]" +
        " [-m <max-receive-count>] [-w <wait-time>]" +
      "\n  -q, --queue-name: The name of the queue you want to create." +
      "\n  -d, --dead-letter-queue: The URL of an existing queue to be used as the dead-
letter queue."+
      "\n      If this argument isn't supplied, a new dead-letter queue will be created." +
      "\n  -m, --max-receive-count: The value for maxReceiveCount in the RedrivePolicy of
 the queue." +
      $"\n      Default is {MaxReceiveCount}." +
      "\n  -w, --wait-time: The value for ReceiveMessageWaitTimeSeconds of the queue for
 long polling." +
      $"\n      Default is {ReceiveMessageWaitTime}.");
```

```
    }
  }


  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
=
  // Class that represents a command line on the console or terminal.
  // (This is the same for all examples. When you have seen it once, you can ignore it.)
  static class CommandLine
  {
    //
    // Method to parse a command line of the form: "--key value" or "-k value".
    //
    // Parameters:
    // - args: The command-line arguments passed into the application by the system.
    //
    // Returns:
    // A Dictionary with string Keys and Values.
    //
    // If a key is found without a matching value, Dictionary.Value is set to the key
    //  (including the dashes).
    // If a value is found without a matching key, Dictionary.Key is set to "--NoKeyN",
    //  where "N" represents sequential numbers.
    public static Dictionary<string,string> Parse(string[] args)
    {
      var parsedArgs = new Dictionary<string,string>();
      int i = 0, n = 0;
      while(i < args.Length)
      {
        // If the first argument in this iteration starts with a dash it's an option.
        if(args[i].StartsWith("-"))
        {
          var key = args[i++];
          var value = key;

          // Check to see if there's a value that goes with this option?
          if((i < args.Length) && (!args[i].StartsWith("-"))) value = args[i++];
          parsedArgs.Add(key, value);
        }

        // If the first argument in this iteration doesn't start with a dash, it's a value
        else
        {
          parsedArgs.Add("--NoKey" + n.ToString(), args[i++]);
          n++;
        }
      }

      return parsedArgs;
    }

    //
    // Method to get an argument from the parsed command-line arguments
    //
    // Parameters:
    // - parsedArgs: The Dictionary object returned from the Parse() method (shown above).
    // - defaultValue: The default string to return if the specified key isn't in
parsedArgs.
    // - keys: An array of keys to look for in parsedArgs.
    public static string GetArgument(
      Dictionary<string,string> parsedArgs, string defaultReturn, params string[] keys)
    {
      string retval = null;
      foreach(var key in keys)
        if(parsedArgs.TryGetValue(key, out retval)) break;
      return retval ?? defaultReturn;
```

```
    }

    //
    // Method to exit the application with an error.
    public static void ErrorExit(string msg, int code=1)
    {
      Console.WriteLine("\nError");
      Console.WriteLine(msg);
      Environment.Exit(code);
    }
  }

}
```

Additional considerations

- Your queue name must be composed of alphanumeric characters, hyphens, and underscores.
- Queue names and queue URLs are case-sensitive
- If you need the queue URL but only have the queue name, use one of the `AmazonSQSClient.GetQueueUrlAsync` methods.
- For information about the various queue attributes you can set, see CreateQueueRequest in the AWS SDK for .NET API Reference or SetQueueAttributes in the Amazon Simple Queue Service API Reference.
- This example specifies long polling for all messages on the queue that you create. This is done by using the `ReceiveMessageWaitTimeSeconds` attribute.

  You can also specify long polling during a call to the `ReceiveMessageAsync` methods of the AmazonSQSClient class. For more information, see Receiving Amazon SQS messages (p. 217).

  For information about short polling versus long polling, see Short and long polling in the *Amazon Simple Queue Service Developer Guide*.
- A dead letter queue is one that other (source) queues can target for messages that aren't processed successfully. For more information, see Amazon SQS dead-letter queues in the Amazon Simple Queue Service Developer Guide.
- You can also see the list of queues and the results of this example in the Amazon SQS console.

Updating Amazon SQS queues

This example shows you how to use the AWS SDK for .NET to update an Amazon SQS queue. After some checks, the application updates the given attribute with the given value, and then shows all attributes for the queue.

If only the queue URL is included in the command-line arguments, the application simply shows all attributes for the queue.

The following sections provide snippets of this example. The complete code for the example (p. 207) is shown after that, and can be built and run as is.

Topics

Show queue attributes

The following snippet shows the attributes of the queue identified by the given queue URL.

The example at the end of this topic (p. 207) shows this snippet in use.

```
//
// Method to show all attributes of a queue
private static async Task ShowAllAttributes(IAmazonSQS sqsClient, string qUrl)
{
  GetQueueAttributesResponse responseGetAtt =
    await sqsClient.GetQueueAttributesAsync(qUrl,
      new List<string>{ QueueAttributeName.All });
  Console.WriteLine($"Queue: {qUrl}");
  foreach(var att in responseGetAtt.Attributes)
    Console.WriteLine($"\t{att.Key}: {att.Value}");
}
```

Validate attribute name

The following snippet validates the name of the attribute being updated.

The example at the end of this topic (p. 207) shows this snippet in use.

```
//
// Method to check the name of the attribute
private static bool ValidAttribute(string attribute)
{
  var attOk = false;
  var qAttNameType = typeof(QueueAttributeName);
  List<string> qAttNamefields = new List<string>();
  foreach(var field in qAttNameType.GetFields())
   qAttNamefields.Add(field.Name);
  foreach(var name in qAttNamefields)
    if(attribute == name) { attOk = true; break; }
  return attOk;
}
```

Update queue attribute

The following snippet updates an attribute of the queue identified by the given queue URL.

The example at the end of this topic (p. 207) shows this snippet in use.

```
//
// Method to update a queue attribute
private static async Task UpdateAttribute(
  IAmazonSQS sqsClient, string qUrl, string attribute, string value)
{
  await sqsClient.SetQueueAttributesAsync(qUrl,
    new Dictionary<string, string>{{attribute, value}});
}
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.SQS

Programming elements:

- Namespace Amazon.SQS

  Class AmazonSQSClient

  Class QueueAttributeName
- Namespace Amazon.SQS.Model

  Class GetQueueAttributesResponse

The code

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.SQS;
using Amazon.SQS.Model;

namespace SQSUpdateQueue
{
  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 =
  // Class to update a queue
  class Program
  {
    private const int MaxArgs = 3;
    private const int InvalidArgCount = 2;

    static async Task Main(string[] args)
    {
      // Parse the command line and show help if necessary
      var parsedArgs = CommandLine.Parse(args);
      if(parsedArgs.Count == 0)
      {
        PrintHelp();
        return;
      }
      if((parsedArgs.Count > MaxArgs) || (parsedArgs.Count == InvalidArgCount))
        CommandLine.ErrorExit("\nThe number of command-line arguments is incorrect." +
          "\nRun the command with no arguments to see help.");

      // Get the application arguments from the parsed list
      var qUrl = CommandLine.GetArgument(parsedArgs, null, "-q");
      var attribute = CommandLine.GetArgument(parsedArgs, null, "-a");
      var value = CommandLine.GetArgument(parsedArgs, null, "-v", "--value");

      if(string.IsNullOrEmpty(qUrl))
        CommandLine.ErrorExit("\nYou must supply at least a queue URL." +
          "\nRun the command with no arguments to see help.");

      // Create the Amazon SQS client
```

```
  var sqsClient = new AmazonSQSClient();

  // In the case of one command-line argument, just show the attributes for the queue
  if(parsedArgs.Count == 1)
    await ShowAllAttributes(sqsClient, qUrl);

  // Otherwise, attempt to update the given queue attribute with the given value
  else
  {
    // Check to see if the attribute is valid
    if(ValidAttribute(attribute))
    {
      // Perform the update and then show all the attributes of the queue
      await UpdateAttribute(sqsClient, qUrl, attribute, value);
      await ShowAllAttributes(sqsClient, qUrl);
    }
    else
    {
      Console.WriteLine($"\nThe given attribute name, {attribute}, isn't valid.");
    }
  }
}


//
// Method to show all attributes of a queue
private static async Task ShowAllAttributes(IAmazonSQS sqsClient, string qUrl)
{
  GetQueueAttributesResponse responseGetAtt =
    await sqsClient.GetQueueAttributesAsync(qUrl,
      new List<string>{ QueueAttributeName.All });
  Console.WriteLine($"Queue: {qUrl}");
  foreach(var att in responseGetAtt.Attributes)
    Console.WriteLine($"\t{att.Key}: {att.Value}");
}


//
// Method to check the name of the attribute
private static bool ValidAttribute(string attribute)
{
  var attOk = false;
  var qAttNameType = typeof(QueueAttributeName);
  List<string> qAttNamefields = new List<string>();
  foreach(var field in qAttNameType.GetFields())
   qAttNamefields.Add(field.Name);
  foreach(var name in qAttNamefields)
    if(attribute == name) { attOk = true; break; }
  return attOk;
}


//
// Method to update a queue attribute
private static async Task UpdateAttribute(
  IAmazonSQS sqsClient, string qUrl, string attribute, string value)
{
  await sqsClient.SetQueueAttributesAsync(qUrl,
    new Dictionary<string, string>{{attribute, value}});
}


//
// Command-line help
private static void PrintHelp()
{
```

```
      Console.WriteLine("\nUsage: SQSUpdateQueue -q queue_url [-a attribute -v value]");
      Console.WriteLine("  -q: The URL of the queue you want to update.");
      Console.WriteLine("  -a: The name of the attribute to update.");
      Console.WriteLine("  -v, --value: The value to assign to the attribute.");
    }
  }


  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
=
  // Class that represents a command line on the console or terminal.
  // (This is the same for all examples. When you have seen it once, you can ignore it.)
  static class CommandLine
  {
    //
    // Method to parse a command line of the form: "--key value" or "-k value".
    //
    // Parameters:
    // - args: The command-line arguments passed into the application by the system.
    //
    // Returns:
    // A Dictionary with string Keys and Values.
    //
    // If a key is found without a matching value, Dictionary.Value is set to the key
    //  (including the dashes).
    // If a value is found without a matching key, Dictionary.Key is set to "--NoKeyN",
    //  where "N" represents sequential numbers.
    public static Dictionary<string,string> Parse(string[] args)
    {
      var parsedArgs = new Dictionary<string,string>();
      int i = 0, n = 0;
      while(i < args.Length)
      {
        // If the first argument in this iteration starts with a dash it's an option.
        if(args[i].StartsWith("-"))
        {
          var key = args[i++];
          var value = key;

          // Check to see if there's a value that goes with this option?
          if((i < args.Length) && (!args[i].StartsWith("-"))) value = args[i++];
          parsedArgs.Add(key, value);
        }

        // If the first argument in this iteration doesn't start with a dash, it's a value
        else
        {
          parsedArgs.Add("--NoKey" + n.ToString(), args[i++]);
          n++;
        }
      }

      return parsedArgs;
    }

    //
    // Method to get an argument from the parsed command-line arguments
    //
    // Parameters:
    // - parsedArgs: The Dictionary object returned from the Parse() method (shown above).
    // - defaultValue: The default string to return if the specified key isn't in
parsedArgs.
    // - keys: An array of keys to look for in parsedArgs.
    public static string GetArgument(
      Dictionary<string,string> parsedArgs, string defaultReturn, params string[] keys)
    {
```

```
      string retval = null;
      foreach(var key in keys)
        if(parsedArgs.TryGetValue(key, out retval)) break;
      return retval ?? defaultReturn;
    }

    //
    // Method to exit the application with an error.
    public static void ErrorExit(string msg, int code=1)
    {
      Console.WriteLine("\nError");
      Console.WriteLine(msg);
      Environment.Exit(code);
    }
  }

}
```

Additional considerations

- To update the `RedrivePolicy` attribute, you must quote the entire value and escape the quotes for the key/value pairs, as appropriate for you operating system.

  On Windows, for example, the value is constructed in a manner similar to the following:

```
"{\"deadLetterTargetArn\":\"DEAD_LETTER-QUEUE-ARN\",\"maxReceiveCount\":\"10\"}"
```

Deleting Amazon SQS queues

This example shows you how to use the AWS SDK for .NET to delete an Amazon SQS queue. The application deletes the queue, waits for up to a given amount of time for the queue to be gone, and then shows a list of the remaining queues.

If you don't supply any command-line arguments, the application simply shows a list of the existing queues.

The following sections provide snippets of this example. The complete code for the example (p. 211) is shown after that, and can be built and run as is.

Topics

Delete the queue

The following snippet deletes the queue identified by the given queue URL.

The example at the end of this topic (p. 211) shows this snippet in use.

```
    //
    // Method to delete an SQS queue
    private static async Task DeleteQueue(IAmazonSQS sqsClient, string qUrl)
    {
      Console.WriteLine($"Deleting queue {qUrl}...");
```

```
      await sqsClient.DeleteQueueAsync(qUrl);
      Console.WriteLine($"Queue {qUrl} has been deleted.");
    }
```

Wait for the queue to be gone

The following snippet waits for the deletion process to finish, which might take 60 seconds.

The example at the end of this topic (p. 211) shows this snippet in use.

```
    //
    // Method to wait up to a given number of seconds
    private static async Task Wait(
      IAmazonSQS sqsClient, int numSeconds, string qUrl)
    {
      Console.WriteLine($"Waiting for up to {numSeconds} seconds.");
      Console.WriteLine("Press any key to stop waiting. (Response might be slightly
delayed.)");
      for(int i=0; i<numSeconds; i++)
      {
        Console.Write(".");
        Thread.Sleep(1000);
        if(Console.KeyAvailable) break;

        // Check to see if the queue is gone yet
        var found = false;
        ListQueuesResponse responseList = await sqsClient.ListQueuesAsync("");
        foreach(var url in responseList.QueueUrls)
        {
          if(url == qUrl)
          {
            found = true;
            break;
          }
        }
        if(!found) break;
      }
    }
```

Show a list of existing queues

The following snippet shows a list of the existing queues in the SQS client's region.

The example at the end of this topic (p. 211) shows this snippet in use.

```
    //
    // Method to show a list of the existing queues
    private static async Task ListQueues(IAmazonSQS sqsClient)
    {
      ListQueuesResponse responseList = await sqsClient.ListQueuesAsync("");
      Console.WriteLine("\nList of queues:");
      foreach(var qUrl in responseList.QueueUrls)
        Console.WriteLine($"- {qUrl}");
    }
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.SQS

Programming elements:

- Namespace Amazon.SQS

  Class AmazonSQSClient
- Namespace Amazon.SQS.Model

  Class ListQueuesResponse

The code

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Amazon.SQS;
using Amazon.SQS.Model;

namespace SQSDeleteQueue
{
  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 =
  // Class to update a queue
  class Program
  {
    private const int TimeToWait = 60;

    static async Task Main(string[] args)
    {
      // Create the Amazon SQS client
      var sqsClient = new AmazonSQSClient();

      // If no command-line arguments, just show a list of the queues
      if(args.Length == 0)
      {
        Console.WriteLine("\nUsage: SQSCreateQueue queue_url");
        Console.WriteLine("   queue_url - The URL of the queue you want to delete.");
        Console.WriteLine("\nNo arguments specified.");
        Console.Write("Do you want to see a list of the existing queues? ((y) or n): ");
        var response = Console.ReadLine();
        if((string.IsNullOrEmpty(response)) || (response.ToLower() == "y"))
          await ListQueues(sqsClient);
        return;
      }

      // If given a queue URL, delete that queue
      if(args[0].StartsWith("https://sqs."))
      {
        // Delete the queue
        await DeleteQueue(sqsClient, args[0]);
        // Wait for a little while because it takes a while for the queue to disappear
        await Wait(sqsClient, TimeToWait, args[0]);
        // Show a list of the remaining queues
        await ListQueues(sqsClient);
      }
      else
      {
        Console.WriteLine("The command-line argument isn't a queue URL:");
        Console.WriteLine($"{args[0]}");
      }
    }
```

```
    //
    // Method to delete an SQS queue
    private static async Task DeleteQueue(IAmazonSQS sqsClient, string qUrl)
    {
      Console.WriteLine($"Deleting queue {qUrl}...");
      await sqsClient.DeleteQueueAsync(qUrl);
      Console.WriteLine($"Queue {qUrl} has been deleted.");
    }


    //
    // Method to wait up to a given number of seconds
    private static async Task Wait(
      IAmazonSQS sqsClient, int numSeconds, string qUrl)
    {
      Console.WriteLine($"Waiting for up to {numSeconds} seconds.");
      Console.WriteLine("Press any key to stop waiting. (Response might be slightly
 delayed.)");
      for(int i=0; i<numSeconds; i++)
      {
        Console.Write(".");
        Thread.Sleep(1000);
        if(Console.KeyAvailable) break;

        // Check to see if the queue is gone yet
        var found = false;
        ListQueuesResponse responseList = await sqsClient.ListQueuesAsync("");
        foreach(var url in responseList.QueueUrls)
        {
          if(url == qUrl)
          {
            found = true;
            break;
          }
        }
        if(!found) break;
      }
    }


    //
    // Method to show a list of the existing queues
    private static async Task ListQueues(IAmazonSQS sqsClient)
    {
      ListQueuesResponse responseList = await sqsClient.ListQueuesAsync("");
      Console.WriteLine("\nList of queues:");
      foreach(var qUrl in responseList.QueueUrls)
        Console.WriteLine($"- {qUrl}");
    }
  }
}
```

Additional considerations

- The `DeleteQueueAsync` API call doesn't check to see if the queue you're deleting is being used as a dead-letter queue. A more sophisticated procedure could check for this.

- You can also see the list of queues and the results of this example in the Amazon SQS console.

Sending Amazon SQS messages

This example shows you how to use the AWS SDK for .NET to send messages to an Amazon SQS queue, which you can create programmatically (p. 199) or by using the Amazon SQS console. The application sends a single message to the queue and then a batch of messages. The application then waits for user input, which can be additional messages to send to the queue or a request to exit the application.

This example and the next example about receiving messages (p. 217) can be used together to see message flow in Amazon SQS.

The following sections provide snippets of this example. The complete code for the example (p. 215) is shown after that, and can be built and run as is.

Topics

Send a message

The following snippet sends a message to the queue identified by the given queue URL.

The example at the end of this topic (p. 215) shows this snippet in use.

```
//
// Method to put a message on a queue
// Could be expanded to include message attributes, etc., in a SendMessageRequest
private static async Task SendMessage(
  IAmazonSQS sqsClient, string qUrl, string messageBody)
{
  SendMessageResponse responseSendMsg =
    await sqsClient.SendMessageAsync(qUrl, messageBody);
  Console.WriteLine($"Message added to queue\n  {qUrl}");
  Console.WriteLine($"HttpStatusCode: {responseSendMsg.HttpStatusCode}");
}
```

Send a batch of messages

The following snippet sends a batch of messages to the queue identified by the given queue URL.

The example at the end of this topic (p. 215) shows this snippet in use.

```
//
// Method to put a batch of messages on a queue
// Could be expanded to include message attributes, etc.,
// in the SendMessageBatchRequestEntry objects
private static async Task SendMessageBatch(
  IAmazonSQS sqsClient, string qUrl, List<SendMessageBatchRequestEntry> messages)
{
  Console.WriteLine($"\nSending a batch of messages to queue\n  {qUrl}");
  SendMessageBatchResponse responseSendBatch =
    await sqsClient.SendMessageBatchAsync(qUrl, messages);
  // Could test responseSendBatch.Failed here
  foreach(SendMessageBatchResultEntry entry in responseSendBatch.Successful)
```

```
        Console.WriteLine($"Message {entry.Id} successfully queued.");
    }
```

Delete all messages from the queue

The following snippet deletes all messages from the queue identified by the given queue URL. This is also known as *purging the queue*.

The example at the end of this topic (p. 215) shows this snippet in use.

```
    //
    // Method to delete all messages from the queue
    private static async Task DeleteAllMessages(IAmazonSQS sqsClient, string qUrl)
    {
      Console.WriteLine($"\nPurging messages from queue\n  {qUrl}...");
      PurgeQueueResponse responsePurge = await sqsClient.PurgeQueueAsync(qUrl);
      Console.WriteLine($"HttpStatusCode: {responsePurge.HttpStatusCode}");
    }
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.SQS

Programming elements:

- Namespace Amazon.SQS

  Class AmazonSQSClient
- Namespace Amazon.SQS.Model

  Class PurgeQueueResponse

  Class SendMessageBatchResponse

  Class SendMessageResponse

  Class SendMessageBatchRequestEntry

  Class SendMessageBatchResultEntry

The code

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.SQS;
using Amazon.SQS.Model;

namespace SQSSendMessages
{
  // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 =
```

```
  // Class to send messages to a queue
  class Program
  {
    // Some example messages to send to the queue
    private const string JsonMessage = "{\"product\":[{\"name\":\"Product A\",\"price\":
 \"32\"},{\"name\": \"Product B\",\"price\": \"27\"}]}";
    private const string XmlMessage = "<products><product name=\"Product A\" price=\"32\" /
><product name=\"Product B\" price=\"27\" /></products>";
    private const string CustomMessage = "||product|Product A|32||product|Product B|27||";
    private const string TextMessage = "Just a plain text message.";

    static async Task Main(string[] args)
    {
      // Do some checks on the command-line
      if(args.Length == 0)
      {
        Console.WriteLine("\nUsage: SQSSendMessages queue_url");
        Console.WriteLine("   queue_url - The URL of an existing SQS queue.");
        return;
      }
      if(!args[0].StartsWith("https://sqs."))
      {
        Console.WriteLine("\nThe command-line argument isn't a queue URL:");
        Console.WriteLine($"{args[0]}");
        return;
      }

      // Create the Amazon SQS client
      var sqsClient = new AmazonSQSClient();

      // (could verify that the queue exists)
      // Send some example messages to the given queue
      // A single message
      await SendMessage(sqsClient, args[0], JsonMessage);

      // A batch of messages
      var batchMessages = new List<SendMessageBatchRequestEntry>{
        new SendMessageBatchRequestEntry("xmlMsg", XmlMessage),
        new SendMessageBatchRequestEntry("customeMsg", CustomMessage),
        new SendMessageBatchRequestEntry("textMsg", TextMessage)};
      await SendMessageBatch(sqsClient, args[0], batchMessages);

      // Let the user send their own messages or quit
      await InteractWithUser(sqsClient, args[0]);

      // Delete all messages that are still in the queue
      await DeleteAllMessages(sqsClient, args[0]);
    }


    //
    // Method to put a message on a queue
    // Could be expanded to include message attributes, etc., in a SendMessageRequest
    private static async Task SendMessage(
      IAmazonSQS sqsClient, string qUrl, string messageBody)
    {
      SendMessageResponse responseSendMsg =
        await sqsClient.SendMessageAsync(qUrl, messageBody);
      Console.WriteLine($"Message added to queue\n  {qUrl}");
      Console.WriteLine($"HttpStatusCode: {responseSendMsg.HttpStatusCode}");
    }


    //
    // Method to put a batch of messages on a queue
    // Could be expanded to include message attributes, etc.,
```

```
    // in the SendMessageBatchRequestEntry objects
    private static async Task SendMessageBatch(
      IAmazonSQS sqsClient, string qUrl, List<SendMessageBatchRequestEntry> messages)
    {
      Console.WriteLine($"\nSending a batch of messages to queue\n  {qUrl}");
      SendMessageBatchResponse responseSendBatch =
        await sqsClient.SendMessageBatchAsync(qUrl, messages);
      // Could test responseSendBatch.Failed here
      foreach(SendMessageBatchResultEntry entry in responseSendBatch.Successful)
        Console.WriteLine($"Message {entry.Id} successfully queued.");
    }


    //
    // Method to get input from the user
    // They can provide messages to put in the queue or exit the application
    private static async Task InteractWithUser(IAmazonSQS sqsClient, string qUrl)
    {
      string response;
      while (true)
      {
        // Get the user's input
        Console.WriteLine("\nType a message for the queue or \"exit\" to quit:");
        response = Console.ReadLine();
        if(response.ToLower() == "exit") break;

        // Put the user's message in the queue
        await SendMessage(sqsClient, qUrl, response);
      }
    }


    //
    // Method to delete all messages from the queue
    private static async Task DeleteAllMessages(IAmazonSQS sqsClient, string qUrl)
    {
      Console.WriteLine($"\nPurging messages from queue\n  {qUrl}...");
      PurgeQueueResponse responsePurge = await sqsClient.PurgeQueueAsync(qUrl);
      Console.WriteLine($"HttpStatusCode: {responsePurge.HttpStatusCode}");
    }
  }
}
```

Additional considerations

- For information about various limitations on messages, including the allowed characters, see Quotas related to messages in the Amazon Simple Queue Service Developer Guide.

- Messages stay in queues until they are deleted or the queue is purged. When a message has been received by an application, it won't be visible in the queue even though it still exists in the queue. For more information about visibility timeouts, see Amazon SQS visibility timeout.

- In additional to the message body, you can also add attributes to messages. For more information, see Message metadata.

Receiving Amazon SQS messages

This example shows you how to use the AWS SDK for .NET to receive messages from an Amazon SQS queue, which you can create programmatically (p. 199) or by using the Amazon SQS console. The application reads a single message from the queue, processes the message (in this case, displays the

message body on the console), and then deletes the message from the queue. The application repeats these steps until the user types a key on the keyboard.

This example and the previous example about sending messages (p. 214) can be used together to see message flow in Amazon SQS.

The following sections provide snippets of this example. The complete code for the example (p. 218) is shown after that, and can be built and run as is.

Topics

Receive a message

The following snippet receives a message from the queue identified by the given queue URL.

The example at the end of this topic (p. 218) shows this snippet in use.

```
//
// Method to read a message from the given queue
// In this example, it gets one message at a time
private static async Task<ReceiveMessageResponse> GetMessage(
  IAmazonSQS sqsClient, string qUrl, int waitTime=0)
{
  return await sqsClient.ReceiveMessageAsync(new ReceiveMessageRequest{
    QueueUrl=qUrl,
    MaxNumberOfMessages=MaxMessages,
    WaitTimeSeconds=waitTime
    // (Could also request attributes, set visibility timeout, etc.)
  });
}
```

Delete a message

The following snippet deletes a message from the queue identified by the given queue URL.

The example at the end of this topic (p. 218) shows this snippet in use.

```
//
// Method to delete a message from a queue
private static async Task DeleteMessage(
  IAmazonSQS sqsClient, Message message, string qUrl)
{
  Console.WriteLine($"\nDeleting message {message.MessageId} from queue...");
  await sqsClient.DeleteMessageAsync(qUrl, message.ReceiptHandle);
}
```

Complete code

This section shows relevant references and the complete code for this example.

SDK references

NuGet packages:

- AWSSDK.SQS

Programming elements:

- Namespace Amazon.SQS

  Class AmazonSQSClient
- Namespace Amazon.SQS.Model

  Class ReceiveMessageRequest

  Class ReceiveMessageResponse

The code

```
using System;
using System.Threading.Tasks;
using Amazon.SQS;
using Amazon.SQS.Model;

namespace SQSReceiveMessages
{
  class Program
  {
    private const int MaxMessages = 1;
    private const int WaitTime = 2;
    static async Task Main(string[] args)
    {
      // Do some checks on the command-line
      if(args.Length == 0)
      {
        Console.WriteLine("\nUsage: SQSReceiveMessages queue_url");
        Console.WriteLine("   queue_url - The URL of an existing SQS queue.");
        return;
      }
      if(!args[0].StartsWith("https://sqs."))
      {
        Console.WriteLine("\nThe command-line argument isn't a queue URL:");
        Console.WriteLine($"{args[0]}");
        return;
      }

      // Create the Amazon SQS client
      var sqsClient = new AmazonSQSClient();

      // (could verify that the queue exists)
      // Read messages from the queue and perform appropriate actions
      Console.WriteLine($"Reading messages from queue\n  {args[0]}");
      Console.WriteLine("Press any key to stop. (Response might be slightly delayed.)");
      do
      {
        var msg = await GetMessage(sqsClient, args[0], WaitTime);
        if(msg.Messages.Count != 0)
        {
          if(ProcessMessage(msg.Messages[0]))
            await DeleteMessage(sqsClient, msg.Messages[0], args[0]);
        }
      } while(!Console.KeyAvailable);
    }


    //
    // Method to read a message from the given queue
    // In this example, it gets one message at a time
    private static async Task<ReceiveMessageResponse> GetMessage(
```

```
      IAmazonSQS sqsClient, string qUrl, int waitTime=0)
    {
      return await sqsClient.ReceiveMessageAsync(new ReceiveMessageRequest{
        QueueUrl=qUrl,
        MaxNumberOfMessages=MaxMessages,
        WaitTimeSeconds=waitTime
        // (Could also request attributes, set visibility timeout, etc.)
      });
    }


    //
    // Method to process a message
    // In this example, it simply prints the message
    private static bool ProcessMessage(Message message)
    {
      Console.WriteLine($"\nMessage body of {message.MessageId}:");
      Console.WriteLine($"{message.Body}");
      return true;
    }


    //
    // Method to delete a message from a queue
    private static async Task DeleteMessage(
      IAmazonSQS sqsClient, Message message, string qUrl)
    {
      Console.WriteLine($"\nDeleting message {message.MessageId} from queue...");
      await sqsClient.DeleteMessageAsync(qUrl, message.ReceiptHandle);
    }
  }
}
```

Additional considerations

- To specify long polling, this example uses the `WaitTimeSeconds` property for each call to the `ReceiveMessageAsync` method.

  You can also specify long polling for all messages on a queue by using the `ReceiveMessageWaitTimeSeconds` attribute when creating (p. 199) or updating (p. 205) the queue.

  For information about short polling versus long polling, see Short and long polling in the *Amazon Simple Queue Service Developer Guide*.

- During message processing, you can use the receipt handle to change the message visibility timeout. For information about how to do this, see the `ChangeMessageVisibilityAsync` methods of the AmazonSQSClient class.

- Calling the `DeleteMessageAsync` method unconditionally removes the message from the queue, regardless of the visibility timeout setting.

Additional code examples for the AWS SDK for .NET

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

The examples are divided into the following categories:

Single-service actions

Code excerpts that show you how to call individual service functions.

Single-service scenarios

Code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Cross-service examples

Sample applications that work across multiple AWS services.

Examples

- Single-service actions and scenarios using AWS SDK for .NET (p. 221)
- Cross-service examples using AWS SDK for .NET (p. 312)

Single-service actions and scenarios using AWS SDK for .NET

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

Actions are code excerpts that show you how to call individual service functions.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Services

- CloudWatch examples using AWS SDK for .NET (p. 221)
- CloudWatch Logs examples using AWS SDK for .NET (p. 230)
- Amazon Comprehend examples using AWS SDK for .NET (p. 235)
- IAM examples using AWS SDK for .NET (p. 242)
- AWS KMS examples using AWS SDK for .NET (p. 262)
- Amazon Rekognition examples using AWS SDK for .NET (p. 269)
- Amazon S3 examples using AWS SDK for .NET (p. 284)
- Amazon SNS examples using AWS SDK for .NET (p. 293)
- Amazon SQS examples using AWS SDK for .NET (p. 298)

CloudWatch examples using AWS SDK for .NET

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

Actions are code excerpts that show you how to call individual CloudWatch functions.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple CloudWatch functions.

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

Topics

Actions

Delete alarms

The following code example shows how to delete Amazon CloudWatch alarms.

AWS SDK for .NET

```
   using System;
   using System.Collections.Generic;
   using System.Threading.Tasks;
   using Amazon.CloudWatch;
   using Amazon.CloudWatch.Model;

   /// <summary>
   /// This example shows how to delete Amazon CloudWatch alarms. The example
   /// was created using the AWS SDK for .NET version 3.7 and .NET Core 5.0.
   /// </summary>
   public class DeleteAlarms
   {
       public static async Task Main()
       {
           IAmazonCloudWatch cwClient = new AmazonCloudWatchClient();

           var alarmNames = CreateAlarmNameList();
           await DeleteAlarmsAsyncExample(cwClient, alarmNames);
       }

       /// <summary>
       /// Delete the alarms whose names are listed in the alarmNames parameter.
       /// </summary>
       /// <param name="client">The initialized Amazon CloudWatch client.</param>
       /// <param name="alarmNames">A list of names for the alarms to be
       /// deleted.</param>
       public static async Task DeleteAlarmsAsyncExample(IAmazonCloudWatch client,
List<string> alarmNames)
       {
           var request = new DeleteAlarmsRequest
           {
               AlarmNames = alarmNames,
           };

           try
           {
               var response = await client.DeleteAlarmsAsync(request);

               if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
               {
                   Console.WriteLine("Alarms successfully deleted:");
                   alarmNames
                       .ForEach(name => Console.WriteLine($"{name}"));
               }
           }
           catch (ResourceNotFoundException ex)
           {
               Console.WriteLine($"Error: {ex.Message}");
```

```
            }
        }

        /// <summary>
        /// Defines and returns the list of alarm names to delete.
        /// </summary>
        /// <returns>A list of alarm names.</returns>
        public static List<string> CreateAlarmNameList()
        {
            // The list of alarm names passed to DeleteAlarmsAsync
            // can contain up to 100 alarm names.
            var theList = new List<string>
            {
                "ALARM_NAME_1",
                "ALARM_NAME_2",
            };

            return theList;
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see DeleteAlarms in *AWS SDK for .NET API Reference*.

Describe alarm history

The following code example shows how to describe Amazon CloudWatch alarm history.

AWS SDK for .NET

Get a list of CloudWatch alarms, then retrieve the history for each alarm.

```
using System;
using System.Threading.Tasks;
using Amazon.CloudWatch;
using Amazon.CloudWatch.Model;

/// <summary>
/// This example retrieves a list of Amazon CloudWatch alarms and, for
/// each one, displays its history. The example was created using the
/// AWS SDK for .NET 3.7 and .NET Core 5.0.
/// </summary>
public class DescribeAlarmHistories
{
    /// <summary>
    /// Retrieves a list of alarms and then passes each name to the
    /// DescribeAlarmHistoriesAsync method to retrieve its history.
    /// </summary>
    public static async Task Main()
    {
        IAmazonCloudWatch cwClient = new AmazonCloudWatchClient();
        var response = await cwClient.DescribeAlarmsAsync();

        foreach (var alarm in response.MetricAlarms)
        {
            await DescribeAlarmHistoriesAsync(cwClient, alarm.AlarmName);
        }
    }

    /// <summary>
    /// Retrieves the CloudWatch alarm history for the alarm name passed
```

```
        /// to the method.
        /// </summary>
        /// <param name="client">An initialized CloudWatch client object.</param>
        /// <param name="alarmName">The CloudWatch alarm for which to retrieve
        /// history information.</param>
        public static async Task DescribeAlarmHistoriesAsync(IAmazonCloudWatch client,
string alarmName)
        {
            var request = new DescribeAlarmHistoryRequest
            {
                AlarmName = alarmName,
                EndDateUtc = DateTime.Today,
                HistoryItemType = HistoryItemType.Action,
                MaxRecords = 1,
                StartDateUtc = DateTime.Today.Subtract(TimeSpan.FromDays(30)),
            };

            var response = new DescribeAlarmHistoryResponse();

            do
            {
                response = await client.DescribeAlarmHistoryAsync(request);

                foreach (var item in response.AlarmHistoryItems)
                {
                    Console.WriteLine(item.AlarmName);
                    Console.WriteLine(item.HistorySummary);
                    Console.WriteLine();
                }

                request.NextToken = response.NextToken;
            }
            while (!string.IsNullOrEmpty(response.NextToken));
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see DescribeAlarmHistory in *AWS SDK for .NET API Reference*.

Disable alarm actions

The following code example shows how to disable Amazon CloudWatch alarm actions.

AWS SDK for .NET

```
    using System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using Amazon.CloudWatch;
    using Amazon.CloudWatch.Model;

    /// <summary>
    /// This example shows how to disable the Amazon CloudWatch actions for
    /// one or more CloudWatch alarms. The example was created using the
    /// AWS SDK for .NET version 3.7 and .NET Core 5.0.
    /// </summary>
    public class DisableAlarmActions
    {
        public static async Task Main()
        {
```

```
            IAmazonCloudWatch cwClient = new AmazonCloudWatchClient();
            var alarmNames = new List<string>
            {
                "ALARM_NAME",
                "ALARM_NAME_2",
            };

            var success = await DisableAlarmsActionsAsync(cwClient, alarmNames);

            if (success)
            {
                Console.WriteLine("Alarm action(s) successfully disabled.");
            }
            else
            {
                Console.WriteLine("Alarm action(s) were not disabled.")
            }
        }

        /// <summary>
        /// Disable the actions for the list of CloudWatch alarm names passed
        /// in the alarmNames parameter.
        /// </summary>
        /// <param name="client">An initialized CloudWatch client object.</param>
        /// <param name="alarmNames">The list of CloudWatch alarms to disable.</param>
        /// <returns>A Boolean value indicating the success of the call.</returns>
        public static async Task<bool> DisableAlarmsActionsAsync(
            IAmazonCloudWatch client,
            List<string> alarmNames)
        {
            var request = new DisableAlarmActionsRequest
            {
                AlarmNames = alarmNames,
            };

            var response = await client.DisableAlarmActionsAsync(request);

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see DisableAlarmActions in *AWS SDK for .NET API Reference*.

Enable alarm actions

The following code example shows how to enable Amazon CloudWatch alarm actions.

AWS SDK for .NET

```
    using System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using Amazon.CloudWatch;
    using Amazon.CloudWatch.Model;

    /// <summary>
    /// This example shows how to enable the Amazon CloudWatch actions for
    /// one or more CloudWatch alarms. The example was created using the
    /// AWS SDK for .NET version 3.7 and .NET Core 5.0.
```

```
    /// </summary>
    public class EnableAlarmActions
    {
        public static async Task Main()
        {
            IAmazonCloudWatch cwClient = new AmazonCloudWatchClient();
            var alarmNames = new List<string>
            {
                "ALARM_NAME",
                "ALARM_NAME_2",
            };

            var success = await EnableAlarmActionsAsync(cwClient, alarmNames);

            if (success)
            {
                Console.WriteLine("Alarm action(s) successfully enabled.");
            }
            else
            {
                Console.WriteLine("Alarm action(s) were not enabled.")
            }
        }

        /// <summary>
        /// Enable the actions for the list of CloudWatch alarm names passed
        /// in the alarmNames parameter.
        /// </summary>
        /// <param name="client">An initialized CloudWatch client object.</param>
        /// <param name="alarmNames">The list of CloudWatch alarms to enable.</param>
        /// <returns>A Boolean value indicating the success of the call.</returns>
        public static async Task<bool> EnableAlarmActionsAsync(IAmazonCloudWatch
client, List<string> alarmNames)
        {
            var request = new EnableAlarmActionsRequest
            {
                AlarmNames = alarmNames,
            };

            var response = await client.EnableAlarmActionsAsync(request);

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see EnableAlarmActions in *AWS SDK for .NET API Reference*.

Get dashboard details

The following code example shows how to get Amazon CloudWatch dashboard details.

AWS SDK for .NET

```
    using System;
    using System.Threading.Tasks;
    using Amazon.CloudWatch;
    using Amazon.CloudWatch.Model;

    /// <summary>
```

```
    /// This example shows how to retrieve the details of an Amazon CloudWatch
    /// dashboard. The return value from the call to GetDashboard is a json
    /// object representing the widgets in the dashboard. The example was
    /// created using the AWS SDK for .NET version 3.7 and .NET Core 5.0.
    /// </summary>
    public class GetDashboard
    {
        public static async Task Main()
        {
            IAmazonCloudWatch cwClient = new AmazonCloudWatchClient();
            string dashboardName = "CloudWatch-Default";

            var body = await GetDashboardAsync(cwClient, dashboardName);

            Console.WriteLine(body);
        }

        /// <summary>
        /// Get the json that represents the dashboard.
        /// </summary>
        /// <param name="client">An initialized CloudWatch client.</param>
        /// <param name="dashboardName">The name of the dashboard.</param>
        /// <returns>The string containing the json value describing the
        /// contents and layout of the CloudWatch dashboard.</returns>
        public static async Task<string> GetDashboardAsync(IAmazonCloudWatch client,
string dashboardName)
        {

            var request = new GetDashboardRequest
            {
                DashboardName = dashboardName,
            };

            var response = await client.GetDashboardAsync(request);

            return response.DashboardBody;
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see GetDashboard in *AWS SDK for .NET API Reference*.

List dashboards

The following code example shows how to list Amazon CloudWatch dashboards.

AWS SDK for .NET

```
    using System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using Amazon.CloudWatch;
    using Amazon.CloudWatch.Model;

    /// <summary>
    /// Shows how to retrieve a list of Amazon CloudWatch dashboards. This
    /// example was written using AWSSDK for .NET version 3.7 and .NET Core 5.0.
    /// </summary>
    public class ListDashboards
    {
```

```
        public static async Task Main()
        {
            IAmazonCloudWatch cwClient = new AmazonCloudWatchClient();
            var dashboards = await ListDashboardsAsync(cwClient);

            DisplayDashboardList(dashboards);
        }

        /// <summary>
        /// Get the list of available dashboards.
        /// </summary>
        /// <param name="client">The initialized CloudWatch client used to
        /// retrieve a list of defined dashboards.</param>
        /// <returns>A list of DashboardEntry objects.</returns>
        public static async Task<List<DashboardEntry>>
ListDashboardsAsync(IAmazonCloudWatch client)
        {
            var response = await client.ListDashboardsAsync(new
ListDashboardsRequest());
            return response.DashboardEntries;
        }

        /// <summary>
        /// Displays the name of each CloudWatch Dashboard in the list passed
        /// to the method.
        /// </summary>
        /// <param name="dashboards">A list of DashboardEntry objects.</param>
        public static void DisplayDashboardList(List<DashboardEntry> dashboards)
        {
            if (dashboards.Count > 0)
            {
                Console.WriteLine("The following dashboards are defined:");
                foreach (var dashboard in dashboards)
                {
                    Console.WriteLine($"Name: {dashboard.DashboardName} Last modified:
{dashboard.LastModified}");
                }
            }
            else
            {
                Console.WriteLine("No dashboards found.");
            }
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see ListDashboards in *AWS SDK for .NET API Reference*.

List metrics

The following code example shows how to list Amazon CloudWatch metrics.

AWS SDK for .NET

```
    using System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using Amazon.CloudWatch;
    using Amazon.CloudWatch.Model;
```

```
/// <summary>
/// This example demonstrates how to list metrics for Amazon CloudWatch.
/// The example was created using the AWS SDK for .NET version 3.7 and
/// .NET Core 5.0.
/// </summary>
public class ListMetrics
{
    public static async Task Main()
    {
        IAmazonCloudWatch cwClient = new AmazonCloudWatchClient();

        var filter = new DimensionFilter
        {
            Name = "InstanceType",
            Value = "t1.micro",
        };
        string metricName = "CPUUtilization";
        string namespaceName = "AWS/EC2";

        await ListMetricsAsync(cwClient, filter, metricName, namespaceName);
    }

    /// <summary>
    /// Retrieve CloudWatch metrics using the supplied filter, metrics name,
    /// and namespace.
    /// </summary>
    /// <param name="client">An initialized CloudWatch client.</param>
    /// <param name="filter">The filter to apply in retrieving metrics.</param>
    /// <param name="metricName">The metric name for which to retrieve
    /// information.</param>
    /// <param name="nameSpaceName">The name of the namespace from which
    /// to retrieve metric information.</param>
    public static async Task ListMetricsAsync(
        IAmazonCloudWatch client,
        DimensionFilter filter,
        string metricName,
        string nameSpaceName)
    {
        var request = new ListMetricsRequest
        {
            Dimensions = new List<DimensionFilter>() { filter },
            MetricName = metricName,
            Namespace = nameSpaceName,
        };

        var response = new ListMetricsResponse();
        do
        {
            response = await client.ListMetricsAsync(request);

            if (response.Metrics.Count > 0)
            {
                foreach (var metric in response.Metrics)
                {
                    Console.WriteLine(metric.MetricName +
                      " (" + metric.Namespace + ")");

                    foreach (var dimension in metric.Dimensions)
                    {
                        Console.WriteLine("  " + dimension.Name + ": "
                          + dimension.Value);
                    }
                }
            }
            else
            {
```

```
                    Console.WriteLine("No metrics found.");
                }

                request.NextToken = response.NextToken;
            }
            while (!string.IsNullOrEmpty(response.NextToken));
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see ListMetrics in *AWS SDK for .NET API Reference*.

CloudWatch Logs examples using AWS SDK for .NET

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

Actions are code excerpts that show you how to call individual CloudWatch Logs functions.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple CloudWatch Logs functions.

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

Topics

Actions

Associate an AWS KMS key to log group

The following code example shows how to associate an AWS KMS key with an existing CloudWatch Logs log group.

AWS SDK for .NET

```
        public static async Task Main()
        {
            // This client object will be associated with the same AWS Region
            // as the default user on this system. If you need to use a
            // different AWS Region, pass it as a parameter to the client
            // constructor.
            var client = new AmazonCloudWatchLogsClient();

            string kmsKeyId = "arn:aws:kms:us-west-2:<account-
number>:key/7c9eccc2-38cb-4c4f-9db3-766ee8dd3ad4";
            string groupName = "cloudwatchlogs-example-loggroup";

            var request = new AssociateKmsKeyRequest
            {
                KmsKeyId = kmsKeyId,
                LogGroupName = groupName,
            };
```

```
            var response = await client.AssociateKmsKeyAsync(request);

            if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
            {
                Console.WriteLine($"Successfully associated KMS key ID: {kmsKeyId} with
log group: {groupName}.");
            }
            else
            {
                Console.WriteLine("Could not make the association between: {kmsKeyId}
and {groupName}.");
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see AssociateKmsKey in *AWS SDK for .NET API Reference*.

Cancel an export task

The following code example shows how to cancel an existing CloudWatch Logs export task.

AWS SDK for .NET

```
        public static async Task Main()
        {
            // This client object will be associated with the same AWS Region
            // as the default user on this system. If you need to use a
            // different AWS Region, pass it as a parameter to the client
            // constructor.
            var client = new AmazonCloudWatchLogsClient();
            string taskId = "exampleTaskId";

            var request = new CancelExportTaskRequest
            {
                TaskId = taskId,
            };

            var response = await client.CancelExportTaskAsync(request);

            if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
            {
                Console.WriteLine($"{taskId} successfully canceled.");
            }
            else
            {
                Console.WriteLine($"{taskId} could not be canceled.");
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see CancelExportTask in *AWS SDK for .NET API Reference*.

Create a log group

The following code example shows how to create a new CloudWatch Logs log group.

AWS SDK for .NET

```
        public static async Task Main()
        {
            // This client object will be associated with the same AWS Region
            // as the default user on this system. If you need to use a
            // different AWS Region, pass it as a parameter to the client
            // constructor.
            var client = new AmazonCloudWatchLogsClient();

            string logGroupName = "cloudwatchlogs-example-loggroup";

            var request = new CreateLogGroupRequest
            {
                LogGroupName = logGroupName,
            };

            var response = await client.CreateLogGroupAsync(request);

            if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
            {
                Console.WriteLine($"Successfully create log group with ID:
{logGroupName}.");
            }
            else
            {
                Console.WriteLine("Could not create log group.");
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see CreateLogGroup in *AWS SDK for .NET API Reference*.

Create a new log stream

The following code example shows how to create a new CloudWatch Logs log stream.

AWS SDK for .NET

```
        public static async Task Main()
        {
            // This client object will be associated with the same AWS Region
            // as the default user on this system. If you need to use a
            // different AWS Region, pass it as a parameter to the client
            // constructor.
            var client = new AmazonCloudWatchLogsClient();
            string logGroupName = "cloudwatchlogs-example-loggroup";
            string logStreamName = "cloudwatchlogs-example-logstream";

            var request = new CreateLogStreamRequest
            {
                LogGroupName = logGroupName,
                LogStreamName = logStreamName,
            };

            var response = await client.CreateLogStreamAsync(request);

            if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
```

```
            {
                Console.WriteLine($"{logStreamName} successfully created for
{logGroupName}.");
            }
            else
            {
                Console.WriteLine("Could not create stream.");
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see CreateLogStream in *AWS SDK for .NET API Reference*.

Delete a log group

The following code example shows how to delete an existing CloudWatch Logs log group.

AWS SDK for .NET

```
        public static async Task Main()
        {
            var client = new AmazonCloudWatchLogsClient();
            string logGroupName = "cloudwatchlogs-example-loggroup";

            var request = new DeleteLogGroupRequest
            {
                LogGroupName = logGroupName,
            };

            var response = await client.DeleteLogGroupAsync(request);

            if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
            {
                Console.WriteLine($"Successfully deleted CloudWatch log group,
{logGroupName}.");
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see DeleteLogGroup in *AWS SDK for .NET API Reference*.

Describe export tasks

The following code example shows how to describe CloudWatch Logs export tasks.

AWS SDK for .NET

```
        public static async Task Main()
        {
            // This client object will be associated with the same AWS Region
            // as the default user on this system. If you need to use a
            // different AWS Region, pass it as a parameter to the client
            // constructor.
            var client = new AmazonCloudWatchLogsClient();
```

```
            var request = new DescribeExportTasksRequest
            {
                Limit = 5,
            };

            var response = new DescribeExportTasksResponse();

            do
            {
                response = await client.DescribeExportTasksAsync(request);
                response.ExportTasks.ForEach(t =>
                {
                   Console.WriteLine($"{t.TaskName} with ID: {t.TaskId} has status:
{t.Status}");
                });
            }
            while (response.NextToken is not null);
        }
```

- Find instructions and more code on GitHub.
- For API details, see DescribeExportTasks in *AWS SDK for .NET API Reference*.

Describe log groups

The following code example shows how to describe CloudWatch Logs log groups.

AWS SDK for .NET

```
        public static async Task Main()
        {
            // Creates a CloudWatch Logs client using the default
            // user. If you need to work with resources in another
            // AWS Region than the one defined for the default user,
            // pass the AWS Region as a parameter to the client constructor.
            var client = new AmazonCloudWatchLogsClient();

            var request = new DescribeLogGroupsRequest
            {
                Limit = 5,
            };

            var response = await client.DescribeLogGroupsAsync(request);

            if (response.LogGroups.Count > 0)
            {
                do
                {
                    response.LogGroups.ForEach(lg =>
                    {
                        Console.WriteLine($"{lg.LogGroupName} is associated with the
key: {lg.KmsKeyId}.");
                        Console.WriteLine($"Created on: {lg.CreationTime.Date.Date}");
                        Console.WriteLine($"Date for this group will be stored for:
{lg.RetentionInDays} days.\n");
                    });
                }
                while (response.NextToken is not null);
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see DescribeLogGroups in *AWS SDK for .NET API Reference*.

Amazon Comprehend examples using AWS SDK for .NET

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

Actions are code excerpts that show you how to call individual Amazon Comprehend functions.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple Amazon Comprehend functions.

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

Topics

Actions

Detect entities in a document

The following code example shows how to detect entities in a document with Amazon Comprehend.

AWS SDK for .NET

```
using System;
using System.Threading.Tasks;
using Amazon.Comprehend;
using Amazon.Comprehend.Model;

/// <summary>
/// This example shows how to use the AmazonComprehend service detect any
/// entities in submitted text. This example was created using the AWS SDK
/// for .NET 3.7 and .NET Core 5.0.
/// </summary>
public static class DetectEntities
{
    /// <summary>
    /// The main method calls the DetectEntitiesAsync method to find any
    /// entities in the sample code.
    /// </summary>
    public static async Task Main()
    {
        string text = "It is raining today in Seattle";

        var comprehendClient = new AmazonComprehendClient();

        Console.WriteLine("Calling DetectEntities\n");
        var detectEntitiesRequest = new DetectEntitiesRequest()
        {
            Text = text,
            LanguageCode = "en",
        };
```

```
            var detectEntitiesResponse = await
comprehendClient.DetectEntitiesAsync(detectEntitiesRequest);

            foreach (var e in detectEntitiesResponse.Entities)
            {
                Console.WriteLine($"Text: {e.Text}, Type: {e.Type}, Score: {e.Score},
BeginOffset: {e.BeginOffset}, EndOffset: {e.EndOffset}");
            }

            Console.WriteLine("Done");
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see DetectEntities in *AWS SDK for .NET API Reference*.

Detect key phrases in a document

The following code example shows how to detect key phrases in a document with Amazon Comprehend.

AWS SDK for .NET

```
    using System;
    using System.Threading.Tasks;
    using Amazon.Comprehend;
    using Amazon.Comprehend.Model;

    /// <summary>
    /// This example shows how to use the Amazon Comprehend service to
    /// search text for key phrases. It was created using the AWS SDK for
    /// .NET version 3.7 and .NET Core 5.0.
    /// </summary>
    public static class DetectKeyPhrase
    {
        /// <summary>
        /// This method calls the Amazon Comprehend method DetectKeyPhrasesAsync
        /// to detect any key phrases in the sample text.
        /// </summary>
        public static async Task Main()
        {
            string text = "It is raining today in Seattle";

            var comprehendClient = new
AmazonComprehendClient(Amazon.RegionEndpoint.USWest2);

            // Call DetectKeyPhrases API
            Console.WriteLine("Calling DetectKeyPhrases");
            var detectKeyPhrasesRequest = new DetectKeyPhrasesRequest()
            {
                Text = text,
                LanguageCode = "en",
            };
            var detectKeyPhrasesResponse = await
comprehendClient.DetectKeyPhrasesAsync(detectKeyPhrasesRequest);
            foreach (var kp in detectKeyPhrasesResponse.KeyPhrases)
            {
                Console.WriteLine($"Text: {kp.Text}, Score: {kp.Score}, BeginOffset:
{kp.BeginOffset}, EndOffset: {kp.EndOffset}");
            }
```

```
            Console.WriteLine("Done");
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see DetectKeyPhrases in *AWS SDK for .NET API Reference*.

Detect personally identifiable information in a document

The following code example shows how to detect personally identifiable information (PII) in a document with Amazon Comprehend.

AWS SDK for .NET

```
    using System;
    using System.Threading.Tasks;
    using Amazon.Comprehend;
    using Amazon.Comprehend.Model;

    /// <summary>
    /// This example shows how to use the Amazon Comprehend service to find
    /// personally identifiable information (PII) within text submitted to the
    /// DetectPiiEntitiesAsync method. The example was created using the AWS
    /// SDK for .NET version 3.7 and .NET Core 5.0.
    /// </summary>
    public class DetectingPII
    {
        /// <summary>
        /// This method calls the DetectPiiEntitiesAsync method to locate any
        /// personally dientifiable information within the supplied text.
        /// </summary>
        public static async Task Main()
        {
            var comprehendClient = new AmazonComprehendClient();
            var text = @"Hello Paul Santos. The latest statement for your
                         credit card account 1111-0000-1111-0000 was
                         mailed to 123 Any Street, Seattle, WA 98109.";

            var request = new DetectPiiEntitiesRequest
            {
                Text = text,
                LanguageCode = "EN",
            };

            var response = await comprehendClient.DetectPiiEntitiesAsync(request);

            if (response.Entities.Count > 0)
            {
                foreach (var entity in response.Entities)
                {
                    var entityValue = text.Substring(entity.BeginOffset,
entity.EndOffset - entity.BeginOffset);
                    Console.WriteLine($"{entity.Type}: {entityValue}");
                }
            }
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see DetectPiiEntities in *AWS SDK for .NET API Reference*.

Detect syntactical elements of a document

The following code example shows how to detect syntactial elements of a document with Amazon Comprehend.

AWS SDK for .NET

```
   using System;
   using System.Threading.Tasks;
   using Amazon.Comprehend;
   using Amazon.Comprehend.Model;

   /// <summary>
   /// This example shows how to use Amazon Comprehend to detect syntax
   /// elements by calling the DetectSyntaxAsync method. This example was
   /// created using the AWS SDK for .NET 3.7 and .NET Core 5.0.
   /// </summary>
   public class DetectingSyntax
   {
       /// <summary>
       /// This method calls DetectSynaxAsync to identify the syntax elements
       /// in the sample text.
       /// </summary>
       public static async Task Main()
       {
           string text = "It is raining today in Seattle";

           var comprehendClient = new AmazonComprehendClient();

           // Call DetectSyntax API
           Console.WriteLine("Calling DetectSyntaxAsync\n");
           var detectSyntaxRequest = new DetectSyntaxRequest()
           {
               Text = text,
               LanguageCode = "en",
           };
           DetectSyntaxResponse detectSyntaxResponse = await
comprehendClient.DetectSyntaxAsync(detectSyntaxRequest);
           foreach (SyntaxToken s in detectSyntaxResponse.SyntaxTokens)
           {
               Console.WriteLine($"Text: {s.Text}, PartOfSpeech: {s.PartOfSpeech.Tag},
BeginOffset: {s.BeginOffset}, EndOffset: {s.EndOffset}");
           }

           Console.WriteLine("Done");
       }
   }
```

- Find instructions and more code on GitHub.
- For API details, see DetectSyntax in *AWS SDK for .NET API Reference*.

Detect the dominant language in a document

The following code example shows how to detect the dominant language in a document with Amazon Comprehend.

AWS SDK for .NET

```
   using System;
   using System.Threading.Tasks;
   using Amazon.Comprehend;
   using Amazon.Comprehend.Model;

   /// <summary>
   /// This example calls the Amazon Comprehend service to determine the
   /// dominant language. The example was created using the AWS SDK for .NET
   /// 3.7 and .NET Core 5.0.
   /// </summary>
   public static class DetectDominantLanguage
   {
       /// <summary>
       /// Calls Amazon Comprehend to determine the dominant language used in
       /// the sample text.
       /// </summary>
       public static async Task Main()
       {
           string text = "It is raining today in Seattle.";

           var comprehendClient = new
AmazonComprehendClient(Amazon.RegionEndpoint.USWest2);

           Console.WriteLine("Calling DetectDominantLanguage\n");
           var detectDominantLanguageRequest = new DetectDominantLanguageRequest()
           {
               Text = text,
           };

           var detectDominantLanguageResponse = await
comprehendClient.DetectDominantLanguageAsync(detectDominantLanguageRequest);
           foreach (var dl in detectDominantLanguageResponse.Languages)
           {
               Console.WriteLine($"Language Code: {dl.LanguageCode}, Score:
{dl.Score}");
           }

           Console.WriteLine("Done");
       }
   }
```

- Find instructions and more code on GitHub.
- For API details, see DetectDominantLanguage in *AWS SDK for .NET API Reference*.

Detect the sentiment of a document

The following code example shows how to detect the sentiment of a document with Amazon Comprehend.

AWS SDK for .NET

```
   using System;
   using System.Threading.Tasks;
   using Amazon.Comprehend;
   using Amazon.Comprehend.Model;
```

```
    /// <summary>
    /// This example shows how to detect the overall sentiment of the supplied
    /// text using the Amazon Comprehend service. The example was writing using
    /// the AWS SDK for .NET version 3.7 and .NET Core 5.0.
    /// </summary>
    public static class DetectSentiment
    {
        /// <summary>
        /// This method calls the DetetectSentimentAsync method to analyze the
        /// supplied text and determine the overal sentiment.
        /// </summary>
        public static async Task Main()
        {
            string text = "It is raining today in Seattle";

            var comprehendClient = new
AmazonComprehendClient(Amazon.RegionEndpoint.USWest2);

            // Call DetectKeyPhrases API
            Console.WriteLine("Calling DetectSentiment");
            var detectSentimentRequest = new DetectSentimentRequest()
            {
                Text = text,
                LanguageCode = "en",
            };
            var detectSentimentResponse = await
comprehendClient.DetectSentimentAsync(detectSentimentRequest);
            Console.WriteLine($"Sentiment: {detectSentimentResponse.Sentiment}");
            Console.WriteLine("Done");
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see DetectSentiment in *AWS SDK for .NET API Reference*.

Start a topic modeling job

The following code example shows how to start an Amazon Comprehend topic modeling job.

AWS SDK for .NET

```
    using System;
    using System.Threading.Tasks;
    using Amazon.Comprehend;
    using Amazon.Comprehend.Model;

    /// <summary>
    /// This example scans the documents in an Amazon Simple Storage Service
    /// (Amazon S3) bucket and analyzes it for topics. The results are stored
    /// in another bucket and then the resulting job properties are displayed
    /// on the screen. This example was created using the AWS SDK for .NEt
    /// version 3.7 and .NET Core version 5.0.
    /// </summary>
    public static class TopicModeling
    {
        /// <summary>
        /// This methos calls a topic detection job by calling the Amazon
        /// Comprehend StartTopicsDetectionJobRequest.
        /// </summary>
        public static async Task Main()
```

```
        {
            var comprehendClient = new AmazonComprehendClient();

            string inputS3Uri = "s3://input bucket/input path";
            InputFormat inputDocFormat = InputFormat.ONE_DOC_PER_FILE;
            string outputS3Uri = "s3://output bucket/output path";
            string dataAccessRoleArn = "arn:aws:iam::account ID:role/data access role";
            int numberOfTopics = 10;

            var startTopicsDetectionJobRequest = new StartTopicsDetectionJobRequest()
            {
                InputDataConfig = new InputDataConfig()
                {
                    S3Uri = inputS3Uri,
                    InputFormat = inputDocFormat,
                },
                OutputDataConfig = new OutputDataConfig()
                {
                    S3Uri = outputS3Uri,
                },
                DataAccessRoleArn = dataAccessRoleArn,
                NumberOfTopics = numberOfTopics,
            };

            var startTopicsDetectionJobResponse = await
comprehendClient.StartTopicsDetectionJobAsync(startTopicsDetectionJobRequest);

            var jobId = startTopicsDetectionJobResponse.JobId;
            Console.WriteLine("JobId: " + jobId);

            var describeTopicsDetectionJobRequest = new
DescribeTopicsDetectionJobRequest()
            {
                JobId = jobId,
            };

            var describeTopicsDetectionJobResponse = await
comprehendClient.DescribeTopicsDetectionJobAsync(describeTopicsDetectionJobRequest);

PrintJobProperties(describeTopicsDetectionJobResponse.TopicsDetectionJobProperties);

            var listTopicsDetectionJobsResponse = await
comprehendClient.ListTopicsDetectionJobsAsync(new ListTopicsDetectionJobsRequest());
            foreach (var props in
listTopicsDetectionJobsResponse.TopicsDetectionJobPropertiesList)
            {
                PrintJobProperties(props);
            }
        }

        /// <summary>
        /// This method is a helper method that displays the job properties
        /// from the call to StartTopicsDetectionJobRequest.
        /// </summary>
        /// <param name="props">A list of properties from the call to
        /// StartTopicsDetectionJobRequest.</param>
        private static void PrintJobProperties(TopicsDetectionJobProperties props)
        {
            Console.WriteLine($"JobId: {props.JobId}, JobName: {props.JobName},
JobStatus: {props.JobStatus}");
            Console.WriteLine($"NumberOfTopics: {props.NumberOfTopics}\nInputS3Uri:
{props.InputDataConfig.S3Uri}");
            Console.WriteLine($"InputFormat: {props.InputDataConfig.InputFormat},
OutputS3Uri: {props.OutputDataConfig.S3Uri}");
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see StartTopicsDetectionJob in *AWS SDK for .NET API Reference*.

IAM examples using AWS SDK for .NET

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

Actions are code excerpts that show you how to call individual IAM functions.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple IAM functions.

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

Topics

Actions

Attach a policy to a role

The following code example shows how to attach an IAM policy to a role.

AWS SDK for .NET

```
        /// <summary>
        /// Attach the policy to the role so that the user can assume it.
        /// </summary>
        /// <param name="client">The initialized IAM client object.</param>
        /// <param name="policyArn">The ARN of the policy to attach.</param>
        /// <param name="roleName">The name of the role to attach the policy to.</
param>
        public static async Task AttachRoleAsync(
            AmazonIdentityManagementServiceClient client,
            string policyArn,
            string roleName)
        {
            var request = new AttachRolePolicyRequest
            {
                PolicyArn = policyArn,
                RoleName = roleName,
            };

            var response = await client.AttachRolePolicyAsync(request);

            if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
            {
                Console.WriteLine("Successfully attached the policy to the role.");
            }
            else
            {
```

```
                Console.WriteLine("Could not attach the policy.");
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see AttachRolePolicy in *AWS SDK for .NET API Reference*.

Create a policy

The following code example shows how to create an IAM policy.

AWS SDK for .NET

```
        /// <summary>
        /// Create a policy to allow a user to list the buckets in an account.
        /// </summary>
        /// <param name="client">The initialized IAM client object.</param>
        /// <param name="policyName">The name of the poicy to create.</param>
        /// <param name="policyDocument">The permissions policy document.</param>
        /// <returns>The newly created ManagedPolicy object.</returns>
        public static async Task<ManagedPolicy> CreatePolicyAsync(
            AmazonIdentityManagementServiceClient client,
            string policyName,
            string policyDocument)
        {
            var request = new CreatePolicyRequest
            {
                PolicyName = policyName,
                PolicyDocument = policyDocument,
            };

            var response = await client.CreatePolicyAsync(request);

            return response.Policy;
        }
```

- Find instructions and more code on GitHub.
- For API details, see CreatePolicy in *AWS SDK for .NET API Reference*.

Create a role

The following code example shows how to create an IAM role.

AWS SDK for .NET

```
        /// <summary>
        /// Create a new IAM role which we can attach to a user.
        /// </summary>
        /// <param name="client">The initialized IAM client object.</param>
        /// <param name="roleName">The name of the IAM role to create.</param>
        /// <param name="rolePermissions">The permissions which the role will have.</
param>
        /// <returns>A Role object representing the newly created role.</returns>
```

```
public static async Task<Role> CreateRoleAsync(
    AmazonIdentityManagementServiceClient client,
    string roleName,
    string rolePermissions)
{
    var request = new CreateRoleRequest
    {
        RoleName = roleName,
        AssumeRolePolicyDocument = rolePermissions,
    };

    var response = await client.CreateRoleAsync(request);

    return response.Role;
}
```

- Find instructions and more code on GitHub.
- For API details, see CreateRole in *AWS SDK for .NET API Reference*.

Create a user

The following code example shows how to create an IAM user.

AWS SDK for .NET

```
/// <summary>
/// Create a new IAM user.
/// </summary>
/// <param name="client">The initialized IAM client object.</param>
/// <param name="userName">A string representing the user name of the
/// new user.</param>
/// <returns>The newly created user.</returns>
public static async Task<User> CreateUserAsync(
    AmazonIdentityManagementServiceClient client,
    string userName)
{
    var request = new CreateUserRequest
    {
        UserName = userName,
    };

    var response = await client.CreateUserAsync(request);

    return response.User;
}
```

- Find instructions and more code on GitHub.
- For API details, see CreateUser in *AWS SDK for .NET API Reference*.

Create an access key

The following code example shows how to create an IAM access key.

AWS SDK for .NET

```
        /// <summary>
        /// Create a new AccessKey for the user.
        /// </summary>
        /// <param name="client">The initialized IAM client object.</param>
        /// <param name="userName">The name of the user for whom to create the key.</
param>
        /// <returns>A new IAM access key for the user.</returns>
        public static async Task<AccessKey> CreateAccessKeyAsync(
            AmazonIdentityManagementServiceClient client,
            string userName)
        {
            var request = new CreateAccessKeyRequest
            {
                UserName = userName,
            };

            var response = await client.CreateAccessKeyAsync(request);

            if (response.AccessKey is not null)
            {
                Console.WriteLine($"Successfully created Access Key for {userName}.");
            }

            return response.AccessKey;
        }
```

- Find instructions and more code on GitHub.
- For API details, see CreateAccessKey in *AWS SDK for .NET API Reference*.

Delete a role policy

The following code example shows how to delete an IAM role policy.

AWS SDK for .NET

```
    using System;
    using System.Threading.Tasks;
    using Amazon.IdentityManagement;
    using Amazon.IdentityManagement.Model;

    public class DeleteRolePolicy
    {
        /// <summary>
        /// Initializes the IAM client object and then calls DeleteRolePolicyAsync
        /// to delete the Policy attached to the Role.
        /// </summary>
        public static async Task Main()
        {
            var client = new AmazonIdentityManagementServiceClient();
            var response = await client.DeleteRolePolicyAsync(new
 DeleteRolePolicyRequest
            {
                PolicyName = "ExamplePolicy",
                RoleName = "Test-Role",
            });

            if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
            {
```

```
                Console.WriteLine("Policy successfully deleted.");
            }
            else
            {
                Console.WriteLine("Could not delete pollicy.");
            }
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see DeleteRolePolicy in *AWS SDK for .NET API Reference*.

Delete a user

The following code example shows how to delete an IAM user.

AWS SDK for .NET

```
        /// <summary>
        /// Delete the user, and other resources created for this example.
        /// </summary>
        /// <param name="client">The initialized client object.</param>
        /// <param name=accessKeyId">The Id of the user's access key.</param>"
        /// <param name="userName">The user name of the user to delete.</param>
        /// <param name="policyName">The name of the policy to delete.</param>
        /// <param name="policyArn">The Amazon Resource Name ARN of the Policy to
delete.</param>
        /// <param name="roleName">The name of the role that will be deleted.</param>
        public static async Task DeleteResourcesAsync(
            AmazonIdentityManagementServiceClient client,
            string accessKeyId,
            string userName,
            string policyArn,
            string roleName)
        {
            var detachPolicyResponse = await client.DetachRolePolicyAsync(new
DetachRolePolicyRequest
            {
                PolicyArn = policyArn,
                RoleName = roleName,
            });

            var delPolicyResponse = await client.DeletePolicyAsync(new
DeletePolicyRequest
            {
                PolicyArn = policyArn,
            });

            var delRoleResponse = await client.DeleteRoleAsync(new DeleteRoleRequest
            {
                RoleName = roleName,
            });

            var delAccessKey = await client.DeleteAccessKeyAsync(new
DeleteAccessKeyRequest
            {
                AccessKeyId = accessKeyId,
                UserName = userName,
            });
```

```
            var delUserResponse = await client.DeleteUserAsync(new DeleteUserRequest
            {
                UserName = userName,
            });

        }
```

- Find instructions and more code on GitHub.
- For API details, see DeleteUser in *AWS SDK for .NET API Reference*.

Delete an access key

The following code example shows how to delete an IAM access key.

AWS SDK for .NET

```
        /// <summary>
        /// Delete the user, and other resources created for this example.
        /// </summary>
        /// <param name="client">The initialized client object.</param>
        /// <param name=accessKeyId">The Id of the user's access key.</param>"
        /// <param name="userName">The user name of the user to delete.</param>
        /// <param name="policyName">The name of the policy to delete.</param>
        /// <param name="policyArn">The Amazon Resource Name ARN of the Policy to
delete.</param>
        /// <param name="roleName">The name of the role that will be deleted.</param>
        public static async Task DeleteResourcesAsync(
            AmazonIdentityManagementServiceClient client,
            string accessKeyId,
            string userName,
            string policyArn,
            string roleName)
        {
            var detachPolicyResponse = await client.DetachRolePolicyAsync(new
DetachRolePolicyRequest
            {
                PolicyArn = policyArn,
                RoleName = roleName,
            });

            var delPolicyResponse = await client.DeletePolicyAsync(new
DeletePolicyRequest
            {
                PolicyArn = policyArn,
            });

            var delRoleResponse = await client.DeleteRoleAsync(new DeleteRoleRequest
            {
                RoleName = roleName,
            });

            var delAccessKey = await client.DeleteAccessKeyAsync(new
DeleteAccessKeyRequest
            {
                AccessKeyId = accessKeyId,
                UserName = userName,
            });
```

```
            var delUserResponse = await client.DeleteUserAsync(new DeleteUserRequest
            {
                UserName = userName,
            });

        }
```

- Find instructions and more code on GitHub.
- For API details, see DeleteAccessKey in *AWS SDK for .NET API Reference*.

Detach a policy from a role

The following code example shows how to detach an IAM policy from a role.

AWS SDK for .NET

```
        /// <summary>
        /// Delete the user, and other resources created for this example.
        /// </summary>
        /// <param name="client">The initialized client object.</param>
        /// <param name=accessKeyId">The Id of the user's access key.</param>"
        /// <param name="userName">The user name of the user to delete.</param>
        /// <param name="policyName">The name of the policy to delete.</param>
        /// <param name="policyArn">The Amazon Resource Name ARN of the Policy to
delete.</param>
        /// <param name="roleName">The name of the role that will be deleted.</param>
        public static async Task DeleteResourcesAsync(
            AmazonIdentityManagementServiceClient client,
            string accessKeyId,
            string userName,
            string policyArn,
            string roleName)
        {
            var detachPolicyResponse = await client.DetachRolePolicyAsync(new
DetachRolePolicyRequest
            {
                PolicyArn = policyArn,
                RoleName = roleName,
            });

            var delPolicyResponse = await client.DeletePolicyAsync(new
DeletePolicyRequest
            {
                PolicyArn = policyArn,
            });

            var delRoleResponse = await client.DeleteRoleAsync(new DeleteRoleRequest
            {
                RoleName = roleName,
            });

            var delAccessKey = await client.DeleteAccessKeyAsync(new
DeleteAccessKeyRequest
            {
                AccessKeyId = accessKeyId,
                UserName = userName,
            });

            var delUserResponse = await client.DeleteUserAsync(new DeleteUserRequest
```

```
            {
                UserName = userName,
            });

        }
```

- Find instructions and more code on GitHub.
- For API details, see DetachRolePolicy in *AWS SDK for .NET API Reference*.

Get a policy

The following code example shows how to get an IAM policy.

AWS SDK for .NET

```
using System;
using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

var client = new AmazonIdentityManagementServiceClient();
var request = new GetPolicyRequest
{
    PolicyArn = "POLICY_ARN",
};

var response = await client.GetPolicyAsync(request);

Console.Write($"{response.Policy.PolicyName} was created on ");
Console.WriteLine($"{response.Policy.CreateDate}");
```

- Find instructions and more code on GitHub.
- For API details, see GetPolicy in *AWS SDK for .NET API Reference*.

Get a role

The following code example shows how to get an IAM role.

AWS SDK for .NET

```
using System;
using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

var client = new AmazonIdentityManagementServiceClient();

var response = await client.GetRoleAsync(new GetRoleRequest
{
    RoleName = "LambdaS3Role",
});

if (response.Role is not null)
{
```

```
    Console.WriteLine($"{response.Role.RoleName} with ARN: {response.Role.Arn}");
    Console.WriteLine($"{response.Role.Description}");
    Console.WriteLine($"Created: {response.Role.CreateDate} Last used on:
 { response.Role.RoleLastUsed}");
}
```

- Find instructions and more code on GitHub.
- For API details, see GetRole in *AWS SDK for .NET API Reference*.

Get the account password policy

The following code example shows how to get the IAM account password policy.

AWS SDK for .NET

```
using System;
using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

var client = new AmazonIdentityManagementServiceClient();

try
{
    var request = new GetAccountPasswordPolicyRequest();
    var response = await client.GetAccountPasswordPolicyAsync(request);

    Console.WriteLine($"{response.PasswordPolicy}");
}
catch (NoSuchEntityException ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}
```

- Find instructions and more code on GitHub.
- For API details, see GetAccountPasswordPolicy in *AWS SDK for .NET API Reference*.

List SAML providers

The following code example shows how to list SAML providers for IAM.

AWS SDK for .NET

```
using System;
using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

var client = new AmazonIdentityManagementServiceClient();

var response = await client.ListSAMLProvidersAsync(new ListSAMLProvidersRequest());

response.SAMLProviderList.ForEach(samlProvider =>
{
    Console.WriteLine($"{samlProvider.Arn} created on: {samlProvider.CreateDate}");
});
```

- Find instructions and more code on GitHub.
- For API details, see ListSAMLProviders in *AWS SDK for .NET API Reference*.

List groups

The following code example shows how to list IAM groups.

AWS SDK for .NET

```
using System;
using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

var client = new AmazonIdentityManagementServiceClient();

var request = new ListGroupsRequest
{
    MaxItems = 10,
};

var response = await client.ListGroupsAsync(request);

do
{
    response.Groups.ForEach(group =>
    {
        Console.WriteLine($"{group.GroupName} created on: {group.CreateDate}");
    });

    if (response.IsTruncated)
    {
        request.Marker = response.Marker;
        response = await client.ListGroupsAsync(request);
    }
} while (response.IsTruncated);
```

- Find instructions and more code on GitHub.
- For API details, see ListGroups in *AWS SDK for .NET API Reference*.

List inline policies for a role

The following code example shows how to list inline policies for an IAM role.

AWS SDK for .NET

```
using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;
using System;

var client = new AmazonIdentityManagementServiceClient();
var request = new ListRolePoliciesRequest
{
    RoleName = "LambdaS3Role",
};
```

```
var response = new ListRolePoliciesResponse();

do
{
    response = await client.ListRolePoliciesAsync(request);

    if (response.PolicyNames.Count > 0)
    {
        response.PolicyNames.ForEach(policyName =>
        {
            Console.WriteLine($"{policyName}");
        });
    }

    // As long as response.IsTruncated is true, set request.Marker equal
    // to response.Marker and call ListRolesAsync again.
    if (response.IsTruncated)
    {
        request.Marker = response.Marker;
    }
} while (response.IsTruncated);
```

- Find instructions and more code on GitHub.
- For API details, see ListRolePolicies in *AWS SDK for .NET API Reference*.

List policies

The following code example shows how to list IAM policies.

AWS SDK for .NET

```
using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;
using System;

var client = new AmazonIdentityManagementServiceClient();

var request = new ListPoliciesRequest
{
    MaxItems = 10,
};

var response = new ListPoliciesResponse();

do
{
    response = await client.ListPoliciesAsync(request);
    response.Policies.ForEach(policy =>
    {
        Console.Write($"{policy.PolicyName} ");
        Console.Write($"with ID: {policy.PolicyId} ");
        Console.Write($"and ARN: {policy.Arn}. ");
        Console.WriteLine($"It was created on {policy.CreateDate}.");
    });

    if (response.IsTruncated)
    {
        request.Marker = response.Marker;
```

```
    }
} while (response.IsTruncated);
```

- Find instructions and more code on GitHub.
- For API details, see ListPolicies in *AWS SDK for .NET API Reference*.

List policies attached to a role

The following code example shows how to list policies attached to an IAM role.

AWS SDK for .NET

```
using System;
using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

var client = new AmazonIdentityManagementServiceClient();
var request = new ListAttachedRolePoliciesRequest
{
    MaxItems = 10,
    RoleName = "testAssumeRole",
};

var response = await client.ListAttachedRolePoliciesAsync(request);

do
{
    response.AttachedPolicies.ForEach(policy =>
    {
        Console.WriteLine($"{policy.PolicyName} with ARN: {policy.PolicyArn}");
    });

    if (response.IsTruncated)
    {
        request.Marker = response.Marker;
        response = await client.ListAttachedRolePoliciesAsync(request);
    }

} while (response.IsTruncated);
```

- Find instructions and more code on GitHub.
- For API details, see ListAttachedRolePolicies in *AWS SDK for .NET API Reference*.

List roles

The following code example shows how to list IAM roles.

AWS SDK for .NET

```
using System;
using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

var client = new AmazonIdentityManagementServiceClient();
```

```
// Without the MaxItems value, the ListRolesAsync method will
// return information for up to 100 roles. If there are more
// than the MaxItems value or more than 100 roles, the response
// value IsTruncated will be true.
var request = new ListRolesRequest
{
    MaxItems = 10,
};

var response = new ListRolesResponse();

do
{
    response = await client.ListRolesAsync(request);
    response.Roles.ForEach(role =>
    {
        Console.WriteLine($"{role.RoleName} - ARN {role.Arn}");
    });

    // As long as response.IsTruncated is true, set request.Marker equal
    // to response.Marker and call ListRolesAsync again.
    if (response.IsTruncated)
    {
        request.Marker = response.Marker;
    }
} while (response.IsTruncated);
```

- Find instructions and more code on GitHub.
- For API details, see ListRoles in *AWS SDK for .NET API Reference*.

List users

The following code example shows how to list IAM users.

AWS SDK for .NET

```
using System;
using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

var client = new AmazonIdentityManagementServiceClient();
var request = new ListUsersRequest
{
    MaxItems = 10,
};
var response = await client.ListUsersAsync(request);

do
{
    response.Users.ForEach(user =>
    {
        Console.WriteLine($"{user.UserName} created on {user.CreateDate}.");
        Console.WriteLine($"ARN: {user.Arn}\n");
    });

    request.Marker = response.Marker;
    response = await client.ListUsersAsync(request);
} while (response.IsTruncated);
```

- Find instructions and more code on GitHub.
- For API details, see ListUsers in *AWS SDK for .NET API Reference*.

Scenarios

Create a user and assume a role

The following code example shows how to:

- Create a user who has no permissions.
- Create a role that grants permission to list Amazon S3 buckets for the account.
- Add a policy to let the user assume the role.
- Assume the role and list Amazon S3 buckets using temporary credentials.
- Delete the policy, role, and user.

AWS SDK for .NET

```
   using System;
   using System.IO;
   using System.Threading.Tasks;
   using Amazon;
   using Amazon.IdentityManagement;
   using Amazon.IdentityManagement.Model;
   using Amazon.S3;
   using Amazon.SecurityToken;
   using Amazon.SecurityToken.Model;

   public class IAM_Basics
   {
       // Values needed for user, role, and policies.
       private const string UserName = "example-user";
       private const string S3PolicyName = "s3-list-buckets-policy";
       private const string RoleName = "temporary-role";
       private const string AssumePolicyName = "sts-trust-user";

       private static readonly RegionEndpoint Region = RegionEndpoint.USEast2;

       public static async Task Main()
       {
           DisplayInstructions();

           // Create the IAM client object.
           var client = new AmazonIdentityManagementServiceClient(Region);

           // First create a user. By default, the new user has
           // no permissions.
           Console.WriteLine($"Creating a new user with user name: {UserName}.");
           var user = await CreateUserAsync(client, UserName);
           var userArn = user.Arn;
           Console.WriteLine($"Successfully created user: {UserName} with ARN:
{userArn}.");

           // Create an AccessKey for the user.
           var accessKey = await CreateAccessKeyAsync(client, UserName);

           var accessKeyId = accessKey.AccessKeyId;
```

```
            var secretAccessKey = accessKey.SecretAccessKey;

            // Try listing the Amazon Simple Storage Service (Amazon S3)
            // buckets. This should fail at this point because the user doesn't
            // have permissions to perform this task.
            var s3Client1 = new AmazonS3Client(accessKeyId, secretAccessKey);
            await ListMyBucketsAsync(s3Client1);

            // Define a role policy document that allows the new user
            // to assume the role.
            // string assumeRolePolicyDocument = File.ReadAllText("assumePolicy.json");
            string assumeRolePolicyDocument = "{" +
                "\"Version\": \"2012-10-17\"," +
                "\"Statement\": [{" +
                "\"Effect\": \"Allow\"," +
                "\"Principal\": {" +
                $" \"AWS\": \"{userArn}\"" +
                "}," +
                    "\"Action\": \"sts:AssumeRole\"" +
                "}]" +
            "}";

            // Permissions to list all buckets.
            string policyDocument = "{" +
                "\"Version\": \"2012-10-17\"," +
                " \"Statement\" : [{" +
                    " \"Action\" : [\"s3:ListAllMyBuckets\"]," +
                    " \"Effect\" : \"Allow\"," +
                    " \"Resource\" : \"*\"" +
                "}]" +
            "}";

            // Create the role to allow listing the S3 buckets. Role names are
            // not case sensitive and must be unique to the account for which it
            // is created.
            var role = await CreateRoleAsync(client, RoleName,
 assumeRolePolicyDocument);
            var roleArn = role.Arn;

            // Create a policy with permissions to list S3 buckets
            var policy = await CreatePolicyAsync(client, S3PolicyName, policyDocument);

            // Wait 15 seconds for the policy to be created.
            WaitABit(15, "Waiting for the policy to be available.");

            // Attach the policy to the role you created earlier.
            await AttachRoleAsync(client, policy.Arn, RoleName);

            // Wait 15 seconds for the role to be updated.
            Console.WriteLine();
            WaitABit(15, "Waiting to time for the policy to be attached.");

            // Use the AWS Security Token Service (AWS STS) to have the user
            // assume the role we created.
            var stsClient = new AmazonSecurityTokenServiceClient(accessKeyId,
 secretAccessKey);

            // Wait for the new credentials to become valid.
            WaitABit(10, "Waiting for the credentials to be valid.");

            var assumedRoleCredentials = await AssumeS3RoleAsync(stsClient, "temporary-
session", roleArn);

            // Try again to list the buckets using the client created with
            // the new user's credentials. This time, it should work.
            var s3Client2 = new AmazonS3Client(assumedRoleCredentials);
```

```
            await ListMyBucketsAsync(s3Client2);

            // Now clean up all the resources used in the example.
            await DeleteResourcesAsync(client, accessKeyId, UserName, policy.Arn,
 RoleName);

            Console.WriteLine("IAM Demo completed.");
        }


        /// <summary>
        /// Create a new IAM user.
        /// </summary>
        /// <param name="client">The initialized IAM client object.</param>
        /// <param name="userName">A string representing the user name of the
        /// new user.</param>
        /// <returns>The newly created user.</returns>
        public static async Task<User> CreateUserAsync(
            AmazonIdentityManagementServiceClient client,
            string userName)
        {
            var request = new CreateUserRequest
            {
                UserName = userName,
            };

            var response = await client.CreateUserAsync(request);

            return response.User;
        }



        /// <summary>
        /// Create a new AccessKey for the user.
        /// </summary>
        /// <param name="client">The initialized IAM client object.</param>
        /// <param name="userName">The name of the user for whom to create the key.</
param>
        /// <returns>A new IAM access key for the user.</returns>
        public static async Task<AccessKey> CreateAccessKeyAsync(
            AmazonIdentityManagementServiceClient client,
            string userName)
        {
            var request = new CreateAccessKeyRequest
            {
                UserName = userName,
            };

            var response = await client.CreateAccessKeyAsync(request);

            if (response.AccessKey is not null)
            {
                Console.WriteLine($"Successfully created Access Key for {userName}.");
            }

            return response.AccessKey;
        }



        /// <summary>
        /// Create a policy to allow a user to list the buckets in an account.
        /// </summary>
        /// <param name="client">The initialized IAM client object.</param>
```

```
        /// <param name="policyName">The name of the poicy to create.</param>
        /// <param name="policyDocument">The permissions policy document.</param>
        /// <returns>The newly created ManagedPolicy object.</returns>
        public static async Task<ManagedPolicy> CreatePolicyAsync(
            AmazonIdentityManagementServiceClient client,
            string policyName,
            string policyDocument)
        {
            var request = new CreatePolicyRequest
            {
                PolicyName = policyName,
                PolicyDocument = policyDocument,
            };

            var response = await client.CreatePolicyAsync(request);

            return response.Policy;
        }



        /// <summary>
        /// Attach the policy to the role so that the user can assume it.
        /// </summary>
        /// <param name="client">The initialized IAM client object.</param>
        /// <param name="policyArn">The ARN of the policy to attach.</param>
        /// <param name="roleName">The name of the role to attach the policy to.</
param>
        public static async Task AttachRoleAsync(
            AmazonIdentityManagementServiceClient client,
            string policyArn,
            string roleName)
        {
            var request = new AttachRolePolicyRequest
            {
                PolicyArn = policyArn,
                RoleName = roleName,
            };

            var response = await client.AttachRolePolicyAsync(request);

            if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
            {
                Console.WriteLine("Successfully attached the policy to the role.");
            }
            else
            {
                Console.WriteLine("Could not attach the policy.");
            }
        }



        /// <summary>
        /// Create a new IAM role which we can attach to a user.
        /// </summary>
        /// <param name="client">The initialized IAM client object.</param>
        /// <param name="roleName">The name of the IAM role to create.</param>
        /// <param name="rolePermissions">The permissions which the role will have.</
param>
        /// <returns>A Role object representing the newly created role.</returns>
        public static async Task<Role> CreateRoleAsync(
            AmazonIdentityManagementServiceClient client,
            string roleName,
            string rolePermissions)
        {
```

```
            var request = new CreateRoleRequest
            {
                RoleName = roleName,
                AssumeRolePolicyDocument = rolePermissions,
            };

            var response = await client.CreateRoleAsync(request);

            return response.Role;
        }



        /// <summary>
        /// List the Amazon S3 buckets owned by the user.
        /// </summary>
        /// <param name="accessKeyId">The access key Id for the user.</param>
        /// <param name="secretAccessKey">The Secret access key for the user.</param>
        public static async Task ListMyBucketsAsync(AmazonS3Client client)
        {
            Console.WriteLine("\nPress <Enter> to list the S3 buckets using the new
user.\n");
            Console.ReadLine();

            try
            {
                // Get the list of buckets accessible by the new user.
                var response = await client.ListBucketsAsync();

                // Loop through the list and print each bucket's name
                // and creation date.
                Console.WriteLine(new string('-', 80));
                Console.WriteLine("Listing S3 buckets:\n");
                response.Buckets
                    .ForEach(b => Console.WriteLine($"Bucket name: {b.BucketName},
created on: {b.CreationDate}"));
            }
            catch (AmazonS3Exception ex)
            {
                // Something else went wrong. Display the error message.
                Console.WriteLine($"Error: {ex.Message}");
            }

            Console.WriteLine("Press <Enter> to continue.");
            Console.ReadLine();
        }



        /// <summary>
        /// Have the user assume the role that allows the role to be used to
        /// list all S3 buckets.
        /// </summary>
        /// <param name="client">An initialized AWS STS client object.</param>
        /// <param name="roleSession">The name of the session where the role
        /// assumption will be active.</param>
        /// <param name="roleToAssume">The Amazon Resource Name (ARN) of the
        /// role to assume.</param>
        /// <returns>The AssumedRoleUser object needed to perform the list
        /// buckets procedure.</returns>
        public static async Task<Credentials> AssumeS3RoleAsync(
            AmazonSecurityTokenServiceClient client,
            string roleSession,
            string roleToAssume)
        {
            // Create the request to use with the AssumeRoleAsync call.
```

```
            var request = new AssumeRoleRequest()
            {
                RoleSessionName = roleSession,
                RoleArn = roleToAssume,
            };

            var response = await client.AssumeRoleAsync(request);

            return response.Credentials;
        }



        /// <summary>
        /// Delete the user, and other resources created for this example.
        /// </summary>
        /// <param name="client">The initialized client object.</param>
        /// <param name=accessKeyId">The Id of the user's access key.</param>"
        /// <param name="userName">The user name of the user to delete.</param>
        /// <param name="policyName">The name of the policy to delete.</param>
        /// <param name="policyArn">The Amazon Resource Name ARN of the Policy to
delete.</param>
        /// <param name="roleName">The name of the role that will be deleted.</param>
        public static async Task DeleteResourcesAsync(
            AmazonIdentityManagementServiceClient client,
            string accessKeyId,
            string userName,
            string policyArn,
            string roleName)
        {
            var detachPolicyResponse = await client.DetachRolePolicyAsync(new
DetachRolePolicyRequest
            {
                PolicyArn = policyArn,
                RoleName = roleName,
            });

            var delPolicyResponse = await client.DeletePolicyAsync(new
DeletePolicyRequest
            {
                PolicyArn = policyArn,
            });

            var delRoleResponse = await client.DeleteRoleAsync(new DeleteRoleRequest
            {
                RoleName = roleName,
            });

            var delAccessKey = await client.DeleteAccessKeyAsync(new
DeleteAccessKeyRequest
            {
                AccessKeyId = accessKeyId,
                UserName = userName,
            });

            var delUserResponse = await client.DeleteUserAsync(new DeleteUserRequest
            {
                UserName = userName,
            });

        }


        /// <summary>
        /// Display a countdown and wait for a number of seconds.
        /// </summary>
```

```
        /// <param name="numSeconds">The number of seconds to wait.</param>
        public static void WaitABit(int numSeconds, string msg)
        {
            Console.WriteLine(msg);

            // Wait for the requested number of seconds.
            for (int i = numSeconds; i > 0; i--)
            {
                System.Threading.Thread.Sleep(1000);
                Console.Write($"{i}...");
            }

            Console.WriteLine("\n\nPress <Enter> to continue.");
            Console.ReadLine();
        }

        /// <summary>
        /// Shows the a description of the features of the program.
        /// </summary>
        public static void DisplayInstructions()
        {
            var separator = new string('-', 80);

            Console.WriteLine(separator);
            Console.WriteLine("IAM Basics");
            Console.WriteLine("This application uses the basic features of the AWS
Identity and Access");
            Console.WriteLine("Management (IAM) creating, managing, and controlling
access to resources for");
            Console.WriteLine("users. The application was created using the AWS SDK
for .NET version 3.7 and");
            Console.WriteLine(".NET Core 5. The application performs the following
actions:");
            Console.WriteLine();
            Console.WriteLine("1. Creates a user with no permissions");
            Console.WriteLine("2. Creates a rolw and policy that grants
s3:ListAllMyBuckets permission");
            Console.WriteLine("3. Grants the user permission to assume the role");
            Console.WriteLine("4. Creates an Amazon Simple Storage Service (Amazon S3)
client and tries");
            Console.WriteLine("   to list buckets. (This should fail.)");
            Console.WriteLine("5. Gets temporary credentials by assuming the role.");
            Console.WriteLine("6. Creates an Amazon S3 client object with the temporary
credentials and");
            Console.WriteLine("   lists the buckets. (This time it should work.)");
            Console.WriteLine("7. Deletes all of the resources created.");
            Console.WriteLine(separator);
            Console.WriteLine("Press <Enter> to continue.");
            Console.ReadLine();
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see the following topics in *AWS SDK for .NET API Reference*.
  - AttachRolePolicy
  - CreateAccessKey
  - CreatePolicy
  - CreateRole
  - CreateUser
  - DeleteAccessKey

- DeletePolicy
- DeleteRole
- DeleteUser
- DeleteUserPolicy
- DetachRolePolicy
- PutUserPolicy

AWS KMS examples using AWS SDK for .NET

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

Actions are code excerpts that show you how to call individual AWS KMS functions.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple AWS KMS functions.

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

Topics

Actions

Create a grant for a key

The following code example shows how to create a grant for a KMS key.

AWS SDK for .NET

This examples grants a user permission to use a key for encryption and decryption.

```
        public static async Task Main()
        {
            var client = new AmazonKeyManagementServiceClient();

            // The identity that is given permission to perform the operations
            // specified in the grant.
            var grantee = "arn:aws:iam::111122223333:role/ExampleRole";

            // The identifier of the AWS KMS key to which the grant applies. You
            // can use the key ID or the Amazon Resource Name (ARN) of the KMS key.
            var keyId = "7c9eccc2-38cb-4c4f-9db3-766ee8dd3ad4";

            var request = new CreateGrantRequest
            {
                GranteePrincipal = grantee,
                KeyId = keyId,

                // A list of operations that the grant allows.
                Operations = new List<string>
                {
                    "Encrypt",
                    "Decrypt",
                },
            };

            var response = await client.CreateGrantAsync(request);
```

```
            string grantId = response.GrantId; // The unique identifier of the grant.
            string grantToken = response.GrantToken; // The grant token.

            Console.WriteLine($"Id: {grantId}, Token: {grantToken}");
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see CreateGrant in *AWS SDK for .NET API Reference*.

Create a key

The following code example shows how to create an AWS KMS key.

AWS SDK for .NET

```
    using System;
    using System.Threading.Tasks;
    using Amazon.KeyManagementService;
    using Amazon.KeyManagementService.Model;

    public class CreateKey
    {
         public static async Task Main()
        {
            // Note that if you need to create a Key in an AWS Region
            // other than the region defined for the default user, you need to
            // pass the region to the client constructor.
            var client = new AmazonKeyManagementServiceClient();

            // The call to CreateKeyAsync will create a symmetrical AWS KMS
            // key. For more information about symmetrical and asymmetrical
            // keys, see:
            //
            // https://docs.aws.amazon.com/kms/latest/developerguide/symm-asymm-
choose.html
            var response = await client.CreateKeyAsync(new CreateKeyRequest());

            // The KeyMetadata object contains information about the new AWS KMS key.
            KeyMetadata keyMetadata = response.KeyMetadata;

            if (keyMetadata is not null)
            {
                Console.WriteLine($"KMS Key: {keyMetadata.KeyId} was successfully
 created.");
            }
            else
            {
                Console.WriteLine("Could not create KMS Key.");
            }
        }

    }
```

- Find instructions and more code on GitHub.
- For API details, see CreateKey in *AWS SDK for .NET API Reference*.

Create an alias for a key

The following code example shows how to create an alias for a KMS key key.

AWS SDK for .NET

```
using System;
using System.Threading.Tasks;
using Amazon.KeyManagementService;
using Amazon.KeyManagementService.Model;

public class CreateAlias
{
    public static async Task Main()
    {
        var client = new AmazonKeyManagementServiceClient();

        // The alias name must start with alias/ and can be
        // up to 256 alphanumeric characters long.
        var aliasName = "alias/ExampleAlias";

        // The value supplied as the TargetKeyId can be either
        // the key ID or key Amazon Resource Name (ARN) of the
        // AWS KMS key.
        var keyId = "1234abcd-12ab-34cd-56ef-1234567890ab";

        var request = new CreateAliasRequest
        {
            AliasName = aliasName,
            TargetKeyId = keyId,
        };

        var response = await client.CreateAliasAsync(request);

        if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
        {
            Console.WriteLine($"Alias, {aliasName}, successfully created.");
        }
        else
        {
            Console.WriteLine($"Could not create alias.");
        }
    }
}
```

- Find instructions and more code on GitHub.
- For API details, see CreateAlias in *AWS SDK for .NET API Reference*.

Describe a key

The following code example shows how to describe a KMS key.

AWS SDK for .NET

```
using System;
using System.Threading.Tasks;
using Amazon.KeyManagementService;
```

```
    using Amazon.KeyManagementService.Model;

    public class DescribeKey
    {
        public static async Task Main()
        {
            var keyId = "7c9eccc2-38cb-4c4f-9db3-766ee8dd3ad4";
            var request = new DescribeKeyRequest
            {
                KeyId = keyId,
            };

            var client = new AmazonKeyManagementServiceClient();

            var response = await client.DescribeKeyAsync(request);
            var metadata = response.KeyMetadata;

            Console.WriteLine($"{metadata.KeyId} created on: {metadata.CreationDate}");
            Console.WriteLine($"State: {metadata.KeyState}");
            Console.WriteLine($"{metadata.Description}");
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see DescribeKey in *AWS SDK for .NET API Reference*.

Disable a key

The following code example shows how to disable a KMS key.

AWS SDK for .NET

```
    using System;
    using System.Threading.Tasks;
    using Amazon.KeyManagementService;
    using Amazon.KeyManagementService.Model;

    public class DisableKey
    {
        public static async Task Main()
        {
            var client = new AmazonKeyManagementServiceClient();

            // The identifier of the AWS KMS key to disable. You can use the
            // key Id or the Amazon Resource Name (ARN) of the AWS KMS key.
            var keyId = "1234abcd-12ab-34cd-56ef-1234567890ab";

            var request = new DisableKeyRequest
            {
                KeyId = keyId,
            };

            var response = await client.DisableKeyAsync(request);

            if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
            {
                // Retrieve information about the key to show that it has now
                // been disabled.
                var describeResponse = await client.DescribeKeyAsync(new
DescribeKeyRequest
```

```
                {
                    KeyId = keyId,
                });
                Console.WriteLine($"{describeResponse.KeyMetadata.KeyId} - state:
{describeResponse.KeyMetadata.KeyState}");
            }
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see DisableKey in *AWS SDK for .NET API Reference*.

Enable a key

The following code example shows how to enable a KMS key.

AWS SDK for .NET

```
    using System;
    using System.Threading.Tasks;
    using Amazon.KeyManagementService;
    using Amazon.KeyManagementService.Model;

    public class EnableKey
    {
        public static async Task Main()
        {
            var client = new AmazonKeyManagementServiceClient();

            // The identifier of the AWS KMS key to enable. You can use the
            // key Id or the Amazon Resource Name (ARN) of the AWS KMS key.
            var keyId = "1234abcd-12ab-34cd-56ef-1234567890ab";

            var request = new EnableKeyRequest
            {
                KeyId = keyId,
            };

            var response = await client.EnableKeyAsync(request);
            if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
            {
                // Retrieve information about the key to show that it has now
                // been enabled.
                var describeResponse = await client.DescribeKeyAsync(new
DescribeKeyRequest
                {
                    KeyId = keyId,
                });
                Console.WriteLine($"{describeResponse.KeyMetadata.KeyId} - state:
{describeResponse.KeyMetadata.KeyState}");
            }
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see EnableKey in *AWS SDK for .NET API Reference*.

List aliases for a key

The following code example shows how to list aliases for a KMS key.

AWS SDK for .NET

```
   using System;
   using System.Threading.Tasks;
   using Amazon.KeyManagementService;
   using Amazon.KeyManagementService.Model;

   public class ListAliases
   {
       public static async Task Main()
       {
           var client = new AmazonKeyManagementServiceClient();
           var request = new ListAliasesRequest();
           var response = new ListAliasesResponse();

           do
           {
               response = await client.ListAliasesAsync(request);

               response.Aliases.ForEach(alias =>
               {
                   Console.WriteLine($"Created: {alias.CreationDate} Last Update:
{alias.LastUpdatedDate} Name: {alias.AliasName}");
               });

               request.Marker = response.NextMarker;
           }
           while (response.Truncated);
       }
   }
```

- Find instructions and more code on GitHub.
- For API details, see ListAliases in *AWS SDK for .NET API Reference*.

List grants for a key

The following code example shows how to list grants for a KMS key.

AWS SDK for .NET

```
   using System;
   using System.Threading.Tasks;
   using Amazon.KeyManagementService;
   using Amazon.KeyManagementService.Model;

   public class ListGrants
   {
       public static async Task Main()
       {
           // The identifier of the AWS KMS key to disable. You can use the
           // key Id or the Amazon Resource Name (ARN) of the AWS KMS key.
           var keyId = "1234abcd-12ab-34cd-56ef-1234567890ab";
           var client = new AmazonKeyManagementServiceClient();
           var request = new ListGrantsRequest
           {
```

```
            KeyId = keyId,
        };

        var response = new ListGrantsResponse();

        do
        {
            response = await client.ListGrantsAsync(request);

            response.Grants.ForEach(grant =>
            {
                Console.WriteLine($"{grant.GrantId}");
            });

            request.Marker = response.NextMarker;
        }
        while (response.Truncated);
    }
}
```

- Find instructions and more code on GitHub.
- For API details, see ListGrants in *AWS SDK for .NET API Reference*.

List keys

The following code example shows how to list KMS keys.

AWS SDK for .NET

```
using System;
using System.Threading.Tasks;
using Amazon.KeyManagementService;
using Amazon.KeyManagementService.Model;

public class ListKeys
{
    public static async Task Main()
    {
        var client = new AmazonKeyManagementServiceClient();
        var request = new ListKeysRequest();
        var response = new ListKeysResponse();

        do
        {
            response = await client.ListKeysAsync(request);

            response.Keys.ForEach(key =>
            {
                Console.WriteLine($"ID: {key.KeyId}, {key.KeyArn}");
            });

            // Set the Marker property when response.Truncated is true
            // in order to get the next keys.
            request.Marker = response.NextMarker;
        }
        while (response.Truncated);
    }
}
```

- Find instructions and more code on GitHub.
- For API details, see ListKeys in *AWS SDK for .NET API Reference*.

Amazon Rekognition examples using AWS SDK for .NET

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

Actions are code excerpts that show you how to call individual Amazon Rekognition functions.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple Amazon Rekognition functions.

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

Topics

Actions

Compare faces in an image against a reference image

The following code example shows how to compare faces in an image against a reference image with Amazon Rekognition.

For more information, see Comparing faces in images.

AWS SDK for .NET

```
        public static async Task Main()
        {
            float similarityThreshold = 70F;
            string sourceImage = "source.jpg";
            string targetImage = "target.jpg";

            var rekognitionClient = new AmazonRekognitionClient();

            Amazon.Rekognition.Model.Image imageSource = new
Amazon.Rekognition.Model.Image();

            try
            {
                using FileStream fs = new FileStream(sourceImage, FileMode.Open,
FileAccess.Read);
                byte[] data = new byte[fs.Length];
                fs.Read(data, 0, (int)fs.Length);
                imageSource.Bytes = new MemoryStream(data);
            }
            catch (Exception)
            {
                Console.WriteLine($"Failed to load source image: {sourceImage}");
                return;
            }

            Amazon.Rekognition.Model.Image imageTarget = new
Amazon.Rekognition.Model.Image();

            try
            {
```

```
                using FileStream fs = new FileStream(targetImage, FileMode.Open,
FileAccess.Read);
                byte[] data = new byte[fs.Length];
                data = new byte[fs.Length];
                fs.Read(data, 0, (int)fs.Length);
                imageTarget.Bytes = new MemoryStream(data);
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Failed to load target image: {targetImage}");
                Console.WriteLine(ex.Message);
                return;
            }

            var compareFacesRequest = new CompareFacesRequest
            {
                SourceImage = imageSource,
                TargetImage = imageTarget,
                SimilarityThreshold = similarityThreshold,
            };

            // Call operation
            var compareFacesResponse = await
rekognitionClient.CompareFacesAsync(compareFacesRequest);

            // Display results
            compareFacesResponse.FaceMatches.ForEach(match =>
            {
                ComparedFace face = match.Face;
                BoundingBox position = face.BoundingBox;
                Console.WriteLine($"Face at {position.Left} {position.Top} matches with
{match.Similarity}% confidence.");
            });

            Console.WriteLine($"Found {compareFacesResponse.UnmatchedFaces.Count}
face(s) that did not match.");
        }
```

- Find instructions and more code on GitHub.
- For API details, see CompareFaces in *AWS SDK for .NET API Reference*.

Create a collection

The following code example shows how to create an Amazon Rekognition collection.

For more information, see Creating a collection.

AWS SDK for .NET

```
        public static async Task Main()
        {
            var rekognitionClient = new AmazonRekognitionClient();

            string collectionId = "MyCollection";
            Console.WriteLine("Creating collection: " + collectionId);

            var createCollectionRequest = new CreateCollectionRequest
            {
                CollectionId = collectionId,
            };
```

```
            CreateCollectionResponse createCollectionResponse = await
rekognitionClient.CreateCollectionAsync(createCollectionRequest);
            Console.WriteLine($"CollectionArn :
{createCollectionResponse.CollectionArn}");
            Console.WriteLine($"Status code : {createCollectionResponse.StatusCode}");
        }
```

- Find instructions and more code on GitHub.
- For API details, see CreateCollection in *AWS SDK for .NET API Reference*.

Delete a collection

The following code example shows how to delete an Amazon Rekognition collection.

For more information, see Deleting a collection.

AWS SDK for .NET

```
        public static async Task Main()
        {
            var rekognitionClient = new AmazonRekognitionClient();

            string collectionId = "MyCollection";
            Console.WriteLine("Deleting collection: " + collectionId);

            var deleteCollectionRequest = new DeleteCollectionRequest()
            {
                CollectionId = collectionId,
            };

            var deleteCollectionResponse = await
rekognitionClient.DeleteCollectionAsync(deleteCollectionRequest);
            Console.WriteLine($"{collectionId}:
{deleteCollectionResponse.StatusCode}");
        }
```

- Find instructions and more code on GitHub.
- For API details, see DeleteCollection in *AWS SDK for .NET API Reference*.

Delete faces from a collection

The following code example shows how to delete faces from an Amazon Rekognition collection.

For more information, see Deleting faces from a collection.

AWS SDK for .NET

```
        public static async Task Main()
        {
            string collectionId = "MyCollection";
            var faces = new List<string> { "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx" };

            var rekognitionClient = new AmazonRekognitionClient();
```

```
            var deleteFacesRequest = new DeleteFacesRequest()
            {
                CollectionId = collectionId,
                FaceIds = faces,
            };

            DeleteFacesResponse deleteFacesResponse = await
rekognitionClient.DeleteFacesAsync(deleteFacesRequest);
            deleteFacesResponse.DeletedFaces.ForEach(face =>
            {
                Console.WriteLine($"FaceID: {face}");
            });
        }
```

- Find instructions and more code on GitHub.
- For API details, see DeleteFaces in *AWS SDK for .NET API Reference*.

Describe a collection

The following code example shows how to describe an Amazon Rekognition collection.

For more information, see Describing a collection.

AWS SDK for .NET

```
        public static async Task Main()
        {
            var rekognitionClient = new AmazonRekognitionClient();

            string collectionId = "MyCollection";
            Console.WriteLine($"Describing collection: {collectionId}");

            var describeCollectionRequest = new DescribeCollectionRequest()
            {
                CollectionId = collectionId,
            };

            var describeCollectionResponse = await
rekognitionClient.DescribeCollectionAsync(describeCollectionRequest);
            Console.WriteLine($"Collection ARN:
{describeCollectionResponse.CollectionARN}");
            Console.WriteLine($"Face count: {describeCollectionResponse.FaceCount}");
            Console.WriteLine($"Face model version:
{describeCollectionResponse.FaceModelVersion}");
            Console.WriteLine($"Created:
{describeCollectionResponse.CreationTimestamp}");
        }
```

- Find instructions and more code on GitHub.
- For API details, see DescribeCollection in *AWS SDK for .NET API Reference*.

Detect faces in an image

The following code example shows how to detect faces in an image with Amazon Rekognition.

For more information, see Detecting faces in an image.

AWS SDK for .NET

```
        public static async Task Main()
        {
            string photo = "input.jpg";
            string bucket = "bucket";

            var rekognitionClient = new AmazonRekognitionClient();

            var detectFacesRequest = new DetectFacesRequest()
            {
                Image = new Image()
                {
                    S3Object = new S3Object()
                    {
                        Name = photo,
                        Bucket = bucket,
                    },
                },

                // Attributes can be "ALL" or "DEFAULT".
                // "DEFAULT": BoundingBox, Confidence, Landmarks, Pose, and Quality.
                // "ALL": See https://docs.aws.amazon.com/sdkfornet/v3/apidocs/items/
Rekognition/TFaceDetail.html
                Attributes = new List<string>() { "ALL" },
            };

            try
            {
                DetectFacesResponse detectFacesResponse = await
 rekognitionClient.DetectFacesAsync(detectFacesRequest);
                bool hasAll = detectFacesRequest.Attributes.Contains("ALL");
                foreach (FaceDetail face in detectFacesResponse.FaceDetails)
                {
                    Console.WriteLine($"BoundingBox: top={face.BoundingBox.Left}
 left={face.BoundingBox.Top} width={face.BoundingBox.Width}
 height={face.BoundingBox.Height}");
                    Console.WriteLine($"Confidence: {face.Confidence}");
                    Console.WriteLine($"Landmarks: {face.Landmarks.Count}");
                    Console.WriteLine($"Pose: pitch={face.Pose.Pitch}
 roll={face.Pose.Roll} yaw={face.Pose.Yaw}");
                    Console.WriteLine($"Brightness:
 {face.Quality.Brightness}\tSharpness: {face.Quality.Sharpness}");

                    if (hasAll)
                    {
                        Console.WriteLine($"Estimated age is between
 {face.AgeRange.Low} and {face.AgeRange.High} years old.");
                    }
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
```

Display bounding box information for all faces in an image.

```
        public static async Task Main()
        {
```

```
            string photo = @"D:\Development\AWS-Examples\Rekognition\target.jpg"; //
"photo.jpg";

            var rekognitionClient = new AmazonRekognitionClient();

            var image = new Amazon.Rekognition.Model.Image();
            try
            {
                using var fs = new FileStream(photo, FileMode.Open, FileAccess.Read);
                byte[] data = null;
                data = new byte[fs.Length];
                fs.Read(data, 0, (int)fs.Length);
                image.Bytes = new MemoryStream(data);
            }
            catch (Exception)
            {
                Console.WriteLine("Failed to load file " + photo);
                return;
            }

            int height;
            int width;

            // Used to extract original photo width/height
            using (var imageBitmap = new Bitmap(photo))
            {
                height = imageBitmap.Height;
                width = imageBitmap.Width;
            }

            Console.WriteLine("Image Information:");
            Console.WriteLine(photo);
            Console.WriteLine("Image Height: " + height);
            Console.WriteLine("Image Width: " + width);

            try
            {
                var detectFacesRequest = new DetectFacesRequest()
                {
                    Image = image,
                    Attributes = new List<string>() { "ALL" },
                };

                DetectFacesResponse detectFacesResponse = await
rekognitionClient.DetectFacesAsync(detectFacesRequest);
                detectFacesResponse.FaceDetails.ForEach(face =>
                {
                    Console.WriteLine("Face:");
                    ShowBoundingBoxPositions(
                        height,
                        width,
                        face.BoundingBox,
                        detectFacesResponse.OrientationCorrection);

                    Console.WriteLine($"BoundingBox: top={face.BoundingBox.Left}
left={face.BoundingBox.Top} width={face.BoundingBox.Width}
height={face.BoundingBox.Height}");
                    Console.WriteLine($"The detected face is estimated to be between
{face.AgeRange.Low} and {face.AgeRange.High} years old.\n");
                });
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
```

```
        /// <summary>
        /// Display the bounding box information for an image.
        /// </summary>
        /// <param name="imageHeight">The height of the image.</param>
        /// <param name="imageWidth">The width of the image.</param>
        /// <param name="box">The bounding box for a face found within the image.</
param>
        /// <param name="rotation">The rotation of the face's bounding box.</param>
        public static void ShowBoundingBoxPositions(int imageHeight, int imageWidth,
 BoundingBox box, string rotation)
        {
            float left;
            float top;

            if (rotation == null)
            {
                Console.WriteLine("No estimated orientation. Check Exif data.");
                return;
            }

            // Calculate face position based on image orientation.
            switch (rotation)
            {
                case "ROTATE_0":
                    left = imageWidth * box.Left;
                    top = imageHeight * box.Top;
                    break;
                case "ROTATE_90":
                    left = imageHeight * (1 - (box.Top + box.Height));
                    top = imageWidth * box.Left;
                    break;
                case "ROTATE_180":
                    left = imageWidth - (imageWidth * (box.Left + box.Width));
                    top = imageHeight * (1 - (box.Top + box.Height));
                    break;
                case "ROTATE_270":
                    left = imageHeight * box.Top;
                    top = imageWidth * (1 - box.Left - box.Width);
                    break;
                default:
                    Console.WriteLine("No estimated orientation information. Check Exif
 data.");
                    return;
            }

            // Display face location information.
            Console.WriteLine($"Left: {left}");
            Console.WriteLine($"Top: {top}");
            Console.WriteLine($"Face Width: {imageWidth * box.Width}");
            Console.WriteLine($"Face Height: {imageHeight * box.Height}");
        }
```

- Find instructions and more code on GitHub.
- For API details, see DetectFaces in *AWS SDK for .NET API Reference*.

Detect labels in an image

The following code example shows how to detect labels in an image with Amazon Rekognition.

For more information, see Detecting labels in an image.

AWS SDK for .NET

```
        public static async Task Main()
        {
            string photo = "del_river_02092020_01.jpg"; // "input.jpg";
            string bucket = "igsmiths3photos"; // "bucket";

            var rekognitionClient = new AmazonRekognitionClient();

            var detectlabelsRequest = new DetectLabelsRequest
            {
                Image = new Image()
                {
                    S3Object = new S3Object()
                    {
                        Name = photo,
                        Bucket = bucket,
                    },
                },
                MaxLabels = 10,
                MinConfidence = 75F,
            };

            try
            {
                DetectLabelsResponse detectLabelsResponse = await
rekognitionClient.DetectLabelsAsync(detectlabelsRequest);
                Console.WriteLine("Detected labels for " + photo);
                foreach (Label label in detectLabelsResponse.Labels)
                {
                    Console.WriteLine($"Name: {label.Name} Confidence:
{label.Confidence}");
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
```

Detect labels in an image file stored on your computer.

```
        public static async Task Main()
        {
            string photo = "input.jpg";

            var image = new Amazon.Rekognition.Model.Image();
            try
            {
                using var fs = new FileStream(photo, FileMode.Open, FileAccess.Read);
                byte[] data = null;
                data = new byte[fs.Length];
                fs.Read(data, 0, (int)fs.Length);
                image.Bytes = new MemoryStream(data);
            }
            catch (Exception)
            {
                Console.WriteLine("Failed to load file " + photo);
                return;
            }
```

```
            var rekognitionClient = new AmazonRekognitionClient();

            var detectlabelsRequest = new DetectLabelsRequest
            {
                Image = image,
                MaxLabels = 10,
                MinConfidence = 77F,
            };

            try
            {
                DetectLabelsResponse detectLabelsResponse = await
rekognitionClient.DetectLabelsAsync(detectlabelsRequest);
                Console.WriteLine($"Detected labels for {photo}");
                foreach (Label label in detectLabelsResponse.Labels)
                {
                    Console.WriteLine($"{label.Name}: {label.Confidence}");
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see DetectLabels in *AWS SDK for .NET API Reference*.

Detect moderation labels in an image

The following code example shows how to detect moderation labels in an image with Amazon Rekognition. Moderation labels identify content that may be inappropriate for some audiences.

For more information, see Detecting inappropriate images.

AWS SDK for .NET

```
        public static async Task Main(string[] args)
        {
            string photo = "input.jpg";
            string bucket = "bucket";

            var rekognitionClient = new AmazonRekognitionClient();

            var detectModerationLabelsRequest = new DetectModerationLabelsRequest()
            {
                Image = new Image()
                {
                    S3Object = new S3Object()
                    {
                        Name = photo,
                        Bucket = bucket,
                    },
                },
                MinConfidence = 60F,
            };

            try
            {
```

```
                var detectModerationLabelsResponse = await
rekognitionClient.DetectModerationLabelsAsync(detectModerationLabelsRequest);
                Console.WriteLine("Detected labels for " + photo);
                foreach (ModerationLabel label in
detectModerationLabelsResponse.ModerationLabels)
                {
                    Console.WriteLine($"Label: {label.Name}");
                    Console.WriteLine($"Confidence: {label.Confidence}");
                    Console.WriteLine($"Parent: {label.ParentName}");
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see DetectModerationLabels in *AWS SDK for .NET API Reference*.

Detect text in an image

The following code example shows how to detect text in an image with Amazon Rekognition.

For more information, see Detecting text in an image.

AWS SDK for .NET

```
        public static async Task Main()
        {
            string photo = "Dad_photographer.jpg"; // "input.jpg";
            string bucket = "igsmiths3photos"; // "bucket";

            var rekognitionClient = new AmazonRekognitionClient();

            var detectTextRequest = new DetectTextRequest()
            {
                Image = new Image()
                {
                    S3Object = new S3Object()
                    {
                        Name = photo,
                        Bucket = bucket,
                    },
                },
            };

            try
            {
                DetectTextResponse detectTextResponse = await
rekognitionClient.DetectTextAsync(detectTextRequest);
                Console.WriteLine($"Detected lines and words for {photo}");
                detectTextResponse.TextDetections.ForEach(text =>
                {
                    Console.WriteLine($"Detected: {text.DetectedText}");
                    Console.WriteLine($"Confidence: {text.Confidence}");
                    Console.WriteLine($"Id : {text.Id}");
                    Console.WriteLine($"Parent Id: {text.ParentId}");
                    Console.WriteLine($"Type: {text.Type}");
                });
```

```
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see DetectText in *AWS SDK for .NET API Reference*.

Get information about celebrities

The following code example shows how to get information about celebrities using Amazon Rekognition.

AWS SDK for .NET

```
        public static async Task Main()
        {
            string celebId = "nnnnnnnn";

            var rekognitionClient = new AmazonRekognitionClient();

            var celebrityInfoRequest = new GetCelebrityInfoRequest
            {
                Id = celebId,
            };

            Console.WriteLine($"Getting information for celebrity: {celebId}");

            var celebrityInfoResponse = await
rekognitionClient.GetCelebrityInfoAsync(celebrityInfoRequest);

            // Display celebrity information.
            Console.WriteLine($"celebrity name: {celebrityInfoResponse.Name}");
            Console.WriteLine("Further information (if available):");
            celebrityInfoResponse.Urls.ForEach(url =>
            {
                Console.WriteLine(url);
            });
        }
```

- Find instructions and more code on GitHub.
- For API details, see GetCelebrityInfo in *AWS SDK for .NET API Reference*.

Index faces to a collection

The following code example shows how to index faces in an image and add them to an Amazon Rekognition collection.

For more information, see Adding faces to a collection.

AWS SDK for .NET

```
        public static async Task Main()
        {
```

```
            string collectionId = "MyCollection2";
            string bucket = "doc-example-bucket";
            string photo = "input.jpg";

            var rekognitionClient = new AmazonRekognitionClient();

            var image = new Image
            {
                S3Object = new S3Object
                {
                    Bucket = bucket,
                    Name = photo,
                },
            };

            var indexFacesRequest = new IndexFacesRequest
            {
                Image = image,
                CollectionId = collectionId,
                ExternalImageId = photo,
                DetectionAttributes = new List<string>() { "ALL" },
            };

            IndexFacesResponse indexFacesResponse = await
rekognitionClient.IndexFacesAsync(indexFacesRequest);

            Console.WriteLine($"{photo} added");
            foreach (FaceRecord faceRecord in indexFacesResponse.FaceRecords)
            {
                Console.WriteLine($"Face detected: Faceid is
{faceRecord.Face.FaceId}");
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see IndexFaces in *AWS SDK for .NET API Reference*.

List collections

The following code example shows how to list Amazon Rekognition collections.

For more information, see Listing collections.

AWS SDK for .NET

```
        public static async Task Main()
        {
            var rekognitionClient = new AmazonRekognitionClient();

            Console.WriteLine("Listing collections");
            int limit = 10;

            var listCollectionsRequest = new ListCollectionsRequest
            {
                MaxResults = limit,
            };

            var listCollectionsResponse = new ListCollectionsResponse();

            do
```

```
            {
                if (listCollectionsResponse is not null)
                {
                    listCollectionsRequest.NextToken =
listCollectionsResponse.NextToken;
                }

                listCollectionsResponse = await
rekognitionClient.ListCollectionsAsync(listCollectionsRequest);

                listCollectionsResponse.CollectionIds.ForEach(id =>
                {
                    Console.WriteLine(id);
                });
            }
            while (listCollectionsResponse.NextToken is not null);
        }
```

- Find instructions and more code on GitHub.
- For API details, see ListCollections in *AWS SDK for .NET API Reference*.

List faces in a collection

The following code example shows how to list faces in an Amazon Rekognition collection.

For more information, see Listing faces in a collection.

AWS SDK for .NET

```
        public static async Task Main()
        {
            string collectionId = "MyCollection2";

            var rekognitionClient = new AmazonRekognitionClient();

            var listFacesResponse = new ListFacesResponse();
            Console.WriteLine($"Faces in collection {collectionId}");

            var listFacesRequest = new ListFacesRequest
            {
                CollectionId = collectionId,
                MaxResults = 1,
            };

            do
            {
                listFacesResponse = await
rekognitionClient.ListFacesAsync(listFacesRequest);
                listFacesResponse.Faces.ForEach(face =>
                {
                    Console.WriteLine(face.FaceId);
                });

                listFacesRequest.NextToken = listFacesResponse.NextToken;
            }
            while (!string.IsNullOrEmpty(listFacesResponse.NextToken));
        }
```

- Find instructions and more code on GitHub.
- For API details, see ListFaces in *AWS SDK for .NET API Reference*.

Recognize celebrities in an image

The following code example shows how to recognize celebrities in an image with Amazon Rekognition.

For more information, see Recognizing celebrities in an image.

AWS SDK for .NET

```
        public static async Task Main(string[] args)
        {
            string photo = "moviestars.jpg";

            var rekognitionClient = new AmazonRekognitionClient();

            var recognizeCelebritiesRequest = new RecognizeCelebritiesRequest();

            var img = new Amazon.Rekognition.Model.Image();
            byte[] data = null;
            try
            {
                using var fs = new FileStream(photo, FileMode.Open, FileAccess.Read);
                data = new byte[fs.Length];
                fs.Read(data, 0, (int)fs.Length);
            }
            catch (Exception)
            {
                Console.WriteLine($"Failed to load file {photo}");
                return;
            }

            img.Bytes = new MemoryStream(data);
            recognizeCelebritiesRequest.Image = img;

            Console.WriteLine($"Looking for celebrities in image {photo}\n");

            var recognizeCelebritiesResponse = await
rekognitionClient.RecognizeCelebritiesAsync(recognizeCelebritiesRequest);

            Console.WriteLine($"{recognizeCelebritiesResponse.CelebrityFaces.Count}
celebrity(s) were recognized.\n");
            recognizeCelebritiesResponse.CelebrityFaces.ForEach(celeb =>
            {
                Console.WriteLine($"Celebrity recognized: {celeb.Name}");
                Console.WriteLine($"Celebrity ID: {celeb.Id}");
                BoundingBox boundingBox = celeb.Face.BoundingBox;
                Console.WriteLine($"position: {boundingBox.Left} {boundingBox.Top}");
                Console.WriteLine("Further information (if available):");
                celeb.Urls.ForEach(url =>
                {
                    Console.WriteLine(url);
                });
            });

            Console.WriteLine($"{recognizeCelebritiesResponse.UnrecognizedFaces.Count}
face(s) were unrecognized.");
        }
```

- Find instructions and more code on GitHub.
- For API details, see RecognizeCelebrities in *AWS SDK for .NET API Reference*.

Search for faces in a collection

The following code example shows how to search for faces in an Amazon Rekognition collection that match another face from the collection.

For more information, see Searching for a face (face ID).

AWS SDK for .NET

```
        public static async Task Main()
        {
            string collectionId = "MyCollection";
            string faceId = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx";

            var rekognitionClient = new AmazonRekognitionClient();

            // Search collection for faces matching the face id.
            var searchFacesRequest = new SearchFacesRequest
            {
                CollectionId = collectionId,
                FaceId = faceId,
                FaceMatchThreshold = 70F,
                MaxFaces = 2,
            };

            SearchFacesResponse searchFacesResponse = await
rekognitionClient.SearchFacesAsync(searchFacesRequest);

            Console.WriteLine("Face matching faceId " + faceId);

            Console.WriteLine("Matche(s): ");
            searchFacesResponse.FaceMatches.ForEach(face =>
            {
                Console.WriteLine($"FaceId: {face.Face.FaceId} Similarity:
{face.Similarity}");
            });
        }
```

- Find instructions and more code on GitHub.
- For API details, see SearchFaces in *AWS SDK for .NET API Reference*.

Search for faces in a collection compared to a reference image

The following code example shows how to search for faces in an Amazon Rekognition collection compared to a reference image.

For more information, see Searching for a face (image).

AWS SDK for .NET

```
        public static async Task Main()
        {
            string collectionId = "MyCollection";
            string bucket = "bucket";
```

```
            string photo = "input.jpg";

            var rekognitionClient = new AmazonRekognitionClient();

            // Get an image object from S3 bucket.
            var image = new Image()
            {
                S3Object = new S3Object()
                {
                    Bucket = bucket,
                    Name = photo,
                },
            };

            var searchFacesByImageRequest = new SearchFacesByImageRequest()
            {
                CollectionId = collectionId,
                Image = image,
                FaceMatchThreshold = 70F,
                MaxFaces = 2,
            };

            SearchFacesByImageResponse searchFacesByImageResponse = await
rekognitionClient.SearchFacesByImageAsync(searchFacesByImageRequest);

            Console.WriteLine("Faces matching largest face in image from " + photo);
            searchFacesByImageResponse.FaceMatches.ForEach(face =>
            {
                Console.WriteLine($"FaceId: {face.Face.FaceId}, Similarity:
{face.Similarity}");
            });
        }
```

- Find instructions and more code on GitHub.
- For API details, see SearchFacesByImage in *AWS SDK for .NET API Reference*.

Amazon S3 examples using AWS SDK for .NET

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for .NET with Amazon S3.

Actions are code excerpts that show you how to call individual Amazon S3 functions.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple Amazon S3 functions.

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

Topics

Actions

Copy an object from one bucket to another

The following code example shows how to copy an Amazon S3 object from one bucket to another.

AWS SDK for .NET

```
        /// <summary>
        /// Copies an object in an Amazon S3 bucket to a folder within the
        /// same bucket.
        /// </summary>
        /// <param name="client">An initialized Amazon S3 client object.</param>
        /// <param name="bucketName">The name of the Amazon S3 bucket where the
        /// object to copy is located.</param>
        /// <param name="objectName">The object to be copied.</param>
        /// <param name="folderName">The folder to which the object will
        /// be copied.</param>
        /// <returns>A boolean value that indicates the success or failure of
        /// the copy operation.</returns>
        public static async Task<bool> CopyObjectInBucketAsync(
            IAmazonS3 client,
            string bucketName,
            string objectName,
            string folderName)
        {
            try
            {
                var request = new CopyObjectRequest
                {
                    SourceBucket = bucketName,
                    SourceKey = objectName,
                    DestinationBucket = bucketName,
                    DestinationKey = $"{folderName}\\{objectName}",
                };
                var response = await client.CopyObjectAsync(request);
                return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
            }
            catch (AmazonS3Exception ex)
            {
                Console.WriteLine($"Error copying object: '{ex.Message}'");
                return false;
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see CopyObject in *AWS SDK for .NET API Reference*.

Create a bucket

The following code example shows how to create an Amazon S3 bucket.

AWS SDK for .NET

```
        /// <summary>
        /// Shows how to create a new Amazon S3 bucket.
        /// </summary>
        /// <param name="client">An initialized Amazon S3 client object.</param>
        /// <param name="bucketName">The name of the bucket to create.</param>
        /// <returns>A boolean value representing the success or failure of
        /// the bucket creation process.</returns>
        public static async Task<bool> CreateBucketAsync(IAmazonS3 client, string
bucketName)
```

```
        {
            try
            {
                var request = new PutBucketRequest
                {
                    BucketName = bucketName,
                    UseClientRegion = true,
                };

                var response = await client.PutBucketAsync(request);
                return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
            }
            catch (AmazonS3Exception ex)
            {
                Console.WriteLine($"Error creating bucket: '{ex.Message}'");
                return false;
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see CreateBucket in *AWS SDK for .NET API Reference*.

Delete an empty bucket

The following code example shows how to delete an empty Amazon S3 bucket.

AWS SDK for .NET

```
        /// <summary>
        /// Shows how to delete an Amazon S3 bucket.
        /// </summary>
        /// <param name="client">An initialized Amazon S3 client object.</param>
        /// <param name="bucketName">The name of the Amazon S3 bucket to delete.</
param>
        /// <returns>A boolean value that represents the success or failure of
        /// the delete operation.</returns>
        public static async Task<bool> DeleteBucketAsync(IAmazonS3 client, string
 bucketName)
        {
            var request = new DeleteBucketRequest
            {
                BucketName = bucketName,
            };

            var response = await client.DeleteBucketAsync(request);
            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
```

- Find instructions and more code on GitHub.
- For API details, see DeleteBucket in *AWS SDK for .NET API Reference*.

Delete multiple objects

The following code example shows how to delete multiple objects from an Amazon S3 bucket.

AWS SDK for .NET

Delete all objects in an Amazon S3 bucket.

```
        /// <summary>
        /// Delete all of the objects stored in an existing Amazon S3 bucket.
        /// </summary>
        /// <param name="client">An initialized Amazon S3 client object.</param>
        /// <param name="bucketName">The name of the bucket from which the
        /// contents will be deleted.</param>
        /// <returns>A boolean value that represents the success or failure of
        /// deleting all of the objects in the bucket.</returns>
        public static async Task<bool> DeleteBucketContentsAsync(IAmazonS3 client,
string bucketName)
        {
            // Iterate over the contents of the bucket and delete all objects.
            var request = new ListObjectsV2Request
            {
                BucketName = bucketName,
            };

            try
            {
                var response = await client.ListObjectsV2Async(request);

                do
                {
                    response.S3Objects
                        .ForEach(async obj => await
client.DeleteObjectAsync(bucketName, obj.Key));

                    // If the response is truncated, set the request ContinuationToken
                    // from the NextContinuationToken property of the response.
                    request.ContinuationToken = response.NextContinuationToken;
                }
                while (response.IsTruncated);

                return true;
            }
            catch (AmazonS3Exception ex)
            {
                Console.WriteLine($"Error deleting objects: {ex.Message}");
                return false;
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see DeleteObjects in *AWS SDK for .NET API Reference*.

Get an object from a bucket

The following code example shows how to read data from an object in an Amazon S3 bucket.

AWS SDK for .NET

```
        /// <summary>
        /// Shows how to download an object from an Amazon S3 bucket to the
```

```
        /// local computer.
        /// </summary>
        /// <param name="client">An initialized Amazon S3 client object.</param>
        /// <param name="bucketName">The name of the bucket where the object is
        /// currently stored.</param>
        /// <param name="objectName">The name of the object to download.</param>
        /// <param name="filePath">The path, including filename, where the
        /// downloaded object will be stored.</param>
        /// <returns>A boolean value indicating the success or failure of the
        /// download process.</returns>
        public static async Task<bool> DownloadObjectFromBucketAsync(
            IAmazonS3 client,
            string bucketName,
            string objectName,
            string filePath)
        {
            // Create a GetObject request
            var request = new GetObjectRequest
            {
                BucketName = bucketName,
                Key = objectName,
            };

            // Issue request and remember to dispose of the response
            using GetObjectResponse response = await client.GetObjectAsync(request);

            try
            {
                // Save object to local file
                await response.WriteResponseStreamToFileAsync($"{filePath}\
\{objectName}", true, System.Threading.CancellationToken.None);
                return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
            }
            catch (AmazonS3Exception ex)
            {
                Console.WriteLine($"Error saving {objectName}: {ex.Message}");
                return false;
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see GetObject in *AWS SDK for .NET API Reference*.

List objects in a bucket

The following code example shows how to list objects in an Amazon S3 bucket.

AWS SDK for .NET

```
        /// <summary>
        /// Shows how to list the objects in an Amazon S3 bucket.
        /// </summary>
        /// <param name="client">An initialized Amazon S3 client object.</param>
        /// <param name="bucketName">The name of the bucket for which to list
        /// the contents.</param>
        /// <returns>A boolean value indicating the success or failure of the
        /// copy operation.</returns>
        public static async Task<bool> ListBucketContentsAsync(IAmazonS3 client, string
 bucketName)
```

```
        {
            try
            {
                var request = new ListObjectsV2Request
                {
                    BucketName = bucketName,
                    MaxKeys = 5,
                };

                Console.WriteLine("--------------------------------------");
                Console.WriteLine($"Listing the contents of {bucketName}:");
                Console.WriteLine("--------------------------------------");

                var response = new ListObjectsV2Response();

                do
                {
                    response = await client.ListObjectsV2Async(request);

                    response.S3Objects
                        .ForEach(obj => Console.WriteLine($"{obj.Key,-35}
{obj.LastModified.ToShortDateString(),10}{obj.Size,10}"));

                    // If the response is truncated, set the request ContinuationToken
                    // from the NextContinuationToken property of the response.
                    request.ContinuationToken = response.NextContinuationToken;
                }
                while (response.IsTruncated);

                return true;
            }
            catch (AmazonS3Exception ex)
            {
                Console.WriteLine($"Error encountered on server. Message:'{ex.Message}'
 getting list of objects.");
                return false;
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see ListObjects in *AWS SDK for .NET API Reference*.

Upload an object to a bucket

The following code example shows how to upload an object to an Amazon S3 bucket.

AWS SDK for .NET

```
        /// <summary>
        /// Shows how to upload a file from the local computer to an Amazon S3
        /// bucket.
        /// </summary>
        /// <param name="client">An initialized Amazon S3 client object.</param>
        /// <param name="bucketName">The Amazon S3 bucket to which the object
        /// will be uploaded.</param>
        /// <param name="objectName">The object to upload.</param>
        /// <param name="filePath">The path, including file name, of the object
        /// on the local computer to upload.</param>
        /// <returns>A boolean value indicating the success or failure of the
```

```
        /// upload procedure.</returns>
        public static async Task<bool> UploadFileAsync(
            IAmazonS3 client,
            string bucketName,
            string objectName,
            string filePath)
        {
            var request = new PutObjectRequest
            {
                BucketName = bucketName,
                Key = objectName,
                FilePath = filePath,
            };

            var response = await client.PutObjectAsync(request);
            if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
            {
                Console.WriteLine($"Successfully uploaded {objectName} to
{bucketName}.");
                return true;
            }
            else
            {
                Console.WriteLine($"Could not upload {objectName} to {bucketName}.");
                return false;
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see PutObject in *AWS SDK for .NET API Reference*.

Scenarios

Getting started with buckets and objects

The following code example shows how to:

- Create a bucket.
- Upload a file to the bucket.
- Download an object from a bucket.
- Copy an object to a subfolder in a bucket.
- List the objects in a bucket.
- Delete the objects in a bucket.
- Delete a bucket.

AWS SDK for .NET

```
    using System;
    using System.IO;
    using System.Threading.Tasks;
    using Amazon.S3;

    public class S3_Basics
    {
        public static async Task Main()
```

```
        {
            // Create an Amazon S3 client object. The constructor uses the
            // default user installed on the system. To work with Amazon S3
            // features in a different AWS Region, pass the AWS Region as a
            // parameter to the client constructor.
            IAmazonS3 client = new AmazonS3Client();
            string bucketName = string.Empty;
            string filePath = string.Empty;
            string keyName = string.Empty;

            Console.WriteLine("Amazon Simple Storage Service (Amazon S3) basic");
            Console.WriteLine("procedures. This application will:");
            Console.WriteLine("\n\t1. Create a bucket");
            Console.WriteLine("\n\t2. Upload an object to the new bucket");
            Console.WriteLine("\n\t3. Copy the uploaded object to a folder in the
bucket");
            Console.WriteLine("\n\t4. List the items in the new bucket");
            Console.WriteLine("\n\t5. Delete all the items in the bucket");
            Console.WriteLine("\n\t6. Delete the bucket");

Console.WriteLine("------------------------------------------------------------");

            // Create a bucket.
            Console.WriteLine("\nCreate a new Amazon S3 bucket.\n");

            Console.Write("Please enter a name for the new bucket: ");
            bucketName = Console.ReadLine();

            var success = await S3Bucket.CreateBucketAsync(client, bucketName);
            if (success)
            {
                Console.WriteLine($"Successfully created bucket: {bucketName}.\n");
            }
            else
            {
                Console.WriteLine($"Could not create bucket: {bucketName}.\n");
            }

            Console.WriteLine("Upload a file to the new bucket.");

            // Get the local path and filename for the file to upload.
            while (string.IsNullOrEmpty(filePath))
            {
                Console.Write("Please enter the path and filename of the file to
upload: ");
                filePath = Console.ReadLine();

                // Confirm that the file exists on the local computer.
                if (!File.Exists(filePath))
                {
                    Console.WriteLine($"Couldn't find {filePath}. Try again.\n");
                    filePath = string.Empty;
                }
            }

            // Get the file name from the full path.
            keyName = Path.GetFileName(filePath);

            success = await S3Bucket.UploadFileAsync(client, bucketName, keyName,
filePath);

            if (success)
            {
                Console.WriteLine($"Successfully uploaded {keyName} from {filePath} to
{bucketName}.\n");
            }
```

```
            else
            {
                Console.WriteLine($"Could not upload {keyName}.\n");
            }

            // Set the file path to an empty string to avoid overwriting the
            // file we just uploaded to the bucket.
            filePath = string.Empty;

            // Now get a new location where we can save the file.
            while (string.IsNullOrEmpty(filePath))
            {
                // First get the path to which the file will be downloaded.
                Console.Write("Please enter the path where the file will be downloaded:
");
                filePath = Console.ReadLine();

                // Confirm that the file exists on the local computer.
                if (File.Exists($"{filePath}\\{keyName}"))
                {
                    Console.WriteLine($"Sorry, the file already exists in that
location.\n");
                    filePath = string.Empty;
                }
            }

            // Download an object from a bucket.
            success = await S3Bucket.DownloadObjectFromBucketAsync(client, bucketName,
keyName, filePath);

            if (success)
            {
                Console.WriteLine($"Successfully downloaded {keyName}.\n");
            }
            else
            {
                Console.WriteLine($"Sorry, could not download {keyName}.\n");
            }

            // Copy the object to a different folder in the bucket.
            string folderName = string.Empty;

            while (string.IsNullOrEmpty(folderName))
            {
                Console.Write("Please enter the name of the folder to copy your object
to: ");
                folderName = Console.ReadLine();
            }

            while (string.IsNullOrEmpty(keyName))
            {
                // Get the name to give to the object once uploaded.
                Console.Write("Enter the name of the object to copy: ");
                keyName = Console.ReadLine();
            }

            await S3Bucket.CopyObjectInBucketAsync(client, bucketName, keyName,
folderName);

            // List the objects in the bucket.
            await S3Bucket.ListBucketContentsAsync(client, bucketName);

            // Delete the contents of the bucket.
            await S3Bucket.DeleteBucketContentsAsync(client, bucketName);

            // Deleting the bucket too quickly after deleting its contents will
```

```
            // cause an error that the bucket isn't empty. So...
            Console.WriteLine("Press <Enter> when you are ready to delete the
bucket.");
            _ = Console.ReadLine();

            // Delete the bucket.
            await S3Bucket.DeleteBucketAsync(client, bucketName);
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see the following topics in *AWS SDK for .NET API Reference*.
  - CopyObject
  - CreateBucket
  - DeleteBucket
  - DeleteObjects
  - GetObject
  - ListObjects
  - PutObject

Amazon SNS examples using AWS SDK for .NET

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

Actions are code excerpts that show you how to call individual Amazon SNS functions.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple Amazon SNS functions.

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

Topics

Actions

Check whether a phone number is opted out

The following code example shows how to check whether a phone number is opted out of receiving Amazon SNS messages.

AWS SDK for .NET

```
        /// <summary>
        /// Checks to see if the supplied phone number has been opted out.
        /// </summary>
        /// <param name="client">The initialized Amazon SNS Client object used
        /// to check if the phone number has been opted out.</param>
        /// <param name="phoneNumber">A string representing the phone number
        /// to check.</param>
```

```
        public static async Task CheckIfOptedOutAsync(IAmazonSimpleNotificationService
client, string phoneNumber)
        {
            var request = new CheckIfPhoneNumberIsOptedOutRequest
            {
                PhoneNumber = phoneNumber,
            };

            try
            {
                var response = await client.CheckIfPhoneNumberIsOptedOutAsync(request);

                if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
                {
                    string optOutStatus = response.IsOptedOut ? "opted out" : "not
opted out.";
                    Console.WriteLine($"The phone number: {phoneNumber} is
{optOutStatus}");
                }
            }
            catch (AuthorizationErrorException ex)
            {
                Console.WriteLine($"{ex.Message}");
            }
        }
```

- Find instructions and more code on GitHub.
- For API details, see CheckIfPhoneNumberIsOptedOut in *AWS SDK for .NET API Reference*.

Create a topic

The following code example shows how to create an Amazon SNS topic.

AWS SDK for .NET

```
        /// <summary>
        /// Creates a new SNS topic using the supplied topic name.
        /// </summary>
        /// <param name="client">The initialized SNS client object used to
        /// create the new topic.</param>
        /// <param name="topicName">A string representing the topic name.</param>
        /// <returns>The Amazon Resource Name (ARN) of the created topic.</returns>
        public static async Task<string>
CreateSNSTopicAsync(IAmazonSimpleNotificationService client, string topicName)
        {
            var request = new CreateTopicRequest
            {
                Name = topicName,
            };

            var response = await client.CreateTopicAsync(request);

            return response.TopicArn;
        }
```

- Find instructions and more code on GitHub.

- For API details, see CreateTopic in *AWS SDK for .NET API Reference*.

Delete a topic

The following code example shows how to delete an Amazon SNS topic and all subscriptions to that topic.

AWS SDK for .NET

```
   /// <summary>
   /// This example deletes an existing Amazon Simple Notification Service
   /// (Amazon SNS) topic. The example was created using the AWS SDK for .NET
   /// version 3.7 and .NET Core 5.0.
   /// </summary>
   public class DeleteSNSTopic
   {
       public static async Task Main()
       {
           string topicArn = "arn:aws:sns:us-east-2:704825161248:ExampleSNSTopic";
           IAmazonSimpleNotificationService client = new
AmazonSimpleNotificationServiceClient();

           var response = await client.DeleteTopicAsync(topicArn);
       }
   }
```

- Find instructions and more code on GitHub.
- For API details, see DeleteTopic in *AWS SDK for .NET API Reference*.

Get the properties of a topic

The following code example shows how to get the properties of an Amazon SNS topic.

AWS SDK for .NET

```
   /// <summary>
   /// This example shows how to retrieve the attributes of an Amazon Simple
   /// Notification Service (Amazon SNS) topic. The example was written using
   /// the AWS SDK for .NET 3.7 and .NET Core 5.0.
   /// </summary>
   public class GetTopicAttributes
   {
       public static async Task Main()
       {
           string topicArn = "arn:aws:sns:us-west-2:000000000000:ExampleSNSTopic";
           IAmazonSimpleNotificationService client = new
AmazonSimpleNotificationServiceClient();

           var attributes = await GetTopicAttributesAsync(client, topicArn);
           DisplayTopicAttributes(attributes);
       }

       /// <summary>
       /// Given the ARN of the Amazon SNS topic, this method retrieves the topic
       /// attributes.
```

```
        /// </summary>
        /// <param name="client">The initialized Amazon SNS client object used
        /// to retrieve the attributes for the Amazon SNS topic.</param>
        /// <param name="topicArn">The ARN of the topic for which to retrieve
        /// the attributes.</param>
        /// <returns>A Dictionary of topic attributes.</returns>
        public static async Task<Dictionary<string, string>> GetTopicAttributesAsync(
            IAmazonSimpleNotificationService client,
            string topicArn)
        {
            var response = await client.GetTopicAttributesAsync(topicArn);

            return response.Attributes;
        }

        /// <summary>
        /// This method displays the attributes for an Amazon SNS topic.
        /// </summary>
        /// <param name="topicAttributes">A Dictionary containing the
        /// attributes for an Amazon SNS topic.</param>
        public static void DisplayTopicAttributes(Dictionary<string, string>
topicAttributes)
        {
            foreach (KeyValuePair<string, string> entry in topicAttributes)
            {
                Console.WriteLine($"{entry.Key}: {entry.Value}\n");
            }
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see GetTopicAttributes in *AWS SDK for .NET API Reference*.

List the subscribers of a topic

The following code example shows how to retrieve the list of subscribers of an Amazon SNS topic.

AWS SDK for .NET

```
    /// <summary>
    /// This example will retrieve a list of the existing Amazon Simple
    /// Notification Service (Amazon SNS) subscriptions. The example was
    /// created using the AWS SDK for .NET 3.7 and .NET Core 5.0.
    /// </summary>
    public class ListSubscriptions
    {
        public static async Task Main()
        {
            IAmazonSimpleNotificationService client = new
AmazonSimpleNotificationServiceClient();

            var subscriptions = await GetSubscriptionsListAsync(client);

            DisplaySubscriptionList(subscriptions);
        }

        /// <summary>
        /// Gets a list of the existing Amazon SNS subscriptions.
        /// </summary>
```

```
        /// <param name="client">The initialized Amazon SNS client object used
        /// to obtain the list of subscriptions.</param>
        /// <returns>A List containing information about each subscription.</returns>
        public static async Task<List<Subscription>>
GetSubscriptionsListAsync(IAmazonSimpleNotificationService client)
        {
            var response = await client.ListSubscriptionsAsync();

            return response.Subscriptions;
        }

        /// <summary>
        /// Display a list of Amazon SNS subscription information.
        /// </summary>
        /// <param name="subscriptionList">A list containing details for existing
        /// Amazon SNS subscriptions.</param>
        public static void DisplaySubscriptionList(List<Subscription> subscriptionList)
        {
            foreach (var subscription in subscriptionList)
            {
                Console.WriteLine($"Owner: {subscription.Owner}");
                Console.WriteLine($"Subscription ARN: {subscription.SubscriptionArn}");
                Console.WriteLine($"Topic ARN: {subscription.TopicArn}");
                Console.WriteLine($"Endpoint: {subscription.Endpoint}");
                Console.WriteLine($"Protocol: {subscription.Protocol}");
                Console.WriteLine();
            }
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see ListSubscriptions in *AWS SDK for .NET API Reference*.

List topics

The following code example shows how to list Amazon SNS topics.

AWS SDK for .NET

```
        public static async Task Main()
        {
            IAmazonSimpleNotificationService client = new
AmazonSimpleNotificationServiceClient();

            await GetTopicListAsync(client);
        }

        /// <summary>
        /// Retrieves the list of Amazon SNS topics in groups of up to 100
        /// topics.
        /// </summary>
        /// <param name="client">The initialized Amazon SNS client object used
        /// to retrieve the list of topics.</param>
        public static async Task GetTopicListAsync(IAmazonSimpleNotificationService
client)
        {
            // If there are more than 100 Amazon SNS topics, the call to
            // ListTopicsAsync will return a value to pass to the
            // method to retrieve the next 100 (or less) topics.
```

```
        string nextToken = string.Empty;

        do
        {
            var response = await client.ListTopicsAsync(nextToken);
            DisplayTopicsList(response.Topics);
            nextToken = response.NextToken;
        }
        while (!string.IsNullOrEmpty(nextToken));
    }

    /// <summary>
    /// Displays the list of Amazon SNS Topic ARNs.
    /// </summary>
    /// <param name="topicList">The list of Topic ARNs.</param>
    public static void DisplayTopicsList(List<Topic> topicList)
    {
        foreach (var topic in topicList)
        {
            Console.WriteLine($"{topic.TopicArn}");
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see ListTopics in *AWS SDK for .NET API Reference*.

Publish to a topic

The following code example shows how to publish messages to an Amazon SNS topic.

AWS SDK for .NET

- Find instructions and more code on GitHub.
- For API details, see Publish in *AWS SDK for .NET API Reference*.

Amazon SQS examples using AWS SDK for .NET

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

Actions are code excerpts that show you how to call individual Amazon SQS functions.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple Amazon SQS functions.

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

Topics

Actions

Authorize a bucket to send messages to a queue

The following code example shows how to authorize an Amazon S3 bucket to send messages to an Amazon SQS queue.

AWS SDK for .NET

```
    using System;
    using System.Threading.Tasks;
    using Amazon.SQS;

    public class AuthorizeS3ToSendMessage
    {
        /// <summary>
        /// Initializes the Amazon SQS client object and then calls the
        /// AuthorizeS3ToSendMessageAsync method to authorize the named
        /// bucket to send messages in response to S3 events.
        /// </summary>
        public static async Task Main()
        {
            string queueUrl = "https://sqs.us-east-2.amazonaws.com/0123456789ab/
Example_Queue";
            string bucketName = "doc-example-bucket";

            // Create an Amazon SQS client object using the
            // default user. If the AWS Region you want to use
            // is different, supply the AWS Region as a parameter.
            IAmazonSQS client = new AmazonSQSClient();

            var queueARN = await client.AuthorizeS3ToSendMessageAsync(queueUrl,
 bucketName);

            if (!string.IsNullOrEmpty(queueARN))
            {
                Console.WriteLine($"The Amazon S3 bucket: {bucketName} has been
 successfully authorized.");
                Console.WriteLine($"{bucketName} can now send messages to the queue
 with ARN: {queueARN}.");
            }
        }
    }
```

- Find instructions and more code on GitHub.

Create a queue

The following code example shows how to create an Amazon SQS queue.

AWS SDK for .NET

```
    using System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using Amazon.SQS;
    using Amazon.SQS.Model;

    public class CreateQueue
    {
        /// <summary>
        /// Initializes the Amazon SQS client object and then calls the
        /// CreateQueueAsync method to create the new queue. If the call is
        /// successful, it displays the URL of the new queue on the console.
        /// </summary>
```

```
    public static async Task Main()
    {
        // If the Amazon SQS message queue is not in the same AWS Region as your
        // default user, you need to provide the AWS Region as a parameter to the
        // client constructor.
        var client = new AmazonSQSClient();

        string queueName = "New-Example-Queue";
        int maxMessage = 256 * 1024;
        var attrs = new Dictionary<string, string>
        {
            {
                QueueAttributeName.DelaySeconds,
                TimeSpan.FromSeconds(5).TotalSeconds.ToString()
            },
            {
                QueueAttributeName.MaximumMessageSize,
                maxMessage.ToString()
            },
            {
                QueueAttributeName.MessageRetentionPeriod,
                TimeSpan.FromDays(4).TotalSeconds.ToString()
            },
            {
                QueueAttributeName.ReceiveMessageWaitTimeSeconds,
                TimeSpan.FromSeconds(5).TotalSeconds.ToString()
            },
            {
                QueueAttributeName.VisibilityTimeout,
                TimeSpan.FromHours(12).TotalSeconds.ToString()
            },
        };

        var request = new CreateQueueRequest
        {
            Attributes = attrs,
            QueueName = queueName,
        };

        var response = await client.CreateQueueAsync(request);

        if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
        {
            Console.WriteLine("Successfully created Amazon SQS queue.");
            Console.WriteLine($"Queue URL: {response.QueueUrl}");
        }
    }
}
```

Create an Amazon SQS queue and send a message to it.

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon;
using Amazon.SQS;
using Amazon.SQS.Model;

public class CreateSendExample
{
    // Specify your AWS Region (an example Region is shown).
    private static readonly string QueueName = "Example_Queue";
    private static readonly RegionEndpoint ServiceRegion = RegionEndpoint.USWest2;
```

```
        private static IAmazonSQS client;

        public static async Task Main()
        {
            client = new AmazonSQSClient(ServiceRegion);
            var createQueueResponse = await CreateQueue(client, QueueName);

            string queueUrl = createQueueResponse.QueueUrl;

            Dictionary<string, MessageAttributeValue> messageAttributes = new
Dictionary<string, MessageAttributeValue>
            {
                { "Title",   new MessageAttributeValue { DataType = "String",
StringValue = "The Whistler" } },
                { "Author",  new MessageAttributeValue { DataType = "String",
StringValue = "John Grisham" } },
                { "WeeksOn", new MessageAttributeValue { DataType = "Number",
StringValue = "6" } },
            };

            string messageBody = "Information about current NY Times fiction bestseller
for week of 12/11/2016.";

            var sendMsgResponse = await SendMessage(client, queueUrl, messageBody,
messageAttributes);
        }

        /// <summary>
        /// Creates a new Amazon SQS queue using the queue name passed to it
        /// in queueName.
        /// </summary>
        /// <param name="client">An SQS client object used to send the message.</param>
        /// <param name="queueName">A string representing the name of the queue
        /// to create.</param>
        /// <returns>A CreateQueueResponse that contains information about the
        /// newly created queue.</returns>
        public static async Task<CreateQueueResponse> CreateQueue(IAmazonSQS client,
string queueName)
        {
            var request = new CreateQueueRequest
            {
                QueueName = queueName,
                Attributes = new Dictionary<string, string>
                {
                    { "DelaySeconds", "60" },
                    { "MessageRetentionPeriod", "86400" },
                },
            };

            var response = await client.CreateQueueAsync(request);
            Console.WriteLine($"Created a queue with URL : {response.QueueUrl}");

            return response;
        }

        /// <summary>
        /// Sends a message to an SQS queue.
        /// </summary>
        /// <param name="client">An SQS client object used to send the message.</param>
        /// <param name="queueUrl">The URL of the queue to which to send the
        /// message.</param>
        /// <param name="messageBody">A string representing the body of the
        /// message to be sent to the queue.</param>
        /// <param name="messageAttributes">Attributes for the message to be
        /// sent to the queue.</param>
        /// <returns>A SendMessageResponse object that contains information
```

```
        /// about the message that was sent.</returns>
        public static async Task<SendMessageResponse> SendMessage(
            IAmazonSQS client,
            string queueUrl,
            string messageBody,
            Dictionary<string, MessageAttributeValue> messageAttributes)
        {
            var sendMessageRequest = new SendMessageRequest
            {
                DelaySeconds = 10,
                MessageAttributes = messageAttributes,
                MessageBody = messageBody,
                QueueUrl = queueUrl,
            };

            var response = await client.SendMessageAsync(sendMessageRequest);
            Console.WriteLine($"Sent a message with id : {response.MessageId}");

            return response;
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see CreateQueue in *AWS SDK for .NET API Reference*.

Delete a message from a queue

The following code example shows how to delete a message from an Amazon SQS queue.

AWS SDK for .NET

```
    using System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using Amazon.SQS;
    using Amazon.SQS.Model;

    public class DeleteMessage
    {
        /// <summary>
        /// Initializes the Amazon SQS client object. It then calls the
        /// ReceiveMessageAsync method to retrieve information about the
        /// available methods before deleting them.
        /// </summary>
        public static async Task Main()
        {
            string queueUrl = "https://sqs.us-east-2.amazonaws.com/0123456789ab/
Example_Queue";
            var attributeNames = new List<string>() { "All" };
            int maxNumberOfMessages = 5;
            var visibilityTimeout = (int)TimeSpan.FromMinutes(10).TotalSeconds;
            var waitTimeSeconds = (int)TimeSpan.FromSeconds(5).TotalSeconds;

            // If the Amazon SQS message queue is not in the same AWS Region as your
            // default user, you need to provide the AWS Region as a parameter to the
            // client constructor.
            var client = new AmazonSQSClient();

            var request = new ReceiveMessageRequest
            {
```

```
                QueueUrl = queueUrl,
                AttributeNames = attributeNames,
                MaxNumberOfMessages = maxNumberOfMessages,
                VisibilityTimeout = visibilityTimeout,
                WaitTimeSeconds = waitTimeSeconds,
            };

            var response = await client.ReceiveMessageAsync(request);

            if (response.Messages.Count > 0)
            {
                response.Messages.ForEach(async m =>
                {
                    Console.Write($"Message ID: '{m.MessageId}'");

                    var delRequest = new DeleteMessageRequest
                    {
                        QueueUrl = "https://sqs.us-east-1.amazonaws.com/0123456789ab/
MyTestQueue",
                        ReceiptHandle = m.ReceiptHandle,
                    };

                    var delResponse = await client.DeleteMessageAsync(delRequest);
                });
            }
            else
            {
                Console.WriteLine("No messages to delete.");
            }
        }
    }
```

Receive a message from an Amazon SQS queue and then delete the message.

```
        public static async Task Main()
        {
            // If the AWS Region you want to use is different from
            // the AWS Region defined for the default user, supply
            // the specify your AWS Region to the client constructor.
            var client = new AmazonSQSClient();
            string queueName = "Example_Queue";

            var queueUrl = await GetQueueUrl(client, queueName);
            Console.WriteLine($"The SQS queue's URL is {queueUrl}");

            var response = await ReceiveAndDeleteMessage(client, queueUrl);

            Console.WriteLine($"Message: {response.Messages[0]}");
        }

        /// <summary>
        /// Retrieve the queue URL for the queue named in the queueName
        /// property using the client object.
        /// </summary>
        /// <param name="client">The Amazon SQS client used to retrieve the
        /// queue URL.</param>
        /// <param name="queueName">A string representing  name of the queue
        /// for which to retrieve the URL.</param>
        /// <returns>The URL of the queue.</returns>
        public static async Task<string> GetQueueUrl(IAmazonSQS client, string
 queueName)
        {
            var request = new GetQueueUrlRequest
```

```
            {
                QueueName = queueName,
            };

            GetQueueUrlResponse response = await client.GetQueueUrlAsync(request);
            return response.QueueUrl;
        }

        /// <summary>
        /// Retrieves the message from the quque at the URL passed in the
        /// queueURL parameters using the client.
        /// </summary>
        /// <param name="client">The SQS client used to retrieve a message.</param>
        /// <param name="queueUrl">The URL of the queue from which to retrieve
        /// a message.</param>
        /// <returns>The response from the call to ReceiveMessageAsync.</returns>
        public static async Task<ReceiveMessageResponse>
ReceiveAndDeleteMessage(IAmazonSQS client, string queueUrl)
        {
            // Receive a single message from the queue.
            var receiveMessageRequest = new ReceiveMessageRequest
            {
                AttributeNames = { "SentTimestamp" },
                MaxNumberOfMessages = 1,
                MessageAttributeNames = { "All" },
                QueueUrl = queueUrl,
                VisibilityTimeout = 0,
                WaitTimeSeconds = 0,
            };

            var receiveMessageResponse = await
client.ReceiveMessageAsync(receiveMessageRequest);

            // Delete the received message from the queue.
            var deleteMessageRequest = new DeleteMessageRequest
            {
                QueueUrl = queueUrl,
                ReceiptHandle = receiveMessageResponse.Messages[0].ReceiptHandle,
            };

            await client.DeleteMessageAsync(deleteMessageRequest);

            return receiveMessageResponse;
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see DeleteMessage in *AWS SDK for .NET API Reference*.

Delete a queue

The following code example shows how to delete an Amazon SQS queue.

AWS SDK for .NET

```
    using System;
    using System.Threading.Tasks;
    using Amazon.SQS;

    public class DeleteQueue
```

```
    {
        /// <summary>
        /// Initializes the Amazon SQS client object and then calls the
        /// DeleteQueueAsync method to delete the queue.
        /// </summary>
        public static async Task Main()
        {
            // If the Amazon SQS message queue is not in the same AWS Region as your
            // default user, you need to provide the AWS Region as a parameter to the
            // client constructor.
            var client = new AmazonSQSClient();

            string queueUrl = "https://sqs.us-east-2.amazonaws.com/0123456789ab/New-
Example-Queue";

            var response = await client.DeleteQueueAsync(queueUrl);

            if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
            {
                Console.WriteLine("Successfully deleted the queue.");
            }
            else
            {
                Console.WriteLine("Could not delete the crew.");
            }
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see DeleteQueue in *AWS SDK for .NET API Reference*.

Get attributes for a queue

The following code example shows how to get attributes for an Amazon SQS queue.

AWS SDK for .NET

```
    using System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using Amazon.SQS;
    using Amazon.SQS.Model;

    public class GetQueueAttributes
    {
        /// <summary>
        /// Initializes the Amazon SQS client and then uses it to call the
        /// GetQueueAttributesAsync method to retrieve the attributes for the
        /// Amazon SQS queue.
        /// </summary>
        public static async Task Main()
        {
            // If the Amazon SQS message queue is not in the same AWS Region as your
            // default user, you need to provide the AWS Region as a parameter to the
            // client constructor.
            var client = new AmazonSQSClient();

            var queueUrl = "https://sqs.us-east-2.amazonaws.com/0123456789ab/New-
Example-Queue";
            var attrs = new List<string>() { "All" };
```

```
            var request = new GetQueueAttributesRequest
            {
                QueueUrl = queueUrl,
                AttributeNames = attrs,
            };

            var response = await client.GetQueueAttributesAsync(request);

            if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
            {
                DisplayAttributes(response);
            }
        }

        /// <summary>
        /// Displays the attributes passed to the method on the console.
        /// </summary>
        /// <param name="attrs">The attributes for the Amazon SQS queue.</param>
        public static void DisplayAttributes(GetQueueAttributesResponse attrs)
        {
            Console.WriteLine($"Attributes for queue ARN '{attrs.QueueARN}':");
            Console.WriteLine($"  Approximate number of messages:
{attrs.ApproximateNumberOfMessages}");
            Console.WriteLine($"  Approximate number of messages delayed:
{attrs.ApproximateNumberOfMessagesDelayed}");
            Console.WriteLine($"  Approximate number of messages not visible:
{attrs.ApproximateNumberOfMessagesNotVisible}");
            Console.WriteLine($"  Queue created on: {attrs.CreatedTimestamp}");
            Console.WriteLine($"  Delay seconds: {attrs.DelaySeconds}");
            Console.WriteLine($"  Queue last modified on:
{attrs.LastModifiedTimestamp}");
            Console.WriteLine($"  Maximum message size: {attrs.MaximumMessageSize}");
            Console.WriteLine($"  Message retention period:
{attrs.MessageRetentionPeriod}");
            Console.WriteLine($"  Visibility timeout: {attrs.VisibilityTimeout}");
            Console.WriteLine($"  Policy: {attrs.Policy}\n");
            Console.WriteLine("  Attributes:");

            foreach (var attr in attrs.Attributes)
            {
                Console.WriteLine($"    {attr.Key}: {attr.Value}");
            }
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see GetQueueAttributes in *AWS SDK for .NET API Reference*.

Get the URL of a queue

The following code example shows how to get the URL of an Amazon SQS queue.

AWS SDK for .NET

```
    using System;
    using System.Threading.Tasks;
    using Amazon.SQS;
    using Amazon.SQS.Model;
```

```
    public class GetQueueUrl
    {
        /// <summary>
        /// Initializes the Amazon SQS client object and then calls the
        /// GetQueueUrlAsync method to retrieve the URL of an Amazon SQS
        /// queue.
        /// </summary>
        public static async Task Main()
        {
            // If the Amazon SQS message queue is not in the same AWS Region as your
            // default user, you need to provide the AWS Region as a parameter to the
            // client constructor.
            var client = new AmazonSQSClient();

            string queueName = "New-Exampe-Queue";

            try
            {
                var response = await client.GetQueueUrlAsync(queueName);

                if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
                {
                    Console.WriteLine($"The URL for {queueName} is:
{response.QueueUrl}");
                }
            }
            catch (QueueDoesNotExistException ex)
            {
                Console.WriteLine(ex.Message);
                Console.WriteLine($"The queue {queueName} was not found.");
            }
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see GetQueueUrl in *AWS SDK for .NET API Reference*.

Receive messages from a queue

The following code example shows how to receive messages from an Amazon SQS queue.

AWS SDK for .NET

```
    using System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using Amazon.SQS;
    using Amazon.SQS.Model;

    public class ReceiveFromQueue
    {
        public static async Task Main(string[] args)
        {
            string queueUrl = "https://sqs.us-east-2.amazonaws.com/0123456789ab/
Example_Queue";
            var attributeNames = new List<string>() { "All" };
            int maxNumberOfMessages = 5;
            var visibilityTimeout = (int)TimeSpan.FromMinutes(10).TotalSeconds;
            var waitTimeSeconds = (int)TimeSpan.FromSeconds(5).TotalSeconds;
```

```
            // If the Amazon SQS message queue is not in the same AWS Region as your
            // default user, you need to provide the AWS Region as a parameter to the
            // client constructor.
            var client = new AmazonSQSClient();

            var request = new ReceiveMessageRequest
            {
                QueueUrl = queueUrl,
                AttributeNames = attributeNames,
                MaxNumberOfMessages = maxNumberOfMessages,
                VisibilityTimeout = visibilityTimeout,
                WaitTimeSeconds = waitTimeSeconds,
            };

            var response = await client.ReceiveMessageAsync(request);

            if (response.Messages.Count > 0)
            {
                DisplayMessages(response.Messages);
            }
        }

        /// <summary>
        /// Display message information for a list of Amazon SQS messages.
        /// </summary>
        /// <param name="messages">The list of Amazon SQS Message objects to display.</
param>
        public static void DisplayMessages(List<Message> messages)
        {
            messages.ForEach(m =>
            {
                Console.WriteLine($"For message ID {m.MessageId}:");
                Console.WriteLine($"  Body: {m.Body}");
                Console.WriteLine($"  Receipt handle: {m.ReceiptHandle}");
                Console.WriteLine($"  MD5 of body: {m.MD5OfBody}");
                Console.WriteLine($"  MD5 of message attributes:
 {m.MD5OfMessageAttributes}");
                Console.WriteLine("  Attributes:");

                foreach (var attr in m.Attributes)
                {
                    Console.WriteLine($"\t {attr.Key}: {attr.Value}");
                }
            });
        }
    }
```

Receive a message from an Amazon SQS queue, and then delete the message.

```
        public static async Task Main()
        {
            // If the AWS Region you want to use is different from
            // the AWS Region defined for the default user, supply
            // the specify your AWS Region to the client constructor.
            var client = new AmazonSQSClient();
            string queueName = "Example_Queue";

            var queueUrl = await GetQueueUrl(client, queueName);
            Console.WriteLine($"The SQS queue's URL is {queueUrl}");

            var response = await ReceiveAndDeleteMessage(client, queueUrl);

            Console.WriteLine($"Message: {response.Messages[0]}");
```

```
        }

        /// <summary>
        /// Retrieve the queue URL for the queue named in the queueName
        /// property using the client object.
        /// </summary>
        /// <param name="client">The Amazon SQS client used to retrieve the
        /// queue URL.</param>
        /// <param name="queueName">A string representing  name of the queue
        /// for which to retrieve the URL.</param>
        /// <returns>The URL of the queue.</returns>
        public static async Task<string> GetQueueUrl(IAmazonSQS client, string
queueName)
        {
            var request = new GetQueueUrlRequest
            {
                QueueName = queueName,
            };

            GetQueueUrlResponse response = await client.GetQueueUrlAsync(request);
            return response.QueueUrl;
        }

        /// <summary>
        /// Retrieves the message from the quque at the URL passed in the
        /// queueURL parameters using the client.
        /// </summary>
        /// <param name="client">The SQS client used to retrieve a message.</param>
        /// <param name="queueUrl">The URL of the queue from which to retrieve
        /// a message.</param>
        /// <returns>The response from the call to ReceiveMessageAsync.</returns>
        public static async Task<ReceiveMessageResponse>
ReceiveAndDeleteMessage(IAmazonSQS client, string queueUrl)
        {
            // Receive a single message from the queue.
            var receiveMessageRequest = new ReceiveMessageRequest
            {
                AttributeNames = { "SentTimestamp" },
                MaxNumberOfMessages = 1,
                MessageAttributeNames = { "All" },
                QueueUrl = queueUrl,
                VisibilityTimeout = 0,
                WaitTimeSeconds = 0,
            };

            var receiveMessageResponse = await
client.ReceiveMessageAsync(receiveMessageRequest);

            // Delete the received message from the queue.
            var deleteMessageRequest = new DeleteMessageRequest
            {
                QueueUrl = queueUrl,
                ReceiptHandle = receiveMessageResponse.Messages[0].ReceiptHandle,
            };

            await client.DeleteMessageAsync(deleteMessageRequest);

            return receiveMessageResponse;
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see ReceiveMessage in *AWS SDK for .NET API Reference*.

Send a message to a queue

The following code example shows how to send a message to an Amazon SQS queue.

AWS SDK for .NET

```
    using System;
    using System.Threading.Tasks;
    using Amazon.SQS;
    using Amazon.SQS.Model;

    public class SendMessageToQueue
    {
        /// <summary>
        /// Initialize the Amazon SQS client object and use the
        /// SendMessageAsync method to send a message to an Amazon SQS queue.
        /// </summary>
        public static async Task Main()
        {
            string messageBody = "This is a sample message to send to the example
 queue.";
            string queueUrl = "https://sqs.us-east-2.amazonaws.com/0123456789ab/
Example_Queue";

            // Create an Amazon SQS client object using the
            // default user. If the AWS Region you want to use
            // is different, supply the AWS Region as a parameter.
            IAmazonSQS client = new AmazonSQSClient();

            var request = new SendMessageRequest
            {
                MessageBody = messageBody,
                QueueUrl = queueUrl,
            };

            var response = await client.SendMessageAsync(request);

            if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
            {
                Console.WriteLine($"Successfully sent message. Message ID:
 {response.MessageId}");
            }
            else
            {
                Console.WriteLine("Could not send message.");
            }
        }
    }
```

Create an Amazon SQS queue and send a message to it.

```
    using System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using Amazon;
    using Amazon.SQS;
    using Amazon.SQS.Model;

    public class CreateSendExample
    {
```

```
        // Specify your AWS Region (an example Region is shown).
        private static readonly string QueueName = "Example_Queue";
        private static readonly RegionEndpoint ServiceRegion = RegionEndpoint.USWest2;
        private static IAmazonSQS client;

        public static async Task Main()
        {
            client = new AmazonSQSClient(ServiceRegion);
            var createQueueResponse = await CreateQueue(client, QueueName);

            string queueUrl = createQueueResponse.QueueUrl;

            Dictionary<string, MessageAttributeValue> messageAttributes = new
Dictionary<string, MessageAttributeValue>
            {
                { "Title",   new MessageAttributeValue { DataType = "String",
StringValue = "The Whistler" } },
                { "Author",  new MessageAttributeValue { DataType = "String",
StringValue = "John Grisham" } },
                { "WeeksOn", new MessageAttributeValue { DataType = "Number",
StringValue = "6" } },
            };

            string messageBody = "Information about current NY Times fiction bestseller
for week of 12/11/2016.";

            var sendMsgResponse = await SendMessage(client, queueUrl, messageBody,
messageAttributes);
        }

        /// <summary>
        /// Creates a new Amazon SQS queue using the queue name passed to it
        /// in queueName.
        /// </summary>
        /// <param name="client">An SQS client object used to send the message.</param>
        /// <param name="queueName">A string representing the name of the queue
        /// to create.</param>
        /// <returns>A CreateQueueResponse that contains information about the
        /// newly created queue.</returns>
        public static async Task<CreateQueueResponse> CreateQueue(IAmazonSQS client,
string queueName)
        {
            var request = new CreateQueueRequest
            {
                QueueName = queueName,
                Attributes = new Dictionary<string, string>
                {
                    { "DelaySeconds", "60" },
                    { "MessageRetentionPeriod", "86400" },
                },
            };

            var response = await client.CreateQueueAsync(request);
            Console.WriteLine($"Created a queue with URL : {response.QueueUrl}");

            return response;
        }

        /// <summary>
        /// Sends a message to an SQS queue.
        /// </summary>
        /// <param name="client">An SQS client object used to send the message.</param>
        /// <param name="queueUrl">The URL of the queue to which to send the
        /// message.</param>
        /// <param name="messageBody">A string representing the body of the
        /// message to be sent to the queue.</param>
```

```
        /// <param name="messageAttributes">Attributes for the message to be
        /// sent to the queue.</param>
        /// <returns>A SendMessageResponse object that contains information
        /// about the message that was sent.</returns>
        public static async Task<SendMessageResponse> SendMessage(
            IAmazonSQS client,
            string queueUrl,
            string messageBody,
            Dictionary<string, MessageAttributeValue> messageAttributes)
        {
            var sendMessageRequest = new SendMessageRequest
            {
                DelaySeconds = 10,
                MessageAttributes = messageAttributes,
                MessageBody = messageBody,
                QueueUrl = queueUrl,
            };

            var response = await client.SendMessageAsync(sendMessageRequest);
            Console.WriteLine($"Sent a message with id : {response.MessageId}");

            return response;
        }
    }
```

- Find instructions and more code on GitHub.
- For API details, see SendMessage in *AWS SDK for .NET API Reference*.

Cross-service examples using AWS SDK for .NET

The following sample applications use the AWS SDK for .NET to work across multiple AWS services.

Examples

- Build a publish and subscription application that translates messages (p. 312)
- Create a dynamic web application to track DynamoDB data (p. 313)
- Detect objects in images with Amazon Rekognition using an AWS SDK (p. 313)

Build a publish and subscription application that translates messages

AWS SDK for .NET

Shows how to use the Amazon Simple Notification Service .NET API to create a web application that has subscription and publish functionality. In addition, this example application also translates messages.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- Amazon SNS
- Amazon Translate

Create a dynamic web application to track DynamoDB data

AWS SDK for .NET

Shows how to use the Amazon DynamoDB .NET API to create a dynamic web application that tracks DynamoDB work data.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB
- Amazon SES

Detect objects in images with Amazon Rekognition using an AWS SDK

AWS SDK for .NET

Shows how to use Amazon Rekognition .NET API to create an app that uses Amazon Rekognition to identify objects by category in images located in an Amazon Simple Storage Service (Amazon S3) bucket. The app sends the admin an email notification with the results using Amazon Simple Email Service (Amazon SES).

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- Amazon Rekognition
- Amazon S3
- Amazon SES

Programming AWS OpsWorks to Work with stacks and applications

The AWS SDK for .NET supports AWS OpsWorks, which provides a simple and flexible way to create and manage stacks and applications. With AWS OpsWorks, you can provision AWS resources, manage their configuration, deploy applications to those resources, and monitor their health. For more information, see the OpsWorks product page and the AWS OpsWorks User Guide.

APIs

The AWS SDK for .NET provides APIs for AWS OpsWorks. The APIs enable you to work with AWS OpsWorks features such as stacks with their layers, instances, and apps. To view the full set of APIs, see the AWS SDK for .NET API Reference (and scroll to "Amazon.OpsWorks").

The AWS OpsWorks APIs are provided by the AWSSDK.OpsWorks NuGet package.

Prerequisites

Before you begin, be sure you have set up your environment (p. 15). Also review the information in Setting up your project (p. 17) and SDK features (p. 49).

Support for other AWS services and configuration

The AWS SDK for .NET supports AWS services in addition to those described in the preceding sections. For information about the APIs for all supported services, see the AWS SDK for .NET API Reference.

In addition to the namespaces for individual AWS services, the AWS SDK for .NET also provides the following APIs:

Area	Description	Resources
AWS Support	Programmatic access to AWS Support cases and Trusted Advisor features.	See Amazon.AWSSupport and Amazon.AWSSupport.Model.
General	Helper classes and enumerations.	See Amazon and Amazon.Util.

Security for this AWS Product or Service

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 this AWS product or service

The AWS shared responsibility model applies to data protection in this AWS product or service. 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. This content includes 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 user accounts with 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 recommend TLS 1.2 or later.
- Set up API and user activity logging with AWS CloudTrail.
- 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 personal data that is stored in Amazon S3.
- If you require FIPS 140-2 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) 140-2.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form fields such as a **Name** field. This includes when you work with this AWS product or service or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form 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 for this AWS Product or Service

AWS Identity and Access Management (IAM) is an Amazon Web Services (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 resources in AWS services. IAM is an AWS service that you can use with no additional charge.

To use this AWS product or service to access AWS, you need an AWS account and AWS credentials. To increase the security of your AWS account, we recommend that you use an *IAM user* to provide access credentials instead of using your AWS account credentials.

For details about working with IAM, see AWS Identity and Access Management.

For an overview of IAM users and why they are important for the security of your account, see AWS Security Credentials in the Amazon Web Services General Reference.

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.

Compliance Validation for this AWS Product or Service

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.

The security and compliance of AWS services is assessed by third-party auditors as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others. AWS provides a frequently updated list of AWS services in scope of specific compliance programs at AWS Services in Scope by Compliance Program.

Third-party audit reports are available for you to download using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

For more information about AWS compliance programs, see AWS Compliance Programs.

Your compliance responsibility when using this AWS product or service to access an AWS service is determined by the sensitivity of your data, your organization's compliance objectives, and applicable laws and regulations. If your use of an AWS service is subject to compliance with standards such as HIPAA, PCI, or FedRAMP, AWS provides resources to help:

- Security and Compliance Quick Start Guides – Deployment guides that discuss architectural considerations and provide steps for deploying security-focused and compliance-focused baseline environments on AWS.
- Architecting for HIPAA Security and Compliance Whitepaper – A whitepaper that describes how companies can use AWS to create HIPAA-compliant applications.
- AWS Compliance Resources – A collection of workbooks and guides that might apply to your industry and location.
- AWS Config – A service that assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – A comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

Resilience for this AWS Product or Service

The Amazon Web Services (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 Availability 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 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.

Enforcing a minimum TLS version in the AWS SDK for .NET

To increase security when communicating with AWS services, you should configure the AWS SDK for .NET to use TLS 1.2 or later.

The AWS SDK for .NET uses the underlying .NET runtime to determine which security protocol to use. By default, current versions of .NET use the latest configured protocol that the operating system supports. Your application can override this SDK behavior, but it's *not recommended* to do so.

.NET Core

By default, .NET Core uses the latest configured protocol that the operating system supports. The AWS SDK for .NET doesn't provide a mechanism to override this.

If you're using a .NET Core version earlier than 2.1, we *strongly* recommend you upgrade your .NET Core version.

See the following for information specific to each operating system.

Windows

Modern distributions of Windows have TLS 1.2 support enabled by default. If you're running on Windows 7 SP1 or Windows Server 2008 R2 SP1, you need to ensure that TLS 1.2 support is enabled in the registry, as described at https://docs.microsoft.com/en-us/windows-server/security/tls/tls-registry-settings#tls-12. If you're running an earlier distribution, you must upgrade your operating system. For information about TLS 1.3 support in Windows, check the latest Microsoft documentation for the minimum required client or server versions.

macOS

If you're running .NET Core 2.1 or later, TLS 1.2 is enabled by default. TLS 1.2 is supported by OS X Mavericks v10.9 or later. .NET Core version 2.1 and later require newer versions of macOS, as described at https://docs.microsoft.com/en-us/dotnet/core/install/dependencies?tabs=netcore21&pivots=os-macos.

If you're using .NET Core 1.0, .NET Core uses OpenSSL on macOS, a dependency that must be installed separately. OpenSSL added support for TLS 1.2 in version 1.0.1, and added support for TLS 1.3 in version 1.1.1.

Linux

.NET Core on Linux requires OpenSSL, which comes bundled with many Linux distributions. But it can also be installed separately. OpenSSL added support for TLS 1.2 in version 1.0.1, and added support for TLS 1.3 in version 1.1.1. If you're using a modern version of .NET Core (2.1 or later) and have installed a package manager, it's likely that a more modern version of OpenSSL was installed for you.

To be sure, you can run **`openssl version`** in a terminal and verify that the version is later than 1.0.1.

.NET Framework

If you're running a modern version of .NET Framework (4.7 or later) and a modern version of Windows (at least Windows 8 for clients, Windows Server 2012 or later for servers), TLS 1.2 is enabled and used by default.

If you're using a .NET Framework runtime that doesn't use the operating system settings (.NET Framework 3.5 through 4.5.2), the AWS SDK for .NET will attempt to add support for TLS 1.1 and TLS 1.2 to the supported protocols. If you're using .NET Framework 3.5, this will be successful only if the appropriate hot patch is installed, as follows:

- Windows 10 version 1511 and Windows Server 2016 – KB3156421
- Windows 8.1 and Windows Server 2012 R2 – KB3154520
- Windows Server 2012 – KB3154519

- Windows 7 SP1 and Server 2008 R2 SP1 – KB3154518

If your application is running on a newer .NET Framework on Windows 7 SP1 or Windows Server 2008 R2 SP1, you need to ensure that TLS 1.2 support is enabled in the registry, as described at https://docs.microsoft.com/en-us/windows-server/security/tls/tls-registry-settings#tls-12. Newer versions of Windows have it enabled by default.

For detailed best practices for using TLS with .NET Framework, see the Microsoft article at https://docs.microsoft.com/en-us/dotnet/framework/network-programming/tls.

AWS Tools for PowerShell

AWS Tools for PowerShell use the AWS SDK for .NET for all calls to AWS services. The behavior of your environment depends on the version of Windows PowerShell you're running, as follows.

Windows PowerShell 2.0 through 5.x

Windows PowerShell 2.0 through 5.x run on .NET Framework. You can verify which .NET runtime (2.0 or 4.0) is being used by PowerShell by using the following command.

```
$PSVersionTable.CLRVersion
```

- When using .NET Runtime 2.0, follow the instructions provided earlier regarding the AWS SDK for .NET and .NET Framework 3.5.
- When using .NET Runtime 4.0, follow the instructions provided earlier regarding the AWS SDK for .NET and .NET Framework 4+.

Windows PowerShell 6.0

Windows PowerShell 6.0 and newer run on .NET Core. You can verify which version of .NET Core is being used by running the following command.

```
[System.Reflection.Assembly]::GetEntryAssembly().GetCustomAttributes([System.Runtime.Versioning.TargetF
 $true).FrameworkName
```

Follow the instructions provided earlier regarding the AWS SDK for .NET and the relevant version of .NET Core.

Xamarin

For Xamarin, see the directions at https://docs.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/transport-layer-security. In summary:

For Android

- Requires Android 5.0 or later.
- **Project Properties**, **Android Options**: HttpClient implementation must be set to **Android** and the SSL/TLS implementation set to **Native TLS 1.2+**.

For iOS

- Requires iOS 7 or later.
- **Project Properties**, **iOS Build**: HttpClient implementation must be set to **NSUrlSession**.

For macOS

- Requires macOS 10.9 or later.
- **Project Options**, **Build**, **Mac Build**: HttpClient implementation must be set to **NSUrlSession**.

Unity

You must use Unity 2018.2 or later, and use the .NET 4.x Equivalent scripting runtime. You can set this in **Project Settings**, **Configuration**, **Player**, as described at https://docs.unity3d.com/2019.1/Documentation/Manual/ScriptingRuntimeUpgrade.html. The .NET 4.x Equivalent scripting runtime enables TLS 1.2 support to all Unity platforms running Mono or IL2CPP. For more information, see https://blogs.unity3d.com/2018/07/11/scripting-runtime-improvements-in-unity-2018-2/.

Browser (for Blazor WebAssembly)

WebAssembly runs in the browser instead of on the server, and uses the browser for handling HTTP traffic. Therefore, TLS support is determined by browser support.

Blazor WebAssembly, in preview for ASP.NET Core 3.1, is supported only in browsers that support WebAssembly, as described at https://docs.microsoft.com/en-us/aspnet/core/blazor/supported-platforms. All mainstream browsers supported TLS 1.2 before supporting WebAssembly. If this is the case for your browser, then if your app runs, it can communicate over TLS 1.2.

See your browser's documentation for more information and verification.

Amazon S3 Encryption Client Migration

This topic shows how to migrate your applications from Version 1 (V1) of the Amazon Simple Storage Service (Amazon S3) encryption client to Version 2 (V2), and ensure application availability throughout the migration process.

Objects that are encrypted with the V2 client can't be decrypted with the V1 client. In order to ease migration to the new client without having to re-encrypt all objects at once, a "V1-transitional" client has been provided. This client can *decrypt* both V1- and V2-encrypted objects, but *encrypts* objects only in V1-compatible format. The V2 client can *decrypt* both V1- and V2-encrypted objects (when enabled for V1 objects), but *encrypts* objects only in V2-compatible format.

Migration Overview

This migration happens in three phases. These phases are introduced here and described in detail later. Each phase must be completed for *all* clients that use shared objects before the next phase is started.

1. **Update existing clients to V1-transitional clients to read new formats.** First, update your applications to take a dependency on the V1-transitional client instead of the V1 client. The V1-transitional client enables your existing code to decrypt objects written by the new V2 clients and objects written in V1-compatible format.

   Note
   The V1-transitional client is provided for migration purposes only. Proceed to upgrading to the V2 client after moving to the V1-transitional client.
2. **Migrate V1-transitional clients to V2 clients to write new formats.** Next, replace all V1-transitional clients in your applications with V2 clients, and set the security profile to `V2AndLegacy`. Setting this security profile on V2 clients enables those clients to decrypt objects that were encrypted in V1-compatible format.

3. **Update V2 clients to no longer read V1 formats.** Finally, after all clients have been migrated to V2 and all objects have been encrypted or re-encrypted in V2-compatible format, set the V2 security profile to `V2` instead of `V2AndLegacy`. This prevents the decryption of objects that are in V1-compatible format.

Update Existing Clients to V1-transitional Clients to Read New Formats

The V2 encryption client uses encryption algorithms that older versions of the client don't support. The first step in the migration is to update your V1 decryption clients so that they can read the new format.

The V1-transitional client enables your applications to decrypt both V1- and V2-encrypted objects. This client is a part of the Amazon.Extensions.S3.Encryption NuGet package. Perform the following steps on each of your applications to use the V1-transitional client.

1. Take a new dependency on the Amazon.Extensions.S3.Encryption package. If your project depends directly on the **AWSSDK.S3** or **AWSSDK.KeyManagementService** packages, you must either update those dependencies or remove them so that their updated versions will be pulled in with this new package.
2. Change the appropriate `using` statement from `Amazon.S3.Encryption` to `Amazon.Extensions.S3.Encryption`, as follows:

```
// using Amazon.S3.Encryption;
  using Amazon.Extensions.S3.Encryption;
```

3. Rebuild and redeploy your application.

The V1-transitional client is fully API-compatible with the V1 client, so no other code changes are required.

Migrate V1-transitional Clients to V2 Clients to Write New Formats

The V2 client is a part of the Amazon.Extensions.S3.Encryption NuGet package. It enables your applications to decrypt both V1- and V2-encrypted objects (if configured to do so), but encrypts objects only in V2-compatible format.

After updating your existing clients to read the new encryption format, you can proceed to safely update your applications to the V2 encryption and decryption clients. Perform the following steps on each of your applications to use the V2 client:

1. Change `EncryptionMaterials` to `EncryptionMaterialsV2`.
    a. When using KMS:
        i. Provide a KMS key ID.
        ii. Declare the encryption method that you are using; that is, `KmsType.KmsContext`.
        iii. Provide an encryption context to KMS to associate with this data key. You can send an empty dictionary (Amazon encryption context will still be merged in), but providing additional context is encouraged.
    b. When using user-provided key wrap methods (symmetric or asymmetric encryption):
        i. Provide an `AES` or an `RSA` instance that contains the encryption materials.

ii. Declare which encryption algorithm to use; that is, `SymmetricAlgorithmType.AesGcm` or `AsymmetricAlgorithmType.RsaOaepSha1`.

2. Change `AmazonS3CryptoConfiguration` to `AmazonS3CryptoConfigurationV2` with the `SecurityProfile` property set to `SecurityProfile.V2AndLegacy`.
3. Change `AmazonS3EncryptionClient` to `AmazonS3EncryptionClientV2`. This client takes the newly converted `AmazonS3CryptoConfigurationV2` and `EncryptionMaterialsV2` objects from the previous steps.

Example: KMS to KMS+Context

Pre-migration

```
using System.Security.Cryptography;
using Amazon.S3.Encryption;

var encryptionMaterial = new EncryptionMaterials("1234abcd-12ab-34cd-56ef-1234567890ab");
var configuration = new AmazonS3CryptoConfiguration()
{
    StorageMode = CryptoStorageMode.ObjectMetadata
};
var encryptionClient = new AmazonS3EncryptionClient(configuration, encryptionMaterial);
```

Post-migration

```
using System.Security.Cryptography;
using Amazon.Extensions.S3.Encryption;
using Amazon.Extensions.S3.Encryption.Primitives;

var encryptionContext = new Dictionary<string, string>();
var encryptionMaterial = new EncryptionMaterialsV2("1234abcd-12ab-34cd-56ef-1234567890ab",
 KmsType.KmsContext, encryptionContext);
var configuration = new AmazonS3CryptoConfigurationV2(SecurityProfile.V2AndLegacy)
{
    StorageMode = CryptoStorageMode.ObjectMetadata
};
var encryptionClient = new AmazonS3EncryptionClientV2(configuration, encryptionMaterial);
```

Example: Symmetric Algorithm (AES-CBC to AES-GCM Key Wrap)

`StorageMode` can be either `ObjectMetadata` or `InstructionFile`.

Pre-migration

```
using System.Security.Cryptography;
using Amazon.S3.Encryption;

var symmetricAlgorithm = Aes.Create();
var encryptionMaterial = new EncryptionMaterials(symmetricAlgorithm);
var configuration = new AmazonS3CryptoConfiguration()
{
    StorageMode = CryptoStorageMode.ObjectMetadata
};
var encryptionClient = new AmazonS3EncryptionClient(configuration, encryptionMaterial);
```

Post-migration

```
using System.Security.Cryptography;
using Amazon.Extensions.S3.Encryption;
using Amazon.Extensions.S3.Encryption.Primitives;

var symmetricAlgorithm = Aes.Create();
var encryptionMaterial = new EncryptionMaterialsV2(symmetricAlgorithm,
 SymmetricAlgorithmType.AesGcm);
var configuration = new AmazonS3CryptoConfigurationV2(SecurityProfile.V2AndLegacy)
{
    StorageMode = CryptoStorageMode.ObjectMetadata
};
var encryptionClient = new AmazonS3EncryptionClientV2(configuration, encryptionMaterial);
```

Note
When decrypting with AES-GCM, read the entire object to the end before you start using the decrypted data. This is to verify that the object hasn't been modified since it was encrypted.

Example: Asymmetric Algorithm (RSA to RSA-OAEP-SHA1 Key Wrap)

`StorageMode` can be either `ObjectMetadata` or `InstructionFile`.

Pre-migration

```
using System.Security.Cryptography;
using Amazon.S3.Encryption;

var asymmetricAlgorithm = RSA.Create();
var encryptionMaterial = new EncryptionMaterials(asymmetricAlgorithm);
var configuration = new AmazonS3CryptoConfiguration()
{
    StorageMode = CryptoStorageMode.ObjectMetadata
};
var encryptionClient = new AmazonS3EncryptionClient(configuration, encryptionMaterial);
```

Post-migration

```
using System.Security.Cryptography;
using Amazon.Extensions.S3.Encryption;
using Amazon.Extensions.S3.Encryption.Primitives;

var asymmetricAlgorithm = RSA.Create();
var encryptionMaterial = new EncryptionMaterialsV2(asymmetricAlgorithm,
 AsymmetricAlgorithmType.RsaOaepSha1);
var configuration = new AmazonS3CryptoConfigurationV2(SecurityProfile.V2AndLegacy)
{
    StorageMode = CryptoStorageMode.ObjectMetadata
};
var encryptionClient = new AmazonS3EncryptionClientV2(configuration, encryptionMaterial);
```

Update V2 Clients to No Longer Read V1 Formats

Eventually, all objects will have been encrypted or re-encrypted using a V2 client. *After this conversion is complete*, you can disable V1 compatibility in the V2 clients by setting the `SecurityProfile` property to `SecurityProfile.V2`, as shown in the following snippet.

```
//var configuration = new AmazonS3CryptoConfigurationV2(SecurityProfile.V2AndLegacy);
```

```
var configuration = new AmazonS3CryptoConfigurationV2(SecurityProfile.V2);
```

Special considerations for the AWS SDK for .NET

This section contains considerations for special cases where the normal configurations or procedures aren't appropriate or sufficient.

Topics

Obtaining assemblies for the AWS SDK for .NET

This topic describes how you can obtain the AWSSDK assemblies and store them locally (or on premises) for use in your projects. This is **not** the recommended method for handling SDK references, but is required in some environments.

> **Note**
> The recommended method for handling SDK references is to download and install just the NuGet packages that each project needs. That method is described in Install AWSSDK packages with NuGet (p. 28).

If you can't or aren't allowed to download and install NuGet packages on a per-project basis, the following options are available to you.

Download and extract ZIP files

(Remember that this isn't the recommended method (p. 28) for handling references to the AWS SDK for .NET.)

1. Download one of the following ZIP files:
   - aws-sdk-netstandard2.0.zip
   - aws-sdk-net45.zip
   - aws-sdk-net35.zip
2. Extract the assemblies to some "download" folder on your file system; it doesn't matter where. Make note of this folder.
3. When you set up your project, you get the required assemblies from this folder, as described in Install AWSSDK assemblies without NuGet (p. 30).

Install the MSI on Windows

(Remember that this isn't the recommended method (p. 28) for handling references to the AWS SDK for .NET.)

If you are required to install an MSI instead of using NuGet or one of the methods described earlier, you can find the ***legacy*** MSI at https://sdk-for-net.amazonwebservices.com/latest/AWSToolsAndSDKForNet.msi.

By default, the AWS SDK for .NET is installed in the `Program Files` folder, which requires administrator privileges. To install the SDK as a non-administrator, choose a different folder.

Accessing credentials and profiles in an application

The preferred method for using credentials is to allow the AWS SDK for .NET to find and retrieve them for you, as described in Credential and profile resolution (p. 24).

However, you can also configure your application to actively retrieve profiles and credentials, and then explicitly use those credentials when creating an AWS service client.

To actively retrieve profiles and credentials, use classes from the Amazon.Runtime.CredentialManagement namespace.

- To find a profile in a file that uses the AWS credentials file format (either the shared AWS credentials file in its default location (p. 20) or a custom credentials file), use the SharedCredentialsFile class. Files in this format are sometimes simply called *credentials files* in this text for brevity.
- To find a profile in the SDK Store, use the NetSDKCredentialsFile class.
- To search in both a credentials file and the SDK Store, depending on the configuration of a class property, use the CredentialProfileStoreChain class.

  You can use this class to find profiles. You can also use this class to request AWS credentials directly instead of using the `AWSCredentialsFactory` class (described next).
- To retrieve or create various types of credentials from a profile, use the AWSCredentialsFactory class.

The following sections provide examples for these classes.

Examples for class CredentialProfileStoreChain

You can get credentials or profiles from the CredentialProfileStoreChain class by using the TryGetAWSCredentials or TryGetProfile methods. The `ProfilesLocation` property of the class determines the behavior of the methods, as follows:

- If `ProfilesLocation` is null or empty, search the SDK Store if the platform supports it, and then search the shared AWS credentials file in the default location.
- If the `ProfilesLocation` property contains a value, search the credentials file specified in the property.

Get credentials from the SDK Store or the shared AWS credentials file

This example shows you how to get credentials by using the `CredentialProfileStoreChain` class and then use the credentials to create an AmazonS3Client object. The credentials can come from the SDK Store or from the shared AWS credentials file at the default location.

This example also uses the Amazon.Runtime.AWSCredentials class.

```
var chain = new CredentialProfileStoreChain();
```

```
AWSCredentials awsCredentials;
if (chain.TryGetAWSCredentials("some_profile", out awsCredentials))
{
    // Use awsCredentials to create an Amazon S3 service client
    using (var client = new AmazonS3Client(awsCredentials))
    {
        var response = await client.ListBucketsAsync();
        Console.WriteLine($"Number of buckets: {response.Buckets.Count}");
    }
}
```

Get a profile from the SDK Store or the shared AWS credentials file

This example shows you how to get a profile by using the CredentialProfileStoreChain class. The credentials can come from the SDK Store or from the shared AWS credentials file at the default location.

This example also uses the CredentialProfile class.

```
var chain = new CredentialProfileStoreChain();
CredentialProfile basicProfile;
if (chain.TryGetProfile("basic_profile", out basicProfile))
{
    // Use basicProfile
}
```

Get credentials from a custom credentials file

This example shows you how to get credentials by using the CredentialProfileStoreChain class. The credentials come from a file that uses the AWS credentials file format but is at an alternate location.

This example also uses the Amazon.Runtime.AWSCredentials class.

```
var chain = new
    CredentialProfileStoreChain("c:\\Users\\sdkuser\\customCredentialsFile.ini");
AWSCredentials awsCredentials;
if (chain.TryGetAWSCredentials("basic_profile", out awsCredentials))
{
    // Use awsCredentials to create an AWS service client
}
```

Examples for classes SharedCredentialsFile and AWSCredentialsFactory

Create an AmazonS3Client by using the SharedCredentialsFile class

This examples shows you how to find a profile in the shared AWS credentials file, create AWS credentials from the profile, and then use the credentials to create an AmazonS3Client object. The example uses the SharedCredentialsFile class.

This example also uses the CredentialProfile class and the Amazon.Runtime.AWSCredentials class.

```
CredentialProfile basicProfile;
```

```
AWSCredentials awsCredentials;
var sharedFile = new SharedCredentialsFile();
if (sharedFile.TryGetProfile("basic_profile", out basicProfile) &&
    AWSCredentialsFactory.TryGetAWSCredentials(basicProfile, sharedFile, out
 awsCredentials))
{
    // use awsCredentials to create an Amazon S3 service client
    using (var client = new AmazonS3Client(awsCredentials, basicProfile.Region))
    {
        var response = await client.ListBucketsAsync();
        Console.WriteLine($"Number of buckets: {response.Buckets.Count}");
    }
}
```

Note
The NetSDKCredentialsFile class can be used in exactly the same way, except you would instantiate a new NetSDKCredentialsFile object instead of a SharedCredentialsFile object.

Special considerations for Unity support

When using the AWS SDK for .NET and .NET Standard 2.0 for your Unity application, your application must reference the AWS SDK for .NET assemblies (DLL files) directly rather than using NuGet. Given this requirement, the following are important actions you will need to perform.

- You need to obtain the AWS SDK for .NET assemblies and apply them to your project. For information about how to do this, see Download and extract ZIP files (p. 325) in the topic Obtaining AWSSDK assemblies (p. 325).

- You need to include the following DLLs in your Unity project alongside the DLLs for **AWSSDK.Core** and the other AWS services you're using. Starting with version 3.5.109 of the AWS SDK for .NET, the .NET Standard ZIP file contains these additional DLLs.
  - Microsoft.Bcl.AsyncInterfaces.dll
  - System.Runtime.CompilerServices.Unsafe.dll
  - System.Threading.Tasks.Extensions.dll

- If you're using IL2CPP to build your Unity project, you must add a `link.xml` file to your Asset folder to prevent code stripping. The `link.xml` file must list all of the AWSSDK assemblies you are using, and each must include the `preserve="all"` attribute. The following snippet shows an example of this file.

```
<linker>
    <assembly fullname="AWSSDK.Core" preserve="all"/>
    <assembly fullname="AWSSDK.DynamoDBv2" preserve="all"/>
    <assembly fullname="AWSSDK.Lambda" preserve="all"/>
</linker>
```

Note
To read interesting background information related to this requirement, see the article at http://aws.amazon.com/blogs/developer/referencing-the-aws-sdk-for-net-standard-2-0-from-unity-xamarin-or-uwp/.

In addition to these special considerations, see What's changed for version 3.5 (p. 93) for information about migrating your Unity application to version 3.5 of the AWS SDK for .NET.

Special considerations for Xamarin support

Xamarin projects (new and existing) must target .NET Standard 2.0. See .NET Standard 2.0 Support in Xamarin.Forms and .NET implementation support.

Also see the information about Portable Class Library and Xamarin (p. 91).

API reference for the AWS SDK for .NET

The AWS SDK for .NET provides an API that you can use to access AWS services. To see what classes and methods are available in the API, see the AWS SDK for .NET API Reference.

In addition to the general reference given above, each of the examples under the Code examples with guidance (p. 97) section contains references to the specific methods and classes that are used in that example.

Document history

The following table describes the important changes since the last release of the *AWS SDK for .NET Developer Guide*. For notification about updates to this documentation, you can subscribe to an RSS feed.

update-history-change	update-history-description	update-history-date
Retiring EC2-Classic (p. 331)	Added notes about retiring EC2-Classic.	April 13, 2022
Single sign-on with the AWS SDK for .NET (p. 57)	Added information about single sign-on (SSO) when using the AWS SDK for .NET.	March 17, 2022
Enforcing a minimum TLS version (p. 317)	Added information about TLS 1.3.	March 16, 2022
Working with AWS services (p. 97)	Included lists of the code examples that are available on GitHub.	February 28, 2022
Enabling SDK Metrics (p. 331)	Removed information about enabling SDK metrics, which has been deprecated.	January 20, 2022
AWS .NET deployment tool (p. 75)	Added a reference to the AWS Toolkit for Visual Studio, which provides deployment functionality that is similar to the AWS .NET deployment tool.	October 26, 2021
AWS SDK for .NET version 3 guide consolidation (p. 331)	The two AWS SDK for .NET version 3 developer guides, "V3" and "latest", have been consolidated into one guide under the "v3" URL.	August 18, 2021
Working with configuration files (p. 84)	Added information about JSON configuration files for the AWS .NET deployment tool.	July 26, 2021
Migrating from .NET Standard 1.3 (p. 95)	Support for .NET Standard 1.3 on the AWS SDK for .NET has come to its end of life.	March 25, 2021
AWS .NET deployment tool (preview) (p. 75)	Added preview information about the AWS .NET deployment tool, which you can use to deploy an application from the .NET CLI.	March 15, 2021
Version 3.5 of the AWS SDK for .NET (p. 93)	Version 3.5 of the AWS SDK for .NET has been released.	August 25, 2020

Paginators (p. 52)	Added paginators to many service clients, which make pagination of API results more convenient.	August 24, 2020
Retries and timeouts (p. 50)	Added information about retry modes.	August 20, 2020
S3 encryption client migration (p. 320)	Added information about how to migrate your Amazon S3 encryption clients from V1 to V2.	August 7, 2020
Using KMS keys for S3 encryption (p. 190)	Updated example to use version 2 of the S3 encryption client.	August 6, 2020
Migrating from .NET Standard 1.3 (p. 95)	Added information about ending support for .NET Standard 1.3 at the end of 2020.	May 18, 2020
Quick start (p. 5)	Added a quick-start section with basic setup and tutorials to introduce the reader to the AWS SDK for .NET.	March 27, 2020
Enforcing TLS 1.2 (p. 317)	Added information about how to enforce TLS 1.2 in the SDK.	March 10, 2020