AWS SDK for Java

Developer Guide

AWS SDK for Java: Developer Guide

Table of Contents

AWS SDK for Java Developer Guide

The AWS SDK for Java provides a Java API for Amazon Web Services. Using the SDK, you can easily build Java applications that work with Amazon S3, Amazon EC2, Amazon SimpleDB, and more. We regularly add support for new services to the AWS SDK for Java. For a list of the supported services and their API versions that are included with each release of the SDK, view the release notes for the version that you're working with.

AWS SDK for Java 2.0 Developer Preview

Take a look at the new AWS SDK for Java 2.0 developer preview at https://github.com/aws/aws-sdk-java-v2/. It includes much awaited features, such as a way to plug in a HTTP implementation. To get started, see the AWS SDK for Java 2.0 Developer Guide.

Additional Documentation and Resources

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

- AWS SDK for Java API Reference
- Java developer blog
- Java developer forums
- GitHub:
  - Documentation source
  - Documentation issues
  - SDK source
  - SDK issues
  - SDK samples
  - Gitter channel
- @awsforjava (Twitter)
- release notes

Eclipse IDE Support

If you develop code using the Eclipse IDE, you can use the AWS Toolkit for Eclipse to add the AWS SDK for Java to an existing Eclipse project or to create a new AWS SDK for Java project. The toolkit also supports creating and uploading Lambda functions, launching and monitoring Amazon EC2 instances, managing IAM users and security groups, a AWS CloudFormation template editor, and more.

See the AWS Toolkit for Eclipse User Guide for full documentation.

Developing AWS Applications for Android

If you're an Android developer, Amazon Web Services publishes an SDK made specifically for Android development: the AWS Mobile SDK for Android. See the AWS Mobile SDK for Android Developer Guide for full documentation.

Viewing the SDK's Revision History

To view the release history of the AWS SDK for Java, including changes and supported services per SDK version, see the SDK's release notes.

Building Java Reference Documentation for Earlier SDK versions

The AWS SDK for Java API Reference represents the most recent version of the SDK. If you're using an earlier SDK version, you might want to access the SDK reference documentation that matches the version you're using.

The easiest way to build the documentation is using Apache's Maven build tool. *Download and install Maven first if you don't already have it on your system*, then use the following instructions to build the reference documentation.

To build reference documentation for an earlier SDK version

1. Locate and select the SDK version that you're using on the releases page of the SDK repository on GitHub.
2. Choose either the `zip` (most platforms, including Windows) or `tar.gz` (Linux, macOS, or Unix) link to download the SDK to your computer.
3. Unpack the archive to a local directory.
4. On the command line, navigate to the directory where you unpacked the archive, and type the following.

```
mvn javadoc:javadoc
```

5. After building is complete, you'll find the generated HTML documentation in the `aws-java-sdk/target/site/apidocs/` directory.

Getting Started

This section provides information about how to install, set up, and use the AWS SDK for Java.

Topics

Sign Up for AWS and Create an IAM User

To use the AWS SDK for Java to access Amazon Web Services (AWS), you will 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 root account credentials.

Note
For an overview of IAM users and why they are important for the security of your account, see Overview of Identity Management: Users in the *IAM User Guide*.

To sign up for AWS

1. Open https://aws.amazon.com/ and click **Sign Up**.
2. Follow the on-screen instructions. Part of the sign-up procedure involves receiving a phone call and entering a PIN using your phone keypad.

Next, create an IAM user and download (or copy) its secret access key.

To create an IAM user

1. Go to the IAM console (you may need to sign in to AWS first).
2. Click **Users** in the sidebar to view your IAM users.

3. If you don't have any IAM users set up, click **Create New Users** to create one.
4. Select the IAM user in the list that you'll use to access AWS.
5. Open the **Security Credentials** tab, and click **Create Access Key**.

   Note
   You can have a maximum of two active access keys for any given IAM user. If your IAM user has two access keys already, then you'll need to delete one of them before creating a new key.
6. On the resulting dialog box, click the **Download Credentials** button to download the credential file to your computer, or click **Show User Security Credentials** to view the IAM user's access key ID and secret access key (which you can copy and paste).

   Important
   There is no way to obtain the secret access key once you close the dialog box. You can, however, delete its associated access key ID and create a new one.

Next, you should set your credentials (p. 6) in the AWS shared credentials file or in the environment.

Note
If you use the Eclipse IDE, you should consider installing the AWS Toolkit for Eclipse and providing your credentials as described in Set up AWS Credentials in the *AWS Toolkit for Eclipse User Guide*.

Set up the AWS SDK for Java

Describes how to use the AWS SDK for Java in your project.

Prerequisites

To use the AWS SDK for Java, you must have:

- a suitable Java Development Environment (p. 6).
- An AWS account and access keys. For instructions, see Sign Up for AWS and Create an IAM User (p. 3).
- AWS credentials (access keys) set in your environment or using the shared (by the AWS CLI and other SDKs) credentials file. For more information, see Set up AWS Credentials and Region for Development (p. 6).

Including the SDK in your project

To include the SDK your project, use one of the following methods depending on your build system or IDE:

- **Apache Maven**– If you use Apache Maven, you can specify the entire SDK (or specific SDK components) as dependencies in your project. See Using the SDK with Apache Maven (p. 8) for details about how to set up the SDK when using Maven.
- **Gradle**– If you use Gradle, you can import the Maven Bill of Materials (BOM) in your Gradle project to automatically manage SDK dependencies. See Using the SDK with Gradle (p. 10) for more infomation.
- **Eclipse IDE**– If you use the Eclipse IDE, you may want to install and use the AWS Toolkit for Eclipse, which will automatically download, install and update the Java SDK for you. For more information and setup instructions, see the AWS Toolkit for Eclipse User Guide.

If you intend to build your projects using a different IDE, with Apache Ant or by any other means, then download and extract the SDK as shown in the next section.

Downloading and extracting the SDK

We recommend that you use the most recent pre-built version of the SDK for new projects, which provides you with the latest support for all AWS services.

> **Note**
> For information about how to download and build previous versions of the SDK, see Installing previous versions of the SDK (p. 5).

To download and extract the latest version of the SDK

1. Download the SDK from https://sdk-for-java.amazonwebservices.com/latest/aws-java-sdk.zip.
2. After downloading the SDK, extract the contents into a local directory.

The SDK contains the following directories:

- `documentation`– contains the API documentation (also available on the web: AWS SDK for Java API Reference).
- `lib`– contains the SDK `.jar` files.
- `samples`– contains working sample code that demonstrates how to use the SDK.
- `third-party`– contains third-party libraries that are used by the SDK, such as Apache commons logging, AspectJ and the Spring framework.

To use the SDK, add the full path to the `lib` and `third-party` directories to the dependencies in your build file, and add them to your java `CLASSPATH` to run your code.

Installing previous versions of the SDK

Only the latest version of the SDK is provided in pre-built form. However, you can build a previous version of the SDK using Apache Maven (open source). Maven will download all necessary dependencies, build and install the SDK in one step. Visit http://maven.apache.org/ for installation instructions and more information.

To install a previous version of the SDK

1. Go to the SDK's GitHub page at: AWS SDK for Java (GitHub).
2. Choose the tag corresponding to the version number of the SDK that you want. For example, `1.6.10`.
3. Click the **Download ZIP** button to download the version of the SDK you selected.
4. Unzip the file to a directory on your development system. On many systems, you can use your graphical file manager to do this, or use the `unzip` utility in a terminal window.
5. In a terminal window, navigate to the directory where you unzipped the SDK source.
6. Build and install the SDK with the following command (Maven required):

   ```
   mvn clean install
   ```

   The resulting `.jar` file is built into the `target` directory.
7. (Optional) Build the API Reference documentation using the following command:

   ```
   mvn javadoc:javadoc
   ```

The documentation is built into the `target/site/apidocs/` directory.

Installing a Java Development Environment

The AWS SDK for Java requires J2SE Development Kit *6.0 or later*. You can download the latest Java software from http://www.oracle.com/technetwork/java/javase/downloads/.

Important
Java version 1.6 (JS2E 6.0) did not have built-in support for SHA256-signed SSL certificates, which are required for all HTTPS connections with AWS after September 30, 2015.
Java versions 1.7 or newer are packaged with updated certificates and are unaffected by this issue.

Choosing a JVM

For the best performance of your server-based applications with the AWS SDK for Java, we recommend that you use the *64-bit version* of the Java Virtual Machine (JVM). This JVM runs only in server mode, even if you specify the `-Client` option at run time.

Using the 32-bit version of the JVM with the `-Server` option at run time should provide comparable performance to the 64-bit JVM.

Set up AWS Credentials and Region for Development

To connect to any of the supported services with the AWS SDK for Java, you must provide AWS credentials. The AWS SDKs and CLIs use *provider chains* to look for AWS credentials in a number of different places, including system/user environment variables and local AWS configuration files.

This topic provides basic information about setting up your AWS credentials for local application development using the AWS SDK for Java. If you need to set up credentials for use within an EC2 instance or if you're using the Eclipse IDE for development, refer to the following topics instead:

- When using an EC2 instance, create an IAM role and then give your EC2 instance access to that role as shown in Using IAM Roles to Grant Access to AWS Resources on Amazon EC2 (p. 58).
- Set up AWS credentials within Eclipse using the AWS Toolkit for Eclipse. See Set up AWS Credentials in the AWS Toolkit for Eclipse User Guide for more information.

Setting AWS Credentials

Setting your credentials for use by the AWS SDK for Java can be done in a number of ways, but here are the recommended approaches:

- Set credentials in the AWS credentials profile file on your local system, located at:
  - `~/.aws/credentials` on Linux, macOS, or Unix
  - `C:\Users\`*`USERNAME`*` \.aws\credentials` on Windows

  This file should contain lines in the following format:

```
[default]
aws_access_key_id = your_access_key_id
```

```
aws_secret_access_key = your_secret_access_key
```

Substitute your own AWS credentials values for the values *your_access_key_id* and *your_secret_access_key*.

- Set the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables.

  To set these variables on Linux, macOS, or Unix, use **`export`** :

  ```
  export AWS_ACCESS_KEY_ID=your_access_key_id
  export AWS_SECRET_ACCESS_KEY=your_secret_access_key
  ```

  To set these variables on Windows, use **`set`** :

  ```
  set AWS_ACCESS_KEY_ID=your_access_key_id
  set AWS_SECRET_ACCESS_KEY=your_secret_access_key
  ```
- For an EC2 instance, specify an IAM role and then give your EC2 instance access to that role. See IAM Roles for Amazon EC2 in the *Amazon EC2 User Guide for Linux Instances* for a detailed discussion about how this works.

Once you have set your AWS credentials using one of these methods, they will be loaded automatically by the AWS SDK for Java by using the default credential provider chain. For further information about working with AWS credentials in your Java applications, see Working with AWS Credentials (p. 14).

Setting the AWS Region

You should set a default AWS Region that will be used for accessing AWS services with the AWS SDK for Java. For the best network performance, you should choose a region that's geographically close to you (or to your customers).

> **Note**
> If you *don't* select a region, then us-east-1 will be used by default.

You can use similar techniques to setting credentials to set your default AWS region:

- Set the AWS Region in the AWS config file on your local system, located at:
  - `~/.aws/config` on Linux, macOS, or Unix
  - `C:\Users\USERNAME\.aws\config` on Windows

  This file should contain lines in the following format:

  ```
  [default]
  region = your_aws_region
  ```

  Substitute your desired AWS Region (for example, "us-west-2") for *your_aws_region*.
- Set the `AWS_REGION` environment variable.

  On Linux, macOS, or Unix, use **`export`** :

  ```
  export AWS_REGION=your_aws_region
  ```

  On Windows, use **`set`** :

  ```
  set AWS_REGION=your_aws_region
  ```

Where *your_aws_region* is the desired AWS Region name.

Using the SDK with Apache Maven

You can use Apache Maven to configure and build AWS SDK for Java projects, or to build the SDK itself.

Note
You must have Maven installed to use the guidance in this topic. If it isn't already installed, visit http://maven.apache.org/ to download and install it.

Create a new Maven package

To create a basic Maven package, open a terminal (command-line) window and run:

```
mvn -B archetype:generate \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DgroupId=org.example.basicapp \
  -DartifactId=myapp
```

Replace *org.example.basicapp* with the full package namespace of your application, and *myapp* with the name of your project (this will become the name of the directory for your project).

By default, Maven creates a project template for you using the quickstart archetype, which is a good starting place for many projects. There are more archetypes available; visit the Maven archetypes page for a list of archetypes packaged with Maven. You can choose a particular archetype to use by adding the `-DarchetypeArtifactId` argument to the `archetype:generate` command. For example:

```
mvn archetype:generate \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-webapp \
  -DgroupId=org.example.webapp \
  -DartifactId=mywebapp
```

Note
Much more information about creating and configuring Maven projects is provided in the Maven Getting Started Guide.

Configure the SDK as a Maven dependency

To use the AWS SDK for Java in your project, you'll need to declare it as a dependency in your project's `pom.xml` file. Beginning with version 1.9.0, you can import individual components (p. 8) or the entire SDK (p. 9).

Specifying individual SDK modules

To select individual SDK modules, use the AWS SDK for Java bill of materials (BOM) for Maven, which will ensure that the modules you specify use the same version of the SDK and that they're compatible with each other.

To use the BOM, add a `<dependencyManagement>` section to your application's `pom.xml` file, adding `aws-java-sdk-bom` as a dependency and specifying the version of the SDK you want to use:

```
<dependencyManagement>
```

```
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-java-sdk-bom</artifactId>
      <version>1.11.106</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

To view the latest version of the AWS SDK for Java BOM that is available on Maven Central, visit: https://mvnrepository.com/artifact/com.amazonaws/aws-java-sdk-bom. You can also use this page to see which modules (dependencies) are managed by the BOM that you can include within the `<dependencies>` section of your project's `pom.xml` file.

You can now select individual modules from the SDK that you use in your application. Because you already declared the SDK version in the BOM, you don't need to specify the version number for each component.

```
<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk-s3</artifactId>
  </dependency>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk-dynamodb</artifactId>
  </dependency>
</dependencies>
```

Importing all SDK modules

If you would like to pull in the *entire* SDK as a dependency, don't use the BOM method, but simply declare it in your `pom.xml` like this:

```
<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk</artifactId>
    <version>1.11.106</version>
  </dependency>
</dependencies>
```

Build your project

Once you have your project set up, you can build it using Maven's `package` command:

```
mvn package
```

This will create your `.jar` file in the `target` directory.

Build the SDK with Maven

You can use Apache Maven to build the SDK from source. To do so, download the SDK code from GitHub, unpack it locally, and then execute the following Maven command:

```
mvn clean install
```

Using the SDK with Gradle

To use the AWS SDK for Java in your Gradle project, use Spring's dependency management plugin for Gradle, which can be used to import the SDK's Maven Bill of Materials (BOM) to manage SDK dependencies for your project.

To configure the SDK for Gradle

1. Add the dependency management plugin to your `build.gradle` file

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "io.spring.gradle:dependency-management-plugin:1.0.0.RC2"
    }
}

apply plugin: "io.spring.dependency-management"
```

2. Add the BOM to the *dependencyManagement* section of the file

```
dependencyManagement {
    imports {
        mavenBom 'com.amazonaws:aws-java-sdk-bom:1.10.77'
    }
}
```

3. Specify the SDK modules that you'll be using in the *dependencies* section

```
dependencies {
    compile 'com.amazonaws:aws-java-sdk-s3'
    testCompile group: 'junit', name: 'junit', version: '4.11'
}
```

Gradle will automatically resolve the correct version of your SDK dependencies using the information from the BOM.

Note
For more detail about specifying SDK dependencies using the BOM, see Using the SDK with Apache Maven (p. 8).

Using the AWS SDK for Java

This section provides important general information about programming with the AWS SDK for Java that applies to all services you might use with the SDK.

For service-specific programming information and examples (for Amazon EC2, Amazon S3, Amazon SWF, etc.), see Programming Examples (p. 34).

Topics

Best Practices for AWS Development with the AWS SDK for Java

The following best practices can help you avoid issues or trouble as you develop AWS applications with the AWS SDK for Java. We've organized best practices by service.

Amazon S3

Avoid ResetExceptions

When you upload objects to Amazon S3 by using streams (either through an AmazonS3 client or TransferManager), you might encounter network connectivity or timeout issues. By default, the AWS SDK for Java attempts to retry failed transfers by marking the input stream before the start of a transfer and then resetting it before retrying.

If the stream doesn't support mark and reset, the SDK throws a ResetException when there are transient failures and retries are enabled.

Best Practice

We recommend that you use streams that support mark and reset operations.

The most reliable way to avoid a ResetException is to provide data by using a File or FileInputStream, which the AWS SDK for Java can handle without being constrained by mark and reset limits.

If the stream isn't a FileInputStream but does support mark and reset, you can set the mark limit by using the `setReadLimit` method of RequestClientOptions. Its default value is 128 KB. Setting the read limit value to *one byte greater than the size of stream* will reliably avoid a ResetException.

For example, if the maximum expected size of a stream is 100,000 bytes, set the read limit to 100,001 (100,000 + 1) bytes. The mark and reset will always work for 100,000 bytes or less. Be aware that this might cause some streams to buffer that number of bytes into memory.

Creating Service Clients

To make requests to Amazon Web Services, you first create a service client object. The recommended way is to use the service client builder.

Each AWS service has a service interface with methods for each action in the service API. For example, the service interface for Amazon DynamoDB is named AmazonDynamoDB. Each service interface has a corresponding client builder you can use to construct an implementation of the service interface. The client builder class for DynamoDB is named AmazonDynamoDBClientBuilder.

Obtaining a Client Builder

To obtain an instance of the client builder, use the static factory method `standard`, as shown in the following example.

```
AmazonDynamoDBClientBuilder builder = AmazonDynamoDBClientBuilder.standard();
```

Once you have a builder, you can customize the client's properties by using many fluent setters in the builder API. For example, you can set a custom region and a custom credentials provider, as follows.

```
AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.standard()
                        .withRegion(Regions.US_WEST_2)
                        .withCredentials(new ProfileCredentialsProvider("myProfile"))
                        .build();
```

Note

The fluent `withXXX` methods return the `builder` object so that you can chain the method calls for convenience and for more readable code. After you configure the properties you want, you can call the `build` method to create the client. Once a client is created, it's immutable and any calls to `setRegion` or `setEndpoint` will fail.

A builder can create multiple clients with the same configuration. When you're writing your application, be aware that the builder is mutable and not thread-safe.

The following code uses the builder as a factory for client instances.

```
public class DynamoDBClientFactory {
    private final AmazonDynamoDBClientBuilder builder =
        AmazonDynamoDBClientBuilder.standard()
            .withRegion(Regions.US_WEST_2)
            .withCredentials(new ProfileCredentialsProvider("myProfile"));

    public AmazonDynamoDB createClient() {
        return builder.build();
    }
}
```

The builder also exposes fluent setters for ClientConfiguration' and RequestMetricCollector, and a custom list of RequestHandler2.

The following is a complete example that overrides all configurable properties.

```
AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.standard()
        .withRegion(Regions.US_WEST_2)
        .withCredentials(new ProfileCredentialsProvider("myProfile"))
        .withClientConfiguration(new ClientConfiguration().withRequestTimeout(5000))
        .withMetricsCollector(new MyCustomMetricsCollector())
        .withRequestHandlers(new MyCustomRequestHandler(), new MyOtherCustomRequestHandler)
        .build();
```

Creating Async Clients

The AWS SDK for Java has asynchronous (or async) clients for every service (except for Amazon S3), and a corresponding async client builder for every service.

To create an async DynamoDB client with the default ExecutorService

```
AmazonDynamoDBAsync ddbAsync = AmazonDynamoDBAsyncClientBuilder.standard()
        .withRegion(Regions.US_WEST_2)
        .withCredentials(new ProfileCredentialsProvider("myProfile"))
        .build();
```

In addition to the configuration options that the synchronous (or sync) client builder supports, the async client enables you to set a custom ExecutorFactory to change the `ExecutorService` that the async client uses. `ExecutorFactory` is a functional interface, so it interoperates with Java 8 lambda expressions and method references.

To create an async client with a custom executor

```
AmazonDynamoDBAsync ddbAsync = AmazonDynamoDBAsyncClientBuilder.standard()
            .withExecutorFactory(() -> Executors.newFixedThreadPool(10))
            .build();
```

Using DefaultClient

Both the sync and async client builders have another factory method named `defaultClient`. This method creates a service client with the default configuration, using the default provider chain to load credentials and the AWS Region. If credentials or the region can't be determined from the environment that the application is running in, the call to `defaultClient` fails. See Working with AWS Credentials (p. 14) and AWS Region Selection (p. 18) for more information about how credentials and region are determined.

To create a default service client

```
AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();
```

Client Lifecycle

Service clients in the SDK are thread-safe and, for best performance, you should treat them as long-lived objects. Each client has its own connection pool resource that is shut down when the client is garbage collected. To explicitly shut down a client, call the `shutdown` method. After calling `shutdown`, all client resources are released and the client is unusable.

To shut down a client

```
AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();
ddb.shutdown();
// Client is now unusable
```

Working with AWS Credentials

To make requests to Amazon Web Services, you must supply AWS credentials to the AWS SDK for Java. You can do this in the following ways:

- Use the default credential provider chain *(recommended)*.
- Use a specific credential provider or provider chain (or create your own).
- Supply the credentials yourself. These can be root account credentials, IAM credentials, or temporary credentials retrieved from AWS STS.

Important
For security, we *strongly recommend* that you *use IAM users* instead of the root account for AWS access. For more information, see IAM Best Practices in the *IAM User Guide*.

Using the Default Credential Provider Chain

When you initialize a new service client without supplying any arguments, the AWS SDK for Java attempts to find AWS credentials by using the *default credential provider chain* implemented by the DefaultAWSCredentialsProviderChain class. The default credential provider chain looks for credentials in this order:

1. **Environment variables**–`AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`. The AWS SDK for Java uses the EnvironmentVariableCredentialsProvider class to load these credentials.
2. **Java system properties**–`aws.accessKeyId` and `aws.secretKey`. The AWS SDK for Java uses the SystemPropertiesCredentialsProvider to load these credentials.
3. **The default credential profiles file**– typically located at `~/.aws/credentials` (location can vary per platform), and shared by many of the AWS SDKs and by the AWS CLI. The AWS SDK for Java uses the ProfileCredentialsProvider to load these credentials.

You can create a credentials file by using the `aws configure` command provided by the AWS CLI, or you can create it by editing the file with a text editor. For information about the credentials file format, see AWS Credentials File Format (p. 16).

4. **Amazon ECS container credentials**– loaded from the Amazon ECS if the environment variable `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` is set. The AWS SDK for Java uses the ContainerCredentialsProvider to load these credentials.
5. **Instance profile credentials**– used on EC2 instances, and delivered through the Amazon EC2 metadata service. The AWS SDK for Java uses the InstanceProfileCredentialsProvider to load these credentials.

   Note
   Instance profile credentials are used only if `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` is not set. See EC2ContainerCredentialsProviderWrapper for more information.

Setting Credentials

To be able to use AWS credentials, they must be set in *at least one* of the preceding locations. For information about setting credentials, see the following topics:

- To specify credentials in the *environment* or in the default *credential profiles file*, see Set up AWS Credentials and Region for Development (p. 6).
- To set Java *system properties*, see the System Properties tutorial on the official *Java Tutorials* website.
- To set up and use *instance profile credentials* with your EC2 instances, see Using IAM Roles to Grant Access to AWS Resources on Amazon EC2 (p. 58).

Setting an Alternate Credentials Profile

The AWS SDK for Java uses the *default* profile by default, but there are ways to customize which profile is sourced from the credentials file.

You can use the AWS Profile environment variable to change the profile loaded by the SDK.

For example, on Linux, macOS, or Unix you would run the following command to change the profile to *myProfile*.

```
export AWS_PROFILE="myProfile"
```

On Windows you would use the following.

```
set AWS_PROFILE="myProfile"
```

Setting the AWS_PROFILE environment variable affects credential loading for all officially supported AWS SDKs and Tools (including the AWS CLI and the AWS CLI for PowerShell). To change only the profile for a Java application, you can use the system property *aws.profile* instead.

Note
The environment variable takes precedence over the system property.

Setting an Alternate Credentials File Location

The AWS SDK for Java loads AWS credentials automatically from the default credentials file location. However, you can also specify the location by setting the `AWS_CREDENTIAL_PROFILES_FILE` environment variable with the full path to the credentials file.

You can use this feature to temporarily change the location where the AWS SDK for Java looks for your credentials file (for example, by setting this variable with the command line). Or you can set the environment variable in your user or system environment to change it for the user or systemwide.

To override the default credentials file location

- Set the `AWS_CREDENTIAL_PROFILES_FILE` environment variable to the location of your AWS credentials file.
  - On Linux, macOS, or Unix, use **`export`** :

```
export AWS_CREDENTIAL_PROFILES_FILE=path/to/credentials_file
```

  - On Windows, use **`set`** :

```
set AWS_CREDENTIAL_PROFILES_FILE=path/to/credentials_file
```

AWS Credentials File Format

When you use the `aws configure` command to create an AWS credentials file, the command creates a file with the following format.

```
[default]
aws_access_key_id={YOUR_ACCESS_KEY_ID}
aws_secret_access_key={YOUR_SECRET_ACCESS_KEY}

[profile2]
aws_access_key_id={YOUR_ACCESS_KEY_ID}
aws_secret_access_key={YOUR_SECRET_ACCESS_KEY}
```

The profile name is specified in square brackets (for example, `[default]`), followed by the configurable fields in that profile as key-value pairs. You can have multiple profiles in your credentials file, which can be added or edited using `aws configure --profile` *`PROFILE_NAME`* to select the profile to configure.

You can specify additional fields, such as `aws_session_token`, `metadata_service_timeout`, and `metadata_service_num_attempts`. These are not configurable with the CLI—you must edit the file by hand if you want to use them. For more information about the configuration file and its available fields, see Configuring the AWS Command Line Interface in the *AWS CLI User Guide*.

Loading Credentials

After you set credentials, you can load them by using the default credential provider chain.

To do this, you instantiate an AWS Service client without explicitly providing credentials to the builder, as follows.

```
AmazonS3 s3Client = AmazonS3ClientBuilder.standard()
                       .withRegion(Regions.US_WEST_2)
                       .build();
```

Specifying a Credential Provider or Provider Chain

You can specify a credential provider that is different from the *default* credential provider chain by using the client builder.

You provide an instance of a credentials provider or provider chain to a client builder that takes an AWSCredentialsProvider interface as input. The following example shows how to use *environment* credentials specifically.

```
AmazonS3 s3Client = AmazonS3ClientBuilder.standard()
                       .withCredentials(new EnvironmentVariableCredentialsProvider())
                       .build();
```

For the full list of AWS SDK for Java-supplied credential providers and provider chains, see **All Known Implementing Classes** in AWSCredentialsProvider.

Note
You can use this technique to supply credential providers or provider chains that you create by using your own credential provider that implements the `AWSCredentialsProvider` interface, or by subclassing the AWSCredentialsProviderChain class.

Explicitly Specifying Credentials

If the default credential chain or a specific or custom provider or provider chain doesn't work for your code, you can set credentials that you supply explicitly. If you've retrieved temporary credentials using AWS STS, use this method to specify the credentials for AWS access.

To explicitly supply credentials to an AWS client

1. Instantiate a class that provides the AWSCredentials interface, such as BasicAWSCredentials, and supply it with the AWS access key and secret key you will use for the connection.
2. Create an AWSStaticCredentialsProvider with the `AWSCredentials` object.
3. Configure the client builder with the `AWSStaticCredentialsProvider` and build the client.

The following is an example.

```
BasicAWSCredentials awsCreds = new BasicAWSCredentials("access_key_id", "secret_key_id");
AmazonS3 s3Client = AmazonS3ClientBuilder.standard()
                        .withCredentials(new AWSStaticCredentialsProvider(awsCreds))
                        .build();
```

When using temporary credentials obtained from STS (p. 136), create a BasicSessionCredentials object, passing it the STS-supplied credentials and session token.

```
BasicSessionCredentials sessionCredentials = new BasicSessionCredentials(
   session_creds.getAccessKeyId(),
   session_creds.getSecretAccessKey(),
   session_creds.getSessionToken());

AmazonS3 s3 = AmazonS3ClientBuilder.standard()
                        .withCredentials(new
 AWSStaticCredentialsProvider(sessionCredentials)
                        .build();
```

More Info

- Sign Up for AWS and Create an IAM User (p. 3)
- Set up AWS Credentials and Region for Development (p. 6)
- Using IAM Roles to Grant Access to AWS Resources on Amazon EC2 (p. 58)

AWS Region Selection

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

Checking for Service Availability in an AWS Region

To see if a particular AWS service is available in a region, use the `isServiceSupported` method on the region that you'd like to use.

```
Region.getRegion(Regions.US_WEST_2)
    .isServiceSupported(AmazonDynamoDB.ENDPOINT_PREFIX);
```

See the Regions class documentation for the regions you can specify, and use the endpoint prefix of the service to query. Each service's endpoint prefix is defined in the service interface. For example, the DynamoDB endpoint prefix is defined in AmazonDynamoDB.

Choosing a Region

Beginning with version 1.4 of the AWS SDK for Java, you can specify a region name and the SDK will automatically choose an appropriate endpoint for you. To choose the endpoint yourself, see Choosing a Specific Endpoint (p. 18).

To explicitly set a region, we recommend that you use the Regions enum. This is an enumeration of all publicly available regions. To create a client with a region from the enum, use the following code.

```
AmazonEC2 ec2 = AmazonEC2ClientBuilder.standard()
                    .withRegion(Regions.US_WEST_2)
                    .build();
```

If the region you are attempting to use isn't in the `Regions` enum, you can set the region using a *string* that represents the name of the region.

```
AmazonEC2 ec2 = AmazonEC2ClientBuilder.standard()
                    .withRegion("us-west-2")
                    .build();
```

Note
After you build a client with the builder, it's *immutable* and the region *cannot be changed*. If you are working with multiple AWS Regions for the same service, you should create multiple clients—one per region.

Choosing a Specific Endpoint

Each AWS client can be configured to use a *specific endpoint* within a region by calling the `setEndpoint` method.

For example, to configure the Amazon EC2 client to use the EU (Ireland) Region, use the following code.

```
AmazonEC2 ec2 = new AmazonEC2(myCredentials);
ec2.setEndpoint("https://ec2.eu-west-1.amazonaws.com");
```

See Regions and Endpoints for the current list of regions and their corresponding endpoints for all AWS services.

Automatically Determine the AWS Region from the Environment

Important
This section applies only when using a client builder (p. 12) to access AWS services. AWS clients created by using the client constructor will not automatically determine region from the environment and will, instead, use the *default* SDK region (USEast1).

When running on Amazon EC2 or Lambda, you might want to configure clients to use the same region that your code is running on. This decouples your code from the environment it's running in and makes it easier to deploy your application to multiple regions for lower latency or redundancy.

You must use client builders to have the SDK automatically detect the region your code is running in.

To use the default credential/region provider chain to determine the region from the environment, use the client builder's `defaultClient` method.

```
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();
```

This is the same as using `standard` followed by `build`.

```
AmazonEC2 ec2 = AmazonEC2ClientBuilder.standard()
                    .build();
```

If you don't explicitly set a region using the `withRegion` methods, the SDK consults the default region provider chain to try and determine the region to use.

Default Region Provider Chain

The following is the region lookup process:

1. Any explicit region set by using `withRegion` or `setRegion` on the builder itself takes precedence over anything else.
2. The `AWS_REGION` environment variable is checked. If it's set, that region is used to configure the client.

   Note
   This environment variable is set by the Lambda container.
3. The SDK checks the AWS shared configuration file (usually located at `~/.aws/config`). If the *region* property is present, the SDK uses it.
   - The `AWS_CONFIG_FILE` environment variable can be used to customize the location of the shared config file.
   - The `AWS_PROFILE` environment variable or the *aws.profile* system property can be used to customize the profile that is loaded by the SDK.
4. The SDK attempts to use the Amazon EC2 instance metadata service to determine the region of the currently running Amazon EC2 instance.
5. If the SDK still hasn't found a region by this point, client creation fails with an exception.

When developing AWS applications, a common approach is to use the *shared configuration file* (described in Using the Default Credential Provider Chain (p. 14)) to set the region for local development, and rely on the default region provider chain to determine the region when running on AWS infrastructure. This greatly simplifies client creation and keeps your application portable.

Exception Handling

Understanding how and when the AWS SDK for Java throws exceptions is important to building high-quality applications using the SDK. The following sections describe the different cases of exceptions that are thrown by the SDK and how to handle them appropriately.

Why Unchecked Exceptions?

The AWS SDK for Java uses runtime (or unchecked) exceptions instead of checked exceptions for these reasons:

- To allow developers fine-grained control over the errors they want to handle without forcing them to handle exceptional cases they aren't concerned about (and making their code overly verbose)
- To prevent scalability issues inherent with checked exceptions in large applications

In general, checked exceptions work well on small scales, but can become troublesome as applications grow and become more complex.

For more information about the use of checked and unchecked exceptions, see:

- Unchecked Exceptions—The Controversy
- The Trouble with Checked Exceptions
- Java's checked exceptions were a mistake (and here's what I would like to do about it)

AmazonServiceException (and Subclasses)

AmazonServiceException is the most common exception that you'll experience when using the AWS SDK for Java. This exception represents an error response from an AWS service. For example, if you try to terminate an Amazon EC2 instance that doesn't exist, EC2 will return an error response and all the details of that error response will be included in the `AmazonServiceException` that's thrown. For some cases, a subclass of `AmazonServiceException` is thrown to allow developers fine-grained control over handling error cases through catch blocks.

When you encounter an `AmazonServiceException`, you know that your request was successfully sent to the AWS service but couldn't be successfully processed. This can be because of errors in the request's parameters or because of issues on the service side.

`AmazonServiceException` provides you with information such as:

- Returned HTTP status code
- Returned AWS error code
- Detailed error message from the service
- AWS request ID for the failed request

`AmazonServiceException` also includes information about whether the failed request was the caller's fault (a request with illegal values) or the AWS service's fault (an internal service error).

AmazonClientException

AmazonClientException indicates that a problem occurred inside the Java client code, either while trying to send a request to AWS or while trying to parse a response from AWS. An `AmazonClientException` is generally more severe than an `AmazonServiceException`, and indicates a major problem that is preventing the client from making service calls to AWS services. For example, the AWS SDK for Java throws an `AmazonClientException` if no network connection is available when you try to call an operation on one of the clients.

Asynchronous Programming

You can use either *synchronous* or *asynchronous* methods to call operations on AWS services. Synchronous methods block your thread's execution until the client receives a response from the service. Asynchronous methods return immediately, giving control back to the calling thread without waiting for a response.

Because an asynchronous method returns before a response is available, you need a way to get the response when it's ready. The AWS SDK for Java provides two ways: *Future objects* and *callback methods*.

Java Futures

Asynchronous methods in the AWS SDK for Java return a Future object that contains the results of the asynchronous operation *in the future*.

Call the `Future` `isDone()` method to see if the service has provided a response object yet. When the response is ready, you can get the response object by calling the `Future` `get()` method. You can use this mechanism to periodically poll for the asynchronous operation's results while your application continues to work on other things.

Here is an example of an asynchronous operation that calls a Lambda function, receiving a `Future` that can hold an InvokeResult object. The `InvokeResult` object is retrieved only after `isDone()` is `true`.

```
import com.amazonaws.services.lambda.AWSLambdaAsyncClient;
import com.amazonaws.services.lambda.model.InvokeRequest;
import com.amazonaws.services.lambda.model.InvokeResult;
import java.nio.ByteBuffer;
import java.util.concurrent.Future;
import java.util.concurrent.ExecutionException;

public class InvokeLambdaFunctionAsync
{
    public static void main(String[] args)
    {
        String function_name = "HelloFunction";
        String function_input = "{\"who\":\"AWS SDK for Java\"}";

        AWSLambdaAsync lambda = AWSLambdaAsyncClientBuilder.defaultClient();
        InvokeRequest req = new InvokeRequest()
            .withFunctionName(function_name)
            .withPayload(ByteBuffer.wrap(function_input.getBytes()));

        Future<InvokeResult> future_res = lambda.invokeAsync(req);

        System.out.print("Waiting for future");
        while (future_res.isDone() == false) {
            System.out.print(".");
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {
                System.err.println("\nThread.sleep() was interrupted!");
                System.exit(1);
            }
        }
        try {
            InvokeResult res = future_res.get();
            if (res.getStatusCode() == 200) {
                System.out.println("\nLambda function returned:");
                ByteBuffer response_payload = res.getPayload();
                System.out.println(new String(response_payload.array()));
            }
            else {
                System.out.format("Received a non-OK response from AWS: %d\n",
                        res.getStatusCode());
            }
        }
        catch (InterruptedException | ExecutionException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }

        System.exit(0);
    }
}
```

Asynchronous Callbacks

In addition to using the Java `Future` object to monitor the status of asynchronous requests, the SDK also enables you to implement a class that uses the AsyncHandler interface. `AsyncHandler` provides two methods that are called depending on how the request completed: `onSuccess` and `onError`.

The major advantage of the callback interface approach is that it frees you from having to poll the `Future` object to find out when the request has completed. Instead, your code can immediately start its next activity, and rely on the SDK to call your handler at the right time.

```
import com.amazonaws.services.lambda.AWSLambdaAsync;
import com.amazonaws.services.lambda.AWSLambdaAsyncClientBuilder;
import com.amazonaws.services.lambda.model.InvokeRequest;
import com.amazonaws.services.lambda.model.InvokeResult;
import com.amazonaws.handlers.AsyncHandler;
import java.nio.ByteBuffer;
import java.util.concurrent.Future;

public class InvokeLambdaFunctionCallback
{
    private class AsyncLambdaHandler implements AsyncHandler<InvokeRequest, InvokeResult>
    {
        public void onSuccess(InvokeRequest req, InvokeResult res) {
            System.out.println("\nLambda function returned:");
            ByteBuffer response_payload = res.getPayload();
            System.out.println(new String(response_payload.array()));
            System.exit(0);
        }

        public void onError(Exception e) {
            System.out.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void main(String[] args)
    {
        String function_name = "HelloFunction";
        String function_input = "{\"who\":\"AWS SDK for Java\"}";

        AWSLambdaAsync lambda = AWSLambdaAsyncClientBuilder.defaultClient();
        InvokeRequest req = new InvokeRequest()
            .withFunctionName(function_name)
            .withPayload(ByteBuffer.wrap(function_input.getBytes()));

        Future<InvokeResult> future_res = lambda.invokeAsync(req, new
 AsyncLambdaHandler());

        System.out.print("Waiting for async callback");
        while (!future_res.isDone() && !future_res.isCancelled()) {
            // perform some other tasks...
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {
                System.err.println("Thread.sleep() was interrupted!");
                System.exit(0);
            }
            System.out.print(".");
        }
    }
}
```

Best Practices

Callback Execution

Your implementation of `AsyncHandler` is executed inside the thread pool owned by the asynchronous client. Short, quickly executed code is most appropriate inside your `AsyncHandler` implementation. Long-running or blocking code inside your handler methods can cause contention for the thread pool used by the asynchronous client, and can prevent the client from executing requests. If you have a long-running task that needs to begin from a callback, have the callback run its task in a new thread or in a thread pool managed by your application.

Thread Pool Configuration

The asynchronous clients in the AWS SDK for Java provide a default thread pool that should work for most applications. You can implement a custom ExecutorService and pass it to AWS SDK for Java asynchronous clients for more control over how the thread pools are managed.

For example, you could provide an `ExecutorService` implementation that uses a custom ThreadFactory to control how threads in the pool are named, or to log additional information about thread usage.

Amazon S3 Asynchronous Access

The TransferManager class in the SDK offers asynchronous support for working with the Amazon S3. `TransferManager` manages asynchronous uploads and downloads, provides detailed progress reporting on transfers, and supports callbacks into different events.

Logging AWS SDK for Java Calls

The AWS SDK for Java is instrumented with Apache Commons Logging, which is an abstraction layer that enables the use of any one of several logging systems at runtime.

Supported logging systems include the Java Logging Framework and Apache Log4j, among others. This topic shows you how to use Log4j. You can use the SDK's logging functionality without making any changes to your application code.

To learn more about Log4j, see the Apache website.

Note
This topic focuses on Log4j 1.x. Log4j2 doesn't directly support Apache Commons Logging, but provides an adapter that directs logging calls automatically to Log4j2 using the Apache Commons Logging interface. For more information, see Commons Logging Bridge in the Log4j2 documentation.

Download the Log4J JAR

To use Log4j with the SDK, you need to download the Log4j JAR from the Apache website. The SDK doesn't include the JAR. Copy the JAR file to a location that is on your classpath.

Log4j uses a configuration file, log4j.properties. Example configuration files are shown below. Copy this configuration file to a directory on your classpath. The Log4j JAR and the log4j.properties file don't have to be in the same directory.

The log4j.properties configuration file specifies properties such as logging level, where logging output is sent (for example, to a file or to the console), and the format of the output. The logging level is the granularity of output that the logger generates. Log4j supports the concept of multiple logging *hierarchies*. The logging level is set independently for each hierarchy. The following two logging hierarchies are available in the AWS SDK for Java:

- log4j.logger.com.amazonaws
- log4j.logger.org.apache.http.wire

Setting the Classpath

Both the Log4j JAR and the log4j.properties file must be located on your classpath. If you're using Apache Ant, set the classpath in the `path` element in your Ant file. The following example shows a path element from the Ant file for the Amazon S3 example included with the SDK.

```
<path id="aws.java.sdk.classpath">
  <fileset dir="../../third-party" includes="**/*.jar"/>
  <fileset dir="../../lib" includes="**/*.jar"/>
  <pathelement location="."/>
</path>
```

If you're using the Eclipse IDE, you can set the classpath by opening the menu and navigating to **Project | Properties | Java Build Path**.

Service-Specific Errors and Warnings

We recommend that you always leave the "com.amazonaws" logger hierarchy set to "WARN" to catch any important messages from the client libraries. For example, if the Amazon S3 client detects that your application hasn't properly closed an `InputStream` and could be leaking resources, the S3 client reports it through a warning message to the logs. This also ensures that messages are logged if the client has any problems handling requests or responses.

The following log4j.properties file sets the `rootLogger` to WARN, which causes warning and error messages from all loggers in the "com.amazonaws" hierarchy to be included. Alternatively, you can explicitly set the com.amazonaws logger to WARN.

```
log4j.rootLogger=WARN, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
```

```
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5p %c -  %m%n
# Or you can explicitly enable WARN and ERROR messages for the AWS Java clients
log4j.logger.com.amazonaws=WARN
```

Request/Response Summary Logging

Every request to an AWS service generates a unique AWS request ID that is useful if you run into an issue with how an AWS service is handling a request. AWS request IDs are accessible programmatically through Exception objects in the SDK for any failed service call, and can also be reported through the DEBUG log level in the "com.amazonaws.request" logger.

The following log4j.properties file enables a summary of requests and responses, including AWS request IDs.

```
log4j.rootLogger=WARN, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5p %c -  %m%n
# Turn on DEBUG logging in com.amazonaws.request to log
# a summary of requests/responses with AWS request IDs
log4j.logger.com.amazonaws.request=DEBUG
```

Here is an example of the log output.

```
2009-12-17 09:53:04,269 [main] DEBUG com.amazonaws.request - Sending
Request: POST https://rds.amazonaws.com / Parameters: (MaxRecords: 20,
Action: DescribeEngineDefaultParameters, SignatureMethod: HmacSHA256,
AWSAccessKeyId: ACCESSKEYID, Version: 2009-10-16, SignatureVersion: 2,
Engine: mysql5.1, Timestamp: 2009-12-17T17:53:04.267Z, Signature:
q963XH63Lcovl5Rr71APlzlye99rmWwT9DfuQaNznkD, ) 2009-12-17 09:53:04,464
[main] DEBUG com.amazonaws.request - Received successful response: 200, AWS
Request ID: 694d1242-cee0-c85e-f31f-5dab1ea18bc6 2009-12-17 09:53:04,469
[main] DEBUG com.amazonaws.request - Sending Request: POST
https://rds.amazonaws.com / Parameters: (ResetAllParameters: true, Action:
ResetDBParameterGroup, SignatureMethod: HmacSHA256, DBParameterGroupName:
java-integ-test-param-group-0000000000000, AWSAccessKeyId: ACCESSKEYID,
Version: 2009-10-16, SignatureVersion: 2, Timestamp:
2009-12-17T17:53:04.467Z, Signature:
9WcgfPwTobvLVcpyhbrdN7P7l3uH0oviYQ4yZ+TQjsQ=, )

2009-12-17 09:53:04,646 [main] DEBUG com.amazonaws.request - Received
successful response: 200, AWS Request ID:
694d1242-cee0-c85e-f31f-5dab1ea18bc6
```

Verbose Wire Logging

In some cases, it can be useful to see the exact requests and responses that the AWS SDK for Java sends and receives. You shouldn't enable this logging in production systems because writing out large requests (e.g., a file being uploaded to Amazon S3) or responses can significantly slow down an application. If you really need access to this information, you can temporarily enable it through the Apache HttpClient 4 logger. Enabling the DEBUG level on the `apache.http.wire` logger enables logging for all request and response data.

The following log4j.properties file turns on full wire logging in Apache HttpClient 4 and should only be turned on temporarily because it can have a significant performance impact on your application.

```
log4j.rootLogger=WARN, A1
```

```
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5p %c -  %m%n
# Log all HTTP content (headers, parameters, content, etc)  for
# all requests and responses. Use caution with this since it can
# be very expensive to log such verbose data!
log4j.logger.org.apache.http.wire=DEBUG
```

Client Networking Configuration

The AWS SDK for Java enables you to change the default client configuration, which is helpful when you want to:

- Connect to the Internet through proxy
- Change HTTP transport settings, such as connection timeout and request retries
- Specify TCP socket buffer size hints

Proxy Configuration

When constructing a client object, you can pass in an optional ClientConfiguration object to customize the client's configuration.

If you connect to the Internet through a proxy server, you'll need to configure your proxy server settings (proxy host, port, and username/password) through the `ClientConfiguration` object.

HTTP Transport Configuration

You can configure several HTTP transport options by using the ClientConfiguration object. New options are occasionally added; to see the full list of options you can retrieve or set, see the *AWS SDK for Java API Reference*.

Each of the configurable values has a default value defined by a constant. For a list of the constant values for `ClientConfiguration`, see Constant Field Values in the *AWS SDK for Java API Reference*.

Local Address

To set the local address that the HTTP client will bind to, use ClientConfiguration.setLocalAddress.

Maximum Connections

You can set the maximum allowed number of open HTTP connections by using the ClientConfiguration.setMaxConnections method.

Proxy Options

If you use a proxy with your HTTP connections, you might need to set certain options related to HTTP proxies.

Timeouts and Error Handling

You can set options related to timeouts and handling errors with HTTP connections.

- **Connection Timeout**

  The connection timeout is the amount of time (in milliseconds) that the HTTP connection will wait to establish a connection before giving up. The default is 50,000 ms.

  To set this value yourself, use the ClientConfiguration.setConnectionTimeout method.
- **Connection Time to Live (TTL)**

  By default, the SDK will attempt to reuse HTTP connections as long as possible. In failure situations where a connection is established to a server that has been brought out of service, having a finite TTL can help with application recovery. For example, setting a 15 minute TTL will ensure that even if you have a connection established to a server that is experiencing issues, you'll reestablish a connection to a new server within 15 minutes.

  To set the HTTP connection TTL, use the ClientConfiguration.setConnectionTTL method.
- **Maximum Error Retries**

  The default maximum retry count for retriable errors is 3. You can set a different value by using the ClientConfiguration.setMaxErrorRetry method.

TCP Socket Buffer Size Hints

Advanced users who want to tune low-level TCP parameters can additionally set TCP buffer size hints through the ClientConfiguration object. The majority of users will never need to tweak these values, but they are provided for advanced users.

Optimal TCP buffer sizes for an application are highly dependent on network and operating system configuration and capabilities. For example, most modern operating systems provide auto-tuning logic for TCP buffer sizes.This can have a big impact on performance for TCP connections that are held open long enough for the auto-tuning to optimize buffer sizes.

Large buffer sizes (e.g., 2 MB) allow the operating system to buffer more data in memory without requiring the remote server to acknowledge receipt of that information, and so can be particularly useful when the network has high latency.

This is only a *hint*, and the operating system might not to honor it. When using this option, users should always check the operating system's configured limits and defaults. Most operating systems have a maximum TCP buffer size limit configured, and won't let you go beyond that limit unless you explicitly raise the maximum TCP buffer size limit.

Many resources are available to help with configuring TCP buffer sizes and operating system-specific TCP settings, including the following:

- TCP Tuning and Network Troubleshooting
- Host Tuning

Access Control Policies

AWS *access control policies* enable you to specify fine-grained access controls on your AWS resources. An access control policy consists of a collection of *statements*, which take the form:

Account A has permission to perform *action B* on *resource C* where *condition D* applies.

Where:

- *A* is the *principal*– The AWS account that is making a request to access or modify one of your AWS resources.
- *B* is the *action*– The way in which your AWS resource is being accessed or modified, such as sending a message to an Amazon SQS queue, or storing an object in an Amazon S3 bucket.
- *C* is the *resource*– The AWS entity that the principal wants to access, such as an Amazon SQS queue, or an object stored in Amazon S3.
- *D* is a *set of conditions*– The optional constraints that specify when to allow or deny access for the principal to access your resource. Many expressive conditions are available, some specific to each service. For example, you can use date conditions to allow access to your resources only after or before a specific time.

Amazon S3 Example

The following example demonstrates a policy that allows anyone access to read all the objects in a bucket, but restricts access to uploading objects to that bucket to two specific AWS accounts (in addition to the bucket owner's account).

```
Statement allowPublicReadStatement = new Statement(Effect.Allow)
    .withPrincipals(Principal.AllUsers)
    .withActions(S3Actions.GetObject)
    .withResources(new S3ObjectResource(myBucketName, "*"));
Statement allowRestrictedWriteStatement = new Statement(Effect.Allow)
    .withPrincipals(new Principal("123456789"), new Principal("876543210"))
    .withActions(S3Actions.PutObject)
    .withResources(new S3ObjectResource(myBucketName, "*"));

Policy policy = new Policy()
    .withStatements(allowPublicReadStatement, allowRestrictedWriteStatement);

AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
s3.setBucketPolicy(myBucketName, policy.toJson());
```

Amazon SQS Example

One common use of policies is to authorize an Amazon SQS queue to receive messages from an Amazon SNS topic.

```
Policy policy = new Policy().withStatements(
    new Statement(Effect.Allow)
        .withPrincipals(Principal.AllUsers)
        .withActions(SQSActions.SendMessage)
        .withConditions(ConditionFactory.newSourceArnCondition(myTopicArn)));

Map queueAttributes = new HashMap();
queueAttributes.put(QueueAttributeName.Policy.toString(), policy.toJson());

AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
sqs.setQueueAttributes(new SetQueueAttributesRequest(myQueueUrl, queueAttributes));
```

Amazon SNS Example

Some services offer additional conditions that can be used in policies. Amazon SNS provides conditions for allowing or denying subscriptions to SNS topics based on the protocol (e.g., email, HTTP, HTTPS, Amazon SQS) and endpoint (e.g., email address, URL, Amazon SQS ARN) of the request to subscribe to a topic.

```
Condition endpointCondition =
    SNSConditionFactory.newEndpointCondition("*@mycompany.com");

Policy policy = new Policy().withStatements(
    new Statement(Effect.Allow)
        .withPrincipals(Principal.AllUsers)
        .withActions(SNSActions.Subscribe)
        .withConditions(endpointCondition));

AmazonSNS sns = AmazonSNSClientBuilder.defaultClient();
sns.setTopicAttributes(
    new SetTopicAttributesRequest(myTopicArn, "Policy", policy.toJson()));
```

Setting the JVM TTL for DNS Name Lookups

The Java virtual machine (JVM) caches DNS name lookups. When the JVM resolves a hostname to an IP address, it caches the IP address for a specified period of time, known as the *time-to-live* (TTL).

Because AWS resources use DNS name entries that occasionally change, we recommend that you configure your JVM with a TTL value of no more than 60 seconds. This ensures that when a resource's IP address changes, your application will be able to receive and use the resource's new IP address by requerying the DNS.

On some Java configurations, the JVM default TTL is set so that it will *never* refresh DNS entries until the JVM is restarted. Thus, if the IP address for an AWS resource changes while your application is still running, it won't be able to use that resource until you *manually restart* the JVM and the cached IP information is refreshed. In this case, it's crucial to set the JVM's TTL so that it will periodically refresh its cached IP information.

Note
The default TTL can vary according to the version of your JVM and whether a security manager is installed. Many JVMs provide a default TTL less than 60 seconds. If you're using such a JVM and not using a security manager, you can ignore the remainder of this topic.

How to Set the JVM TTL

To modify the JVM's TTL, set the networkaddress.cache.ttl property value. Use one of the following methods, depending on your needs:

- **globally, for all applications that use the JVM**. Set `networkaddress.cache.ttl` in the `$JAVA_HOME/jre/lib/security/java.security` file:

```
networkaddress.cache.ttl=60
```

- **for your application only**, set `networkaddress.cache.ttl` in your application's initialization code:

```
java.security.Security.setProperty("networkaddress.cache.ttl" , "60");
```

Enabling Metrics for the AWS SDK for Java

The AWS SDK for Java can generate metrics for visualization and monitoring with CloudWatch that measure:

- your application's performance when accessing AWS
- the performance of your JVMs when used with AWS
- runtime environment details such as heap memory, number of threads, and opened file descriptors

How to Enable SDK Metric Generation

SDK metrics are *disabled by default*. To enable it for your local development environment, include a system property that points to your AWS security credential file when starting up the JVM. For example:

```
-Dcom.amazonaws.sdk.enableDefaultMetrics=credentialFile=/path/aws.properties
```

You need to specify the path to your credential file so that the SDK can upload the gathered datapoints to CloudWatch for later analysis.

Note
If you are accessing AWS from an Amazon EC2 instance using the Amazon EC2 instance metadata service, you don't need to specify a credential file. In this case, you need only specify:

```
-Dcom.amazonaws.sdk.enableDefaultMetrics
```

All metrics captured by the SDK for Java are under the namespace **AWSSDK/Java**, and are uploaded to the CloudWatch default region (*us-east-1*). To change the region, specify it by using the `cloudwatchRegion` attribute in the system property. For example, to set the CloudWatch region to *us-west-2*, use:

```
-Dcom.amazonaws.sdk.enableDefaultMetrics=credentialFile=/path/
aws.properties,cloudwatchRegion=us-west-2
```

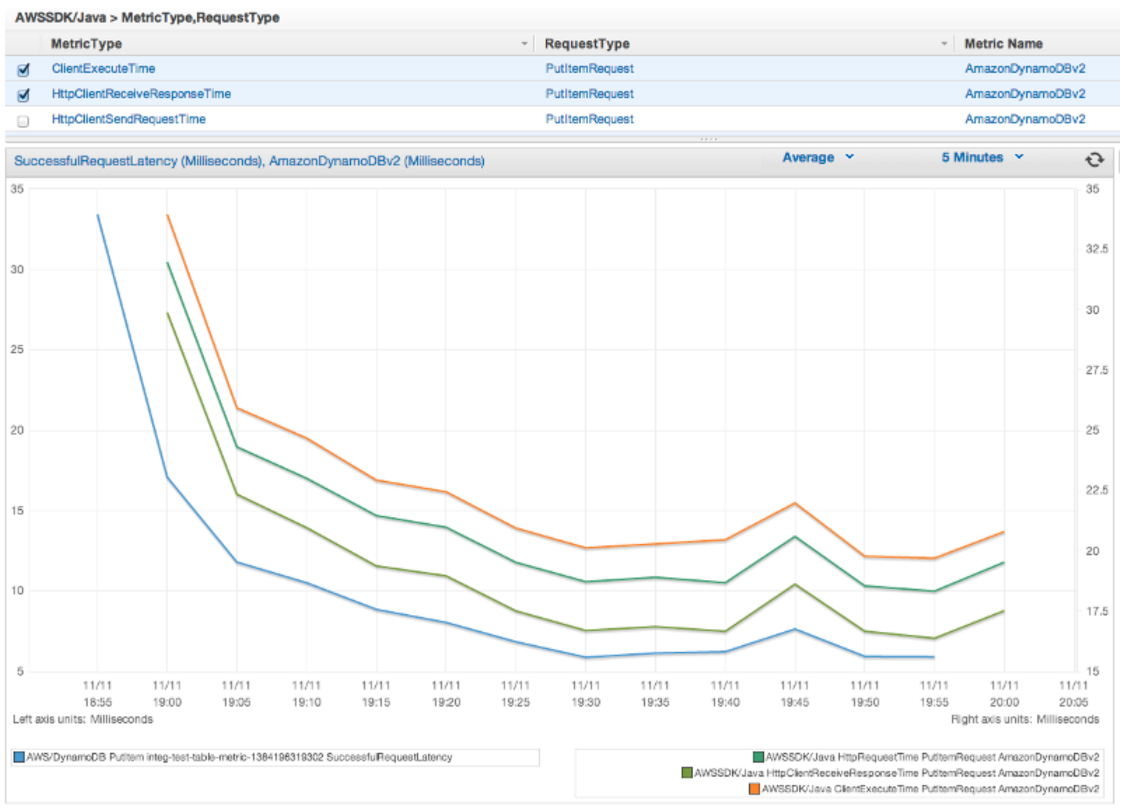
Once you enable the feature, every time there is a service request to AWS from the AWS SDK for Java, metric data points will be generated, queued for statistical summary, and uploaded asynchronously to CloudWatch about once every minute. Once metrics have been uploaded, you can visualize them using the AWS Management Console and set alarms on potential problems such as memory leakage, file descriptor leakage, and so on.

Available Metric Types

The default set of metrics is divided into three major categories:

AWS Request Metrics

Covers areas such as the latency of the HTTP request/response, number of requests, exceptions, and retries.



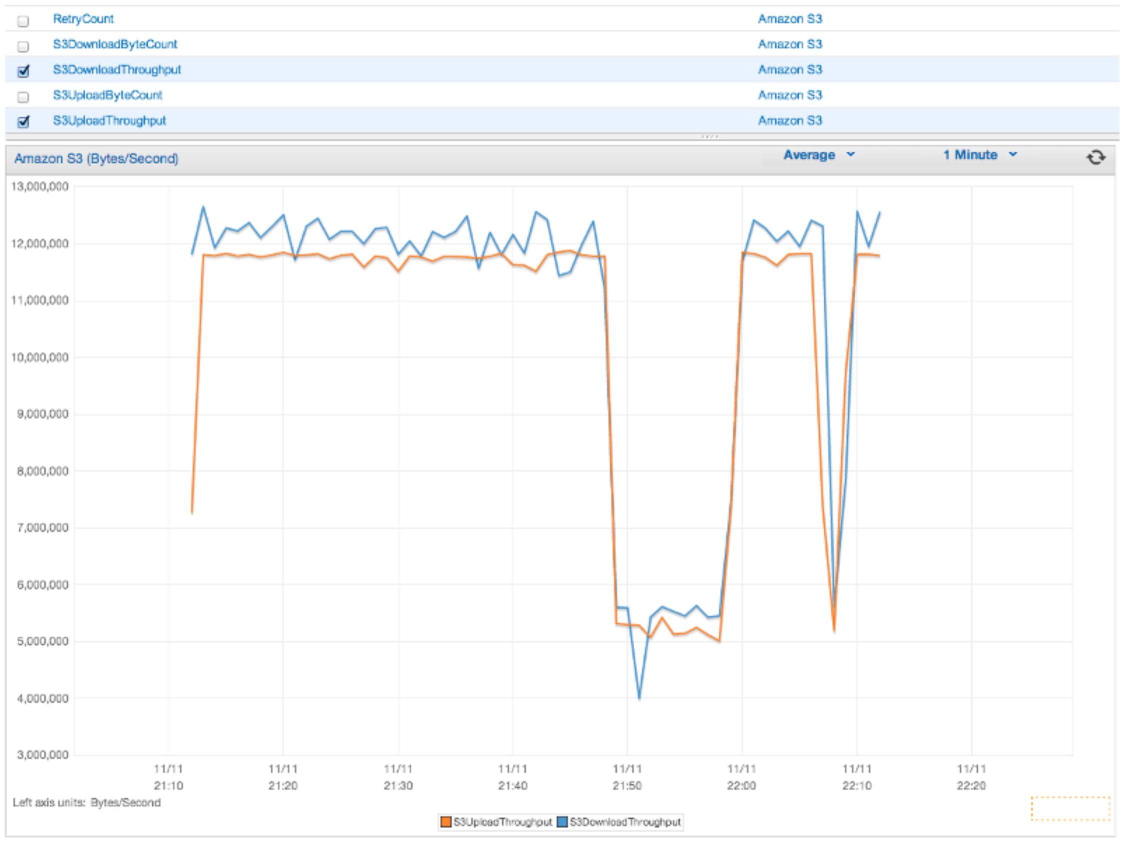
AWSSDK/Java > MetricType,RequestType

	MetricType	RequestType	Metric Name
✓	ClientExecuteTime	PutItemRequest	AmazonDynamoDBv2
✓	HttpClientReceiveResponseTime	PutItemRequest	AmazonDynamoDBv2
☐	HttpClientSendRequestTime	PutItemRequest	AmazonDynamoDBv2

SuccessfulRequestLatency (Milliseconds), AmazonDynamoDBv2 (Milliseconds)
Average
5 Minutes
35
30
25
20
15
10
5
35
32.5
30
27.5
25
22.5
20
17.5
15
11/11 18:55
11/11 19:00
11/11 19:05
11/11 19:10
11/11 19:15
11/11 19:20
11/11 19:25
11/11 19:30
11/11 19:35
11/11 19:40
11/11 19:45
11/11 19:50
11/11 19:55
11/11 20:00
11/11 20:05
Left axis units: Milliseconds
Right axis units: Milliseconds
AWS/DynamoDB PutItem integ-test-table-metric-1384196319302 SuccessfulRequestLatency
AWSSDK/Java HttpRequestTime PutItemRequest AmazonDynamoDBv2
AWSSDK/Java HttpClientReceiveResponseTime PutItemRequest AmazonDynamoDBv2
AWSSDK/Java ClientExecuteTime PutItemRequest AmazonDynamoDBv2

AWS Service Metrics

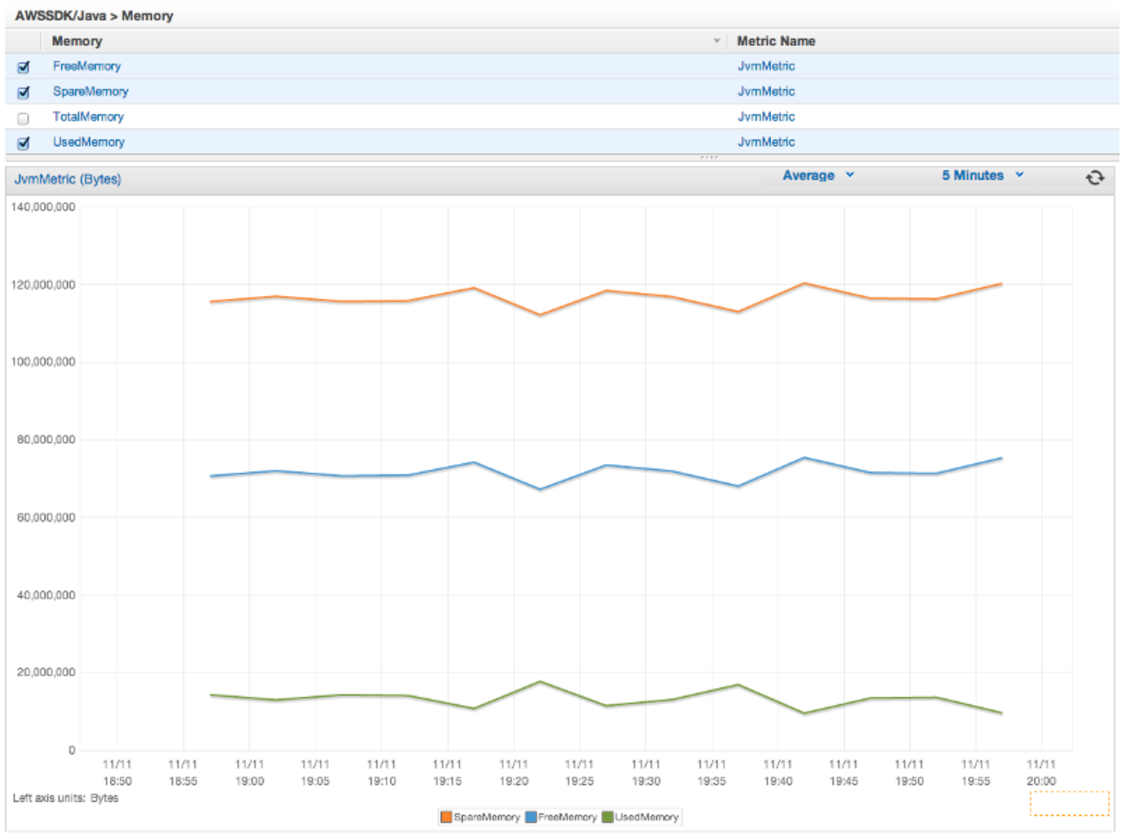
Include AWS service-specific data, such as the throughput and byte count for S3 uploads and downloads.



☐	RetryCount	Amazon S3
☐	S3DownloadByteCount	Amazon S3
☑	S3DownloadThroughput	Amazon S3
☐	S3UploadByteCount	Amazon S3
☑	S3UploadThroughput	Amazon S3

Machine Metrics

Cover the runtime environment, including heap memory, number of threads, and open file descriptors.


AWSSDK/Java > Memory

	Memory	Metric Name
✓	FreeMemory	JvmMetric
✓	SpareMemory	JvmMetric
☐	TotalMemory	JvmMetric
✓	UsedMemory	JvmMetric

If you want to exclude Machine Metrics, add `excludeMachineMetrics` to the system property:

```
-Dcom.amazonaws.sdk.enableDefaultMetrics=credentialFile=/path/
aws.properties,excludeMachineMetrics
```

More Information

- See the amazonaws/metrics package summary for a full list of the predefined core metric types.
- Learn about working with CloudWatch using the AWS SDK for Java in CloudWatch Examples (p. 36).

Programming Examples

This section provides tutorials and examples of using the AWS SDK for Java to program AWS services.

Note
See Additional Documentation and Resources (p. 1) for more examples and additional resources available for AWS SDK for Java developers!

Topics

SDK Code Samples

The AWS SDK for Java comes packaged with code samples that demonstrate many of the features of the SDK in buildable, runnable programs. You can study or modify these to implement your own AWS solutions using the AWS SDK for Java.

How to Get the Samples

The AWS SDK for Java code samples are provided in the *samples* directory of the SDK. If you downloaded and installed the SDK using the information in Set up the AWS SDK for Java (p. 4), you already have the samples on your system.

You can also view the latest samples on the AWS SDK for Java GitHub repository, in the src/samples directory.

Building and Running the Samples Using the Command Line

The samples include Ant build scripts so that you can easily build and run them from the command line. Each sample also contains a README file in HTML format that contains information specific to each sample.

Note
If you're browsing the sample code on GitHub, click the **Raw** button in the source code display when viewing the sample's README.html file. In raw mode, the HTML will render as intended in your browser.

Prerequisites

Before running any of the AWS SDK for Java samples, you need to set your AWS credentials in the environment or with the AWS CLI, as specified in Set up AWS Credentials and Region for Development (p. 6). The samples use the default credential provider chain whenever possible. So by setting your credentials in this way, you can avoid the risky practice of inserting your AWS credentials in files within the source code directory (where they may inadvertently be checked in and shared publicly).

Running the Samples

To run a sample from the command line

1. Change to the directory containing the sample's code. For example, if you're in the root directory of the AWS SDK download and want to run the `AwsConsoleApp` sample, you would type:

   ```
   cd samples/AwsConsoleApp
   ```

2. Build and run the sample with Ant. The default build target performs both actions, so you can just enter:

   ```
   ant
   ```

The sample prints information to standard output—for example:

```
===========================================

Welcome to the AWS Java SDK!

===========================================
You have access to 4 Availability Zones.

You have 0 Amazon EC2 instance(s) running.

You have 13 Amazon SimpleDB domain(s) containing a total of 62 items.

You have 23 Amazon S3 bucket(s), containing 44 objects with a total size of 154767691
 bytes.
```

Building and Running the Samples Using the Eclipse IDE

If you use the AWS Toolkit for Eclipse, you can also start a new project in Eclipse based on the AWS SDK for Java or add the SDK to an existing Java project.

Prerequisites

After installing the AWS Toolkit for Eclipse, we recommend configuring the Toolkit with your security credentials. You can do this anytime by choosing **Preferences** from the **Window** menu in Eclipse, and then choosing the **AWS Toolkit** section.

Running the Samples

To run a sample using the AWS Toolkit for Eclipse

1. Open Eclipse.
2. Create a new AWS Java project. In Eclipse, on the **File** menu, choose **New**, and then click **Project**. The **New Project** wizard opens.
3. Expand the **AWS** category, then choose **AWS Java Project**.
4. Choose **Next**. The project settings page is displayed.
5. Enter a name in the **Project Name** box. The AWS SDK for Java Samples group displays the samples available in the SDK, as described previously.
6. Select the samples you want to include in your project by selecting each check box.
7. Enter your AWS credentials. If you've already configured the AWS Toolkit for Eclipse with your credentials, this is automatically filled in.
8. Choose **Finish**. The project is created and added to the **Project Explorer**.

To run the project

1. Choose the sample `.java` file you want to run. For example, for the Amazon S3 sample, choose `S3Sample.java`.
2. Choose **Run** from the **Run** menu.

To add the SDK to an existing project

1. Right-click the project in **Project Explorer**, point to **Build Path**, and then choose **Add Libraries**.
2. Choose **AWS Java SDK**, choose **Next**, and then follow the remaining on-screen instructions.

CloudWatch Examples

This section provides examples of programming CloudWatch using the AWS SDK for Java.

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

For more information about CloudWatch, see the Amazon CloudWatch User Guide.

Note
The examples include only the code needed to demonstrate each technique. The complete example code is available on GitHub. From there, you can download a single source file or clone the repository locally to get all the examples to build and run.

Topics

Getting Metrics from CloudWatch

Listing Metrics

To list CloudWatch metrics, create a ListMetricsRequest and call the AmazonCloudWatchClient's `listMetrics` method. You can use the `ListMetricsRequest` to filter the returned metrics by namespace, metric name, or dimensions.

Note
A list of metrics and dimensions that are posted by AWS services can be found within the Amazon CloudWatch Metrics and Dimensions Reference in the *Amazon CloudWatch User Guide*.

Imports

```
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.ListMetricsRequest;
import com.amazonaws.services.cloudwatch.model.ListMetricsResult;
import com.amazonaws.services.cloudwatch.model.Metric;
```

Code

```
final AmazonCloudWatch cw =
    AmazonCloudWatchClientBuilder.defaultClient();

boolean done = false;

while(!done) {
    ListMetricsRequest request = new ListMetricsRequest()
        .withMetricName(name)
        .withNamespace(namespace);

    ListMetricsResult response = cw.listMetrics(request);

    for(Metric metric : response.getMetrics()) {
        System.out.printf(
            "Retrieved metric %s", metric.getMetricName());
    }

    request.setNextToken(response.getNextToken());

    if(response.getNextToken() == null) {
        done = true;
    }
}
```

The metrics are returned in a ListMetricsResult by calling its `getMetrics` method. The results may be *paged*. To retrieve the next batch of results, call `setNextToken` on the original request object with the return value of the `ListMetricsResult` object's `getNextToken` method, and pass the modified request object back to another call to `listMetrics`.

More Information

- ListMetrics in the *Amazon CloudWatch API Reference*.

Publishing Custom Metric Data

A number of AWS services publish their own metrics in namespaces beginning with "AWS/" You can also publish custom metric data using your own namespace (as long as it doesn't begin with "AWS/").

Publish Custom Metric Data

To publish your own metric data, call the AmazonCloudWatchClient's `putMetricData` method with a PutMetricDataRequest. The `PutMetricDataRequest` must include the custom namespace to use for the data, and information about the data point itself in a MetricDatum object.

> **Note**
> You cannot specify a namespace that begins with "AWS/". Namespaces that begin with "AWS/" are reserved for use by Amazon Web Services products. .

Imports

```
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.Dimension;
import com.amazonaws.services.cloudwatch.model.MetricDatum;
import com.amazonaws.services.cloudwatch.model.PutMetricDataRequest;
import com.amazonaws.services.cloudwatch.model.PutMetricDataResult;
import com.amazonaws.services.cloudwatch.model.StandardUnit;
```

Code

```
final AmazonCloudWatch cw =
    AmazonCloudWatchClientBuilder.defaultClient();

Dimension dimension = new Dimension()
    .withName("UNIQUE_PAGES")
    .withValue("URLS");

MetricDatum datum = new MetricDatum()
    .withMetricName("PAGES_VISITED")
    .withUnit(StandardUnit.None)
    .withValue(data_point)
    .withDimensions(dimension);

PutMetricDataRequest request = new PutMetricDataRequest()
    .withNamespace("SITE/TRAFFIC")
    .withMetricData(datum);

PutMetricDataResult response = cw.putMetricData(request);
```

More Information

- Using Amazon CloudWatch Metrics in the *Amazon CloudWatch User Guide*.

- AWS Namespaces in the *Amazon CloudWatch User Guide*.
- PutMetricData in the *Amazon CloudWatch API Reference*.

Working with CloudWatch Alarms

Create an Alarm

To create an alarm based on a CloudWatch metric, call the AmazonCloudWatchClient's `putMetricAlarm` method with a PutMetricAlarmRequest filled with the alarm conditions.

Imports

```
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.ComparisonOperator;
import com.amazonaws.services.cloudwatch.model.Dimension;
import com.amazonaws.services.cloudwatch.model.PutMetricAlarmRequest;
import com.amazonaws.services.cloudwatch.model.PutMetricAlarmResult;
import com.amazonaws.services.cloudwatch.model.StandardUnit;
import com.amazonaws.services.cloudwatch.model.Statistic;
```

Code

```
final AmazonCloudWatch cw =
    AmazonCloudWatchClientBuilder.defaultClient();

Dimension dimension = new Dimension()
    .withName("InstanceId")
    .withValue(instanceId);

PutMetricAlarmRequest request = new PutMetricAlarmRequest()
    .withAlarmName(alarmName)
    .withComparisonOperator(
        ComparisonOperator.GreaterThanThreshold)
    .withEvaluationPeriods(1)
    .withMetricName("CPUUtilization")
    .withNamespace("AWS/EC2")
    .withPeriod(60)
    .withStatistic(Statistic.Average)
    .withThreshold(70.0)
    .withActionsEnabled(false)
    .withAlarmDescription(
        "Alarm when server CPU utilization exceeds 70%")
    .withUnit(StandardUnit.Seconds)
    .withDimensions(dimension);

PutMetricAlarmResult response = cw.putMetricAlarm(request);
```

List Alarms

To list the CloudWatch alarms that you have created, call the AmazonCloudWatchClient's `describeAlarms` method with a DescribeAlarmsRequest that you can use to set options for the result.

Imports

```
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.DescribeAlarmsRequest;
import com.amazonaws.services.cloudwatch.model.DescribeAlarmsResult;
import com.amazonaws.services.cloudwatch.model.MetricAlarm;
```

Code

```
final AmazonCloudWatch cw =
    AmazonCloudWatchClientBuilder.defaultClient();

boolean done = false;

while(!done) {
    DescribeAlarmsRequest request = new DescribeAlarmsRequest();

    DescribeAlarmsResult response = cw.describeAlarms(request);

    for(MetricAlarm alarm : response.getMetricAlarms()) {
        System.out.printf("Retrieved alarm %s", alarm.getAlarmName());
    }

    request.setNextToken(response.getNextToken());

    if(response.getNextToken() == null) {
        done = true;
    }
}
```

The list of alarms can be obtained by calling `getMetricAlarms` on the DescribeAlarmsResult that is returned by `describeAlarms`.

The results may be *paged*. To retrieve the next batch of results, call `setNextToken` on the original request object with the return value of the `DescribeAlarmsResult` object's `getNextToken` method, and pass the modified request object back to another call to `describeAlarms`.

> **Note**
> You can also retrieve alarms for a specific metric by using the AmazonCloudWatchClient's `describeAlarmsForMetric` method. Its use is similar to `describeAlarms`.

Delete Alarms

To delete CloudWatch alarms, call the AmazonCloudWatchClient's `deleteAlarms` method with a DeleteAlarmsRequest containing one or more names of alarms that you want to delete.

Imports

```
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.DeleteAlarmsRequest;
import com.amazonaws.services.cloudwatch.model.DeleteAlarmsResult;
```

Code

```
final AmazonCloudWatch cw =
    AmazonCloudWatchClientBuilder.defaultClient();

DeleteAlarmsRequest request = new DeleteAlarmsRequest()
    .withAlarmNames(alarm_name);

DeleteAlarmsResult response = cw.deleteAlarms(request);
```

More Information

- Creating Amazon CloudWatch Alarms in the *Amazon CloudWatch User Guide*
- PutMetricAlarm in the *Amazon CloudWatch API Reference*
- DescribeAlarms in the *Amazon CloudWatch API Reference*
- DeleteAlarms in the *Amazon CloudWatch API Reference*

Using Alarm Actions in CloudWatch

Using CloudWatch alarm actions, you can create alarms that perform actions such as automatically stopping, terminating, rebooting, or recovering Amazon EC2 instances.

> **Note**
> Alarm actions can be added to an alarm by using the PutMetricAlarmRequest's `setAlarmActions` method when creating an alarm (p. 39).

Enable Alarm Actions

To enable alarm actions for a CloudWatch alarm, call the AmazonCloudWatchClient's `enableAlarmActions` with a EnableAlarmActionsRequest containing one or more names of alarms whose actions you want to enable.

Imports

```
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.EnableAlarmActionsRequest;
import com.amazonaws.services.cloudwatch.model.EnableAlarmActionsResult;
```

Code

```
final AmazonCloudWatch cw =
    AmazonCloudWatchClientBuilder.defaultClient();

EnableAlarmActionsRequest request = new EnableAlarmActionsRequest()
    .withAlarmNames(alarm);

EnableAlarmActionsResult response = cw.enableAlarmActions(request);
```

Disable Alarm Actions

To disable alarm actions for a CloudWatch alarm, call the AmazonCloudWatchClient's `disableAlarmActions` with a DisableAlarmActionsRequest containing one or more names of alarms whose actions you want to disable.

Imports

```
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.DisableAlarmActionsRequest;
import com.amazonaws.services.cloudwatch.model.DisableAlarmActionsResult;
```

Code

```
final AmazonCloudWatch cw =
    AmazonCloudWatchClientBuilder.defaultClient();

DisableAlarmActionsRequest request = new DisableAlarmActionsRequest()
    .withAlarmNames(alarmName);

DisableAlarmActionsResult response = cw.disableAlarmActions(request);
```

More Information

- Create Alarms to Stop, Terminate, Reboot, or Recover an Instance in the *Amazon CloudWatch User Guide*
- PutMetricAlarm in the *Amazon CloudWatch API Reference*
- EnableAlarmActions in the *Amazon CloudWatch API Reference*
- DisableAlarmActions in the *Amazon CloudWatch API Reference*

Sending Events to CloudWatch

CloudWatch Events delivers a near real-time stream of system events that describe changes in AWS resources to Amazon EC2 instances, Lambda functions, Kinesis streams, Amazon ECS tasks, Step Functions state machines, Amazon SNS topics, Amazon SQS queues, or built-in targets. You can match events and route them to one or more target functions or streams by using simple rules.

Add Events

To add custom CloudWatch events, call the AmazonCloudWatchEventsClient's `putEvents` method with a PutEventsRequest object that contains one or more PutEventsRequestEntry objects that provide details about each event. You can specify several parameters for the entry such as the source and type of the event, resources associated with the event, and so on.

Note
You can specify a maximum of 10 events per call to `putEvents`.

Imports

```
import com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEvents;
import com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEventsClientBuilder;
import com.amazonaws.services.cloudwatchevents.model.PutEventsRequest;
import com.amazonaws.services.cloudwatchevents.model.PutEventsRequestEntry;
import com.amazonaws.services.cloudwatchevents.model.PutEventsResult;
```

Code

```
final AmazonCloudWatchEvents cwe =
    AmazonCloudWatchEventsClientBuilder.defaultClient();

final String EVENT_DETAILS =
    "{ \"key1\": \"value1\", \"key2\": \"value2\" }";

PutEventsRequestEntry request_entry = new PutEventsRequestEntry()
    .withDetail(EVENT_DETAILS)
    .withDetailType("sampleSubmitted")
    .withResources(resource_arn)
    .withSource("aws-sdk-java-cloudwatch-example");
```

```
PutEventsRequest request = new PutEventsRequest()
    .withEntries(request_entry);

PutEventsResult response = cwe.putEvents(request);
```

Add Rules

To create or update a rule, call the AmazonCloudWatchEventsClient's `putRule` method with a PutRuleRequest with the name of the rule and optional parameters such as the event pattern, IAM role to associate with the rule, and a scheduling expression that describes how often the rule is run.

Imports

```
import com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEvents;
import com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEventsClientBuilder;
import com.amazonaws.services.cloudwatchevents.model.PutRuleRequest;
import com.amazonaws.services.cloudwatchevents.model.PutRuleResult;
import com.amazonaws.services.cloudwatchevents.model.RuleState;
```

Code

```
final AmazonCloudWatchEvents cwe =
    AmazonCloudWatchEventsClientBuilder.defaultClient();

PutRuleRequest request = new PutRuleRequest()
    .withName(rule_name)
    .withRoleArn(role_arn)
    .withScheduleExpression("rate(5 minutes)")
    .withState(RuleState.ENABLED);

PutRuleResult response = cwe.putRule(request);
```

Add Targets

Targets are the resources that are invoked when a rule is triggered. Example targets include Amazon EC2 instances, Lambda functions, Kinesis streams, Amazon ECS tasks, Step Functions state machines, and built-in targets.

To add a target to a rule, call the AmazonCloudWatchEventsClient's `putTargets` method with a PutTargetsRequest containing the rule to update and a list of targets to add to the rule.

Imports

```
import com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEvents;
import com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEventsClientBuilder;
import com.amazonaws.services.cloudwatchevents.model.PutTargetsRequest;
import com.amazonaws.services.cloudwatchevents.model.PutTargetsResult;
import com.amazonaws.services.cloudwatchevents.model.Target;
```

Code

```
final AmazonCloudWatchEvents cwe =
    AmazonCloudWatchEventsClientBuilder.defaultClient();

Target target = new Target()
    .withArn(function_arn)
    .withId(target_id);
```

```
PutTargetsRequest request = new PutTargetsRequest()
    .withTargets(target)
    .withRule(rule_name);

PutTargetsResult response = cwe.putTargets(request);
```

More Information

- Adding Events with PutEvents in the *Amazon CloudWatch Events User Guide*
- Schedule Expressions for Rules in the *Amazon CloudWatch Events User Guide*
- Event Types for CloudWatch Events in the *Amazon CloudWatch Events User Guide*
- Events and Event Patterns in the *Amazon CloudWatch Events User Guide*
- PutEvents in the *Amazon CloudWatch Events API Reference*
- PutTargets in the *Amazon CloudWatch Events API Reference*
- PutRule in the *Amazon CloudWatch Events API Reference*

DynamoDB Examples

This section provides examples of programming DynamoDB using the AWS SDK for Java.

Note
The examples include only the code needed to demonstrate each technique. The complete example code is available on GitHub. From there, you can download a single source file or clone the repository locally to get all the examples to build and run.

Topics

Working with Tables in DynamoDB

Tables are the containers for all items in a DynamoDB database. Before you can add or remove data from DynamoDB, you must create a table.

For each table, you must define:

- A table *name* that is unique for your account and region.
- A *primary key* for which every value must be unique; no two items in your table can have the same primary key value.

  A primary key can be *simple*, consisting of a single partition (HASH) key, or *composite*, consisting of a partition and a sort (RANGE) key.

  Each key value has an associated *data type*, enumerated by the ScalarAttributeType class. The key value can be binary (B), numeric (N), or a string (S). For more information, see Naming Rules and Data Types in the *Amazon DynamoDB Developer Guide*.
- *Provisioned throughput* values that define the number of reserved read/write capacity units for the table.

Note
Amazon DynamoDB pricing is based on the provisioned throughput values that you set on your tables, so reserve only as much capacity as you think you'll need for your table. Provisioned throughput for a table can be modified at any time, so you can adjust capacity if your needs change.

Create a Table

Use the DynamoDB client's `createTable` method to create a new DynamoDB table. You need to construct table attributes and a table schema, both of which are used to identify the primary key of your table. You must also supply initial provisioned throughput values and a table name.

Note
If a table with the name you chose already exists, an AmazonServiceException is thrown.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.CreateTableResult;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
```

Create a Table with a Simple Primary Key

This code creates a table with a simple primary key ("Name").

Code

```

CreateTableRequest request = new CreateTableRequest()
    .withAttributeDefinitions(new AttributeDefinition(
             "Name", ScalarAttributeType.S))
    .withKeySchema(new KeySchemaElement("Name", KeyType.HASH))
    .withProvisionedThroughput(new ProvisionedThroughput(
             new Long(10), new Long(10)))
    .withTableName(table_name);

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

try {
    CreateTableResult result = ddb.createTable(request);
    System.out.println(result.getTableDescription().getTableName());
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
```

See the complete sample.

Create a Table with a Composite Primary Key

Add another AttributeDefinition and KeySchemaElement to CreateTableRequest.

Code

```
CreateTableRequest request = new CreateTableRequest()
    .withAttributeDefinitions(
          new AttributeDefinition("Language", ScalarAttributeType.S),
          new AttributeDefinition("Greeting", ScalarAttributeType.S))
    .withKeySchema(
          new KeySchemaElement("Language", KeyType.HASH),
          new KeySchemaElement("Greeting", KeyType.RANGE))
    .withProvisionedThroughput(
          new ProvisionedThroughput(new Long(10), new Long(10)))
```

See the complete sample.

List Tables

You can list the tables in a particular region by calling the DynamoDB client's `listTables` method.

> **Note**
> If the named table doesn't exist for your account and region, a ResourceNotFoundException is thrown.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.ListTablesResult;
```

Code

```

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

boolean more_tables = true;
while(more_tables) {
    String last_name = null;
    try {
        ListTablesResult table_list = null;
        if (last_name == null) {
            table_list = ddb.listTables();
        }

        List<String> table_names = table_list.getTableNames();

        if (table_names.size() > 0) {
            for (String cur_name : table_names) {
                System.out.format("* %s\n", cur_name);
            }
        } else {
            System.out.println("No tables found!");
            System.exit(0);
        }

        last_name = table_list.getLastEvaluatedTableName();
        if (last_name == null) {
            more_tables = false;
        }
    } catch (AmazonServiceException e) {
        System.err.println(e.getErrorMessage());
        System.exit(1);
    }
}
System.out.println("\nDone!");
```

By default, up to 100 tables are returned per call—use `getLastEvaluatedTableName` on the returned ListTablesResult object to get the last table that was evaluated. You can use this value to start the listing after the last returned value of the previous listing.

See the complete sample.

Describe (Get Information about) a Table

Call the DynamoDB client's `describeTable` method.

Note
If the named table doesn't exist for your account and region, a ResourceNotFoundException is thrown.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughputDescription;
import com.amazonaws.services.dynamodbv2.model.TableDescription;
```

Code

```

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

try {
    TableDescription table_info =
       ddb.describeTable(table_name).getTable();

    if (table_info != null) {
        System.out.format("Table name  : %s\n",
              table_info.getTableName());
        System.out.format("Table ARN   : %s\n",
              table_info.getTableArn());
        System.out.format("Status      : %s\n",
              table_info.getTableStatus());
        System.out.format("Item count  : %d\n",
              table_info.getItemCount().longValue());
        System.out.format("Size (bytes): %d\n",
              table_info.getTableSizeBytes().longValue());

        ProvisionedThroughputDescription throughput_info =
           table_info.getProvisionedThroughput();
        System.out.println("Throughput");
        System.out.format("  Read Capacity : %d\n",
              throughput_info.getReadCapacityUnits().longValue());
        System.out.format("  Write Capacity: %d\n",
              throughput_info.getWriteCapacityUnits().longValue());

        List<AttributeDefinition> attributes =
           table_info.getAttributeDefinitions();
        System.out.println("Attributes");
        for (AttributeDefinition a : attributes) {
            System.out.format("  %s (%s)\n",
                  a.getAttributeName(), a.getAttributeType());
        }
    }
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
```

See the complete sample.

Modify (Update) a Table

You can modify your table's provisioned throughput values at any time by calling the DynamoDB client's `updateTable` method.

Note
If the named table doesn't exist for your account and region, a ResourceNotFoundException is thrown.

Imports

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
```

Code

```

ProvisionedThroughput table_throughput = new ProvisionedThroughput(
      read_capacity, write_capacity);

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

try {
    ddb.updateTable(table_name, table_throughput);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
```

See the complete sample.

Delete a Table

Call the DynamoDB client's `deleteTable` method and pass it the table's name.

Note
If the named table doesn't exist for your account and region, a ResourceNotFoundException is thrown.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
```

Code

```

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

try {
    ddb.deleteTable(table_name);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
```

See the complete sample.

More Info

- Guidelines for Working with Tables in the *Amazon DynamoDB Developer Guide*
- Working with Tables in DynamoDB in the *Amazon DynamoDB Developer Guide*

Working with Items in DynamoDB

In DynamoDB, an item is a collection of *attributes*, each of which has a *name* and a *value*. An attribute value can be a scalar, set, or document type. For more information, see Naming Rules and Data Types in the *Amazon DynamoDB Developer Guide*.

Retrieve (Get) an Item from a Table

Call the AmazonDynamoDB's `getItem` method and pass it a GetItemRequest object with the table name and primary key value of the item you want. It returns a *GetItemResult <services/dynamodbv2/model/GetItemResult>* object.

You can use the returned `GetItemResult` object's `getItem()` method to retrieve a Map of key (String) and value (AttributeValue) pairs that are associated with the item.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
import java.util.HashMap;
import java.util.Map;
```

Code

```
HashMap<String,AttributeValue> key_to_get =
    new HashMap<String,AttributeValue>();

key_to_get.put("Name", new AttributeValue(name));

GetItemRequest request = null;
if (projection_expression != null) {
    request = new GetItemRequest()
        .withKey(key_to_get)
        .withTableName(table_name)
        .withProjectionExpression(projection_expression);
} else {
    request = new GetItemRequest()
        .withKey(key_to_get)
        .withTableName(table_name);
}

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

try {
    Map<String,AttributeValue> returned_item =
       ddb.getItem(request).getItem();
    if (returned_item != null) {
        Set<String> keys = returned_item.keySet();
        for (String key : keys) {
            System.out.format("%s: %s\n",
                    key, returned_item.get(key).toString());
        }
    } else {
        System.out.format("No item found with the key %s!\n", name);
    }
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
```

See the complete sample.

Add a New Item to a Table

Create a Map of key-value pairs that represent the item's attributes. These must include values for the table's primary key fields. If the item identified by the primary key already exists, its fields are *updated* by the request.

Note
If the named table doesn't exist for your account and region, a ResourceNotFoundException is thrown.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.ResourceNotFoundException;
import java.util.ArrayList;
```

Code

```


HashMap<String,AttributeValue> item_values =
    new HashMap<String,AttributeValue>();

item_values.put("Name", new AttributeValue(name));

for (String[] field : extra_fields) {
    item_values.put(field[0], new AttributeValue(field[1]));
}

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

try {
    ddb.putItem(table_name, item_values);
} catch (ResourceNotFoundException e) {
    System.err.format("Error: The table \"%s\" can't be found.\n", table_name);
    System.err.println("Be sure that it exists and that you've typed its name correctly!");
    System.exit(1);
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
```

See the complete sample.

Update an Existing Item in a Table

You can update an attribute for an item that already exists in a table by using the AmazonDynamoDB's `updateItem` method, providing a table name, primary key value, and a map of fields to update.

Note
If the named table doesn't exist for your account and region, or if the item identified by the primary key you passed in doesn't exist, a ResourceNotFoundException is thrown.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeAction;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.AttributeValueUpdate;
import com.amazonaws.services.dynamodbv2.model.ResourceNotFoundException;
import java.util.ArrayList;
```

Code

```

HashMap<String,AttributeValue> item_key =
   new HashMap<String,AttributeValue>();

item_key.put("Name", new AttributeValue(name));

HashMap<String,AttributeValueUpdate> updated_values =
    new HashMap<String,AttributeValueUpdate>();

for (String[] field : extra_fields) {
    updated_values.put(field[0], new AttributeValueUpdate(
                new AttributeValue(field[1]), AttributeAction.PUT));
}

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

try {
    ddb.updateItem(table_name, item_key, updated_values);
} catch (ResourceNotFoundException e) {
    System.err.println(e.getMessage());
    System.exit(1);
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
```

See the complete sample.

More Info

- Guidelines for Working with Items in the *Amazon DynamoDB Developer Guide*
- Working with Items in DynamoDB in the *Amazon DynamoDB Developer Guide*

Managing Tomcat Session State with DynamoDB

Tomcat applications often store session-state data in memory. However, this approach doesn't scale well because once the application grows beyond a single web server, the session state must be shared among servers. A common solution is to set up a dedicated session-state server with MySQL. However, this approach also has drawbacks: you must administer another server, the session-state server is a single pointer of failure, and the MySQL server itself can cause performance problems.

DynamoDB, which is a NoSQL database store from AWS, avoids these drawbacks by providing an effective solution for sharing session state across web servers.

Download the Session Manager

You can download the session manager from the aws/aws-dynamodb-session-tomcat project on GitHub. This project also hosts the session manager source code, so you can contribute to the project by sending us pull requests or opening issues.

Configure the Session-State Provider

To use the DynamoDB session-state provider, you must do the following:

1. Configure the Tomcat server to use the provider.
2. Set the security credentials of the provider so that it can access AWS.

Configure a Tomcat Server to Use DynamoDB as the Session-State Server

Copy `AmazonDynamoDBSessionManagerForTomcat-1.x.x.jar` to the `lib` directory of your Tomcat installation. `AmazonDynamoDBSessionManagerForTomcat-1.x.x.jar` is a complete, standalone JAR that contains all the code and dependencies to run the DynamoDB Tomcat Session Manager.

Edit your server's `context.xml` file to specify *com.amazonaws.services.dynamodb.sessionmanager.DynamoDBSessionManager* as your session manager.

```
<?xml version="1.0" encoding="UTF-8"?>
  <Context>
      <WatchedResource>WEB-INF/web.xml</WatchedResource>
      <Manager
 className="com.amazonaws.services.dynamodb.sessionmanager.DynamoDBSessionManager"
              createIfNotExist="true" />
  </Context>
```

Configure Your AWS Security Credentials

You can specify AWS security credentials for the session manager in multiple ways. They are loaded in the following order of precedence:

1. The `AwsAccessKey` and `AwsSecretKey` attributes of the `Manager` element explicitly provide credentials.
2. The `AwsCredentialsFile` attribute on the `Manager` element specifies a properties file from which to load credentials.

If you don't specify credentials through the `Manager` element, `DefaultAWSCredentialsProviderChain` continues searching for credentials in the following order:

1. Environment variables –`AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`
2. Java system properties –`aws.accessKeyId` and `aws.secretKey`
3. Instance profile credentials delivered through the Amazon EC2 instance metadata service (IMDS)

Configure with Elastic Beanstalk

If you're using the session manager in Elastic Beanstalk, ensure your project has an `.ebextensions` directory at the top level of your output artifact structure. Put the following files in `.ebextensions` directory:

- The `AmazonDynamoDBSessionManagerForTomcat-1.x.x.jar` file

- A `context.xml` file, described previously, to configure the session manager
- A configuration file that copies the JAR into Tomcat's `lib` directory and applies the overridden `context.xml` file.

For more information about customizing Elastic Beanstalk environments, see AWS Elastic Beanstalk Environment Configuration in the *AWS Elastic Beanstalk Developer Guide*.

If you deploy to Elastic Beanstalk with the AWS Toolkit for Eclipse, you can have the toolkit set up the session manager for you; use the **New AWS Java Web Project** wizard and choose DynamoDB for session management. The AWS Toolkit for Eclipse configures the required files and puts them in the `.ebextensions` directory in the `WebContent` directory of your project. If you have problems finding this directory, be sure you aren't hiding files that begin with a period.

Manage Tomcat Session State with DynamoDB

If the Tomcat server is running on an Amazon EC2 instance that is configured to use IAM roles for EC2 instances, you don't need to specify any credentials in the `context.xml` file. In this case, the AWS SDK for Java uses IAM roles credentials obtained through the instance metadata service (IMDS).

When your application starts, it looks for a DynamoDB table named, by default, **Tomcat_SessionState**. The table should have a string hash key named "sessionId" (case-sensitive), no range key, and the desired values for `ReadCapacityUnits` and `WriteCapacityUnits`.

We recommend that you create this table before running your application for the first time. If you don't create the table, however, the extension creates it during initialization. See the `context.xml` options in the next section for a list of attributes that configure how the session-state table is created when it doesn't exist.

Note
For information about working with DynamoDB tables and provisioned throughput, see the Amazon DynamoDB Developer Guide.

After the application is configured and the table is created, you can use sessions with any other session provider.

Options Specified in context.xml

You can use the following configuration attributes in the `Manager` element of your `context.xml` file:

- *AwsAccessKey*– Access key ID to use.
- *AwsSecretKey*– Secret key to use.
- *AwsCredentialsFile*– A properties file containing `accessKey` and `secretKey` properties with your AWS security credentials.
- *Table*– Optional string attribute. The name of the table used to store session data. The default is **Tomcat_SessionState**.
- *RegionId*– Optional string attribute. The AWS Region in which to use DynamoDB. For a list of available AWS Regions, see Regions and Endpoints in the *Amazon Web Services General Reference*.
- *Endpoint*– Optional string attribute that, if present, overrides any value set for the `Region` option. This attribute specifies the regional endpoint of the DynamoDB service to use. For a list of available AWS Regions, see Regions and Endpoints in *Amazon Web Services General Reference*.
- *ReadCapacityUnits*– Optional int attribute. The read capacity units to use if the session manager creates the table. The default is 10.
- *WriteCapacityUnits*– Optional int attribute. The write capacity units to use if the session manager creates the table. The default is 5.

- *CreateIfNotExist*– Optional Boolean attribute. The `CreateIfNotExist` attribute controls whether the session manager autocreates 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 is thrown during Tomcat startup.

Troubleshooting

If you encounter issues with the session manager, the first place to look is in `catalina.out`. If you have access to the Tomcat installation, you can go directly to this log file and look for any error messages from the session manager. If you're using Elastic Beanstalk, you can view the environment logs with the AWS Management Console or the AWS Toolkit for Eclipse.

Limitations

The session manager doesn't support session locking. Therefore, applications that use many concurrent AJAX calls to manipulate session data may not be appropriate for use with the session manager, due to race conditions on session data writes and saves back to the data store.

Amazon EC2 Examples

This section provides examples of programming Amazon EC2 with the AWS SDK for Java.

Topics

Tutorial: Starting an EC2 Instance

This tutorial demonstrates how to use the AWS SDK for Java to start an EC2 instance.

Topics

Prerequisites

Before you begin, be sure that you have created an AWS account and that you have set up your AWS credentials. For more information, see Getting Started (p. 3).

Create an Amazon EC2 Security Group

Create a *security group*, which acts as a virtual firewall that controls the network traffic for one or more EC2 instances. By default, Amazon 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 a Linux instance, you must configure the security group to allow SSH traffic. You can create a security group using the Amazon EC2 console or the AWS SDK for Java.

You create a security group for use in either EC2-Classic or EC2-VPC. For more information about EC2-Classic and EC2-VPC, see Supported Platforms in the *Amazon EC2 User Guide for Linux Instances*.

For more information about creating a security group using the Amazon EC2 console, see Amazon EC2 Security Groups in the *Amazon EC2 User Guide for Linux Instances*.

To create a security group

1. Create and initialize a CreateSecurityGroupRequest instance. Use the withGroupName method to set the security group name, and the withDescription method to set the security group description, as follows:

   ```
   CreateSecurityGroupRequest csgr = new CreateSecurityGroupRequest();
   csgr.withGroupName("JavaSecurityGroup").withDescription("My security group");
   ```

   The security group name must be unique within the AWS region in which you initialize your Amazon EC2 client. You must use US-ASCII characters for the security group name and description.
2. Pass the request object as a parameter to the createSecurityGroup method. The method returns a CreateSecurityGroupResult object, as follows:

   ```
   CreateSecurityGroupResult createSecurityGroupResult =
       amazonEC2Client.createSecurityGroup(csgr);
   ```

   If you attempt to create a security group with the same name as an existing security group, `createSecurityGroup` throws an exception.

By default, a new security group does not allow any inbound traffic to your Amazon EC2 instance. To allow inbound traffic, you must explicitly authorize security group ingress. You can authorize ingress for individual IP addresses, for a range of IP addresses, for a specific protocol, and for TCP/UDP ports.

To authorize security group ingress

1. Create and initialize an IpPermission instance. Use the withIpv4Ranges method to set the range of IP addresses to authorize ingress for, and use the withIpProtocol method to set the IP protocol. Use the withFromPort and withToPort methods to specify range of ports to authorize ingress for, as follows:

   ```
   IpPermission ipPermission =
       new IpPermission();

   IpRange ipRange1 = new IpRange().withCidrIp("111.111.111.111/32");
   IpRange ipRange2 = new IpRange().withCidrIp("150.150.150.150/32");

   ipPermission.withIpv4Ranges(Arrays.asList(new IpRange[] {ipRange1, ipRange2}))
               .withIpProtocol("tcp")
               .withFromPort(22)
               .withToPort(22);
   ```

   All the conditions that you specify in the `IpPermission` object must be met in order for ingress to be allowed.

Specify the IP address using CIDR notation. If you specify the protocol as TCP/UDP, you must provide a source port and a destination port. You can authorize ports only if you specify TCP or UDP.

2. Create and initialize an AuthorizeSecurityGroupIngressRequest instance. Use the `withGroupName` method to specify the security group name, and pass the `IpPermission` object you initialized earlier to the withIpPermissions method, as follows:

```
AuthorizeSecurityGroupIngressRequest authorizeSecurityGroupIngressRequest =
    new AuthorizeSecurityGroupIngressRequest();

authorizeSecurityGroupIngressRequest.withGroupName("JavaSecurityGroup")
                                    .withIpPermissions(ipPermission);
```

3. Pass the request object into the authorizeSecurityGroupIngress method, as follows:

```
amazonEC2Client.authorizeSecurityGroupIngress(authorizeSecurityGroupIngressRequest);
```

If you call `authorizeSecurityGroupIngress` with IP addresses for which ingress is already authorized, the method throws an exception. Create and initialize a new `IpPermission` object to authorize ingress for different IPs, ports, and protocols before calling `AuthorizeSecurityGroupIngress`.

Whenever you call the authorizeSecurityGroupIngress or authorizeSecurityGroupEgress methods, a rule is added to your security group.

Create a Key Pair

You must specify a key pair when you launch an EC2 instance and then specify the private key of the key pair when you connect to the instance. You can create a key pair or use an existing key pair that you've used when launching other instances. For more information, see Amazon EC2 Key Pairs in the *Amazon EC2 User Guide for Linux Instances*.

To create a key pair and save the private key

1. Create and initialize a CreateKeyPairRequest instance. Use the withKeyName method to set the key pair name, as follows:

```
CreateKeyPairRequest createKeyPairRequest = new CreateKeyPairRequest();

createKeyPairRequest.withKeyName(keyName);
```

Important
Key pair names must be unique. If you attempt to create a key pair with the same key name as an existing key pair, you'll get an exception.

2. Pass the request object to the createKeyPair method. The method returns a CreateKeyPairResult instance, as follows:

```
CreateKeyPairResult createKeyPairResult =
  amazonEC2Client.createKeyPair(createKeyPairRequest);
```

3. Call the result object's getKeyPair method to obtain a KeyPair object. Call the `KeyPair` object's getKeyMaterial method to obtain the unencrypted PEM-encoded private key, as follows:

```
KeyPair keyPair = new KeyPair();

keyPair = createKeyPairResult.getKeyPair();
```

```
String privateKey = keyPair.getKeyMaterial();
```

Run an Amazon EC2 Instance

Use the following procedure to launch one or more identically configured EC2 instances from the same Amazon Machine Image (AMI). After you create your EC2 instances, you can check their status. After your EC2 instances are running, you can connect to them.

To launch an Amazon EC2 instance

1. Create and initialize a RunInstancesRequest instance. Make sure that the AMI, key pair, and security group that you specify exist in the region that you specified when you created the client object.

```
RunInstancesRequest runInstancesRequest =
   new RunInstancesRequest();

runInstancesRequest.withImageId("ami-4b814f22")
                   .withInstanceType("m1.small")
                   .withMinCount(1)
                   .withMaxCount(1)
                   .withKeyName("my-key-pair")
                   .withSecurityGroups("my-security-group");
```

withImageId

The ID of the AMI. For a list of public AMIs provided by Amazon, see Amazon Machine Images.

withInstanceType

An instance type that is compatible with the specified AMI. For more information, see Instance Types in the *Amazon EC2 User Guide for Linux Instances*.

withMinCount

The minimum number of EC2 instances to launch. If this is more instances than Amazon EC2 can launch in the target Availability Zone, Amazon EC2 launches no instances.

withMaxCount

The maximum number of EC2 instances to launch. If this is more instances than Amazon EC2 can launch in the target Availability Zone, Amazon EC2 launches the largest possible number of instances above `MinCount`. You can launch between 1 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.

withKeyName

The name of the EC2 key pair. If you launch an instance without specifying a key pair, you can't connect to it. For more information, see Create a Key Pair (p. 56).

withSecurityGroups

One or more security groups. For more information, see Create an Amazon EC2 Security Group (p. 55).

2. Launch the instances by passing the request object to the runInstances method. The method returns a RunInstancesResult object, as follows:

```
RunInstancesResult result = amazonEC2Client.runInstances(
                              runInstancesRequest);
```

After your instance is running, you can connect to it using your key pair. For more information, see Connect to Your Linux Instance. in the *Amazon EC2 User Guide for Linux Instances*.

Using IAM Roles to Grant Access to AWS Resources on Amazon EC2

All requests to Amazon Web Services (AWS) must be cryptographically signed using credentials issued by AWS. You can use *IAM roles* to conveniently grant secure access to AWS resources from your Amazon EC2 instances.

This topic provides information about how to use IAM roles with Java SDK applications running on Amazon EC2. For more information about IAM instances, see IAM Roles for Amazon EC2 in the *Amazon EC2 User Guide for Linux Instances*.

The default provider chain and EC2 instance profiles

If your application creates an AWS client using the default constructor, then the client will search for credentials using the *default credentials provider chain*, in the following order:

1. In system environment variables: `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`.
2. In the Java system properties: `aws.accessKeyId` and `aws.secretKey`.
3. In the default credentials file (the location of this file varies by platform).
4. In the *instance profile credentials*, which exist within the instance metadata associated with the IAM role for the EC2 instance.

The final step in the default provider chain is available only when running your application on an Amazon EC2 instance, but provides the greatest ease of use and best security when working with Amazon EC2 instances. You can also pass an InstanceProfileCredentialsProvider instance directly to the client constructor to get instance profile credentials without proceeding through the entire default provider chain.

For example:

```
AmazonS3 s3 = AmazonS3ClientBuilder.standard()
              .withCredentials(new InstanceProfileCredentialsProvider(false))
              .build();
```

When using this approach, the SDK retrieves temporary AWS credentials that have the same permissions as those associated with the IAM role associated with the Amazon EC2 instance in its instance profile. Although these credentials are temporary and would eventually expire, `InstanceProfileCredentialsProvider` periodically refreshes them for you so that the obtained credentials continue to allow access to AWS.

Important
The automatic credentials refresh happens *only* when you use the default client constructor, which creates its own `InstanceProfileCredentialsProvider` as part of the default provider chain, or when you pass an `InstanceProfileCredentialsProvider` instance directly to the client constructor. If you use another method to obtain or pass instance profile credentials, you are responsible for checking for and refreshing expired credentials.

If the client constructor can't find credentials using the credentials provider chain, it will throw an AmazonClientException.

Walkthrough: Using IAM roles for EC2 instances

The following walkthrough shows you how to retrieve an object from Amazon S3 using an IAM role to manage access.

Create an IAM Role

Create an IAM role that grants read-only access to Amazon S3.

To create the IAM role

1. Open the IAM console.
2. In the navigation pane, select **Roles**, then **Create New Role**.
3. Enter a name for the role, then select **Next Step**. Remember this name, since you'll need it when you launch your Amazon EC2 instance.
4. On the **Select Role Type** page, under **AWS Service Roles**, select **Amazon EC2**.
5. On the **Set Permissions** page, under **Select Policy Template**, select **Amazon S3 Read Only Access**, then **Next Step**.
6. On the **Review** page, select **Create Role**.

Launch an EC2 Instance and Specify Your IAM Role

You can launch an Amazon EC2 instance with an IAM role using the Amazon EC2 console or the AWS SDK for Java.

- To launch an Amazon EC2 instance using the console, follow the directions in Getting Started with Amazon EC2 Linux Instances in the *Amazon EC2 User Guide for Linux Instances*.

  When you reach the **Review Instance Launch** page, select **Edit instance details**. In **IAM role**, choose the IAM role that you created previously. Complete the procedure as directed.

  > **Note**
  > You'll need to create or use an existing security group and key pair to connect to the instance.
- To launch an Amazon EC2 instance with an IAM role using the AWS SDK for Java, see Run an Amazon EC2 Instance (p. 57).

Create your Application

Let's build the sample application to run on the EC2 instance. First, create a directory that you can use to hold your tutorial files (for example, `GetS3ObjectApp`).

Next, copy the AWS SDK for Java libraries into your newly-created directory. If you downloaded the AWS SDK for Java to your `~/Downloads` directory, you can copy them using the following commands:

```
cp -r ~/Downloads/aws-java-sdk-{1.7.5}/lib .
cp -r ~/Downloads/aws-java-sdk-{1.7.5}/third-party .
```

Open a new file, call it `GetS3Ojbect.java`, and add the following code:

```

import java.io.*;

import com.amazonaws.auth.*;
import com.amazonaws.services.s3.*;
import com.amazonaws.services.s3.model.*;
import com.amazonaws.AmazonClientException;
import com.amazonaws.AmazonServiceException;

public class GetS3Object {
  private static String bucketName = "text-content";
  private static String key = "text-object.txt";

  public static void main(String[] args) throws IOException
  {
    AmazonS3 s3Client = AmazonS3ClientBuilder.defaultClient();

    try {
      System.out.println("Downloading an object");
      S3Object s3object = s3Client.getObject(
          new GetObjectRequest(bucketName, key));
      displayTextInputStream(s3object.getObjectContent());
    }
    catch(AmazonServiceException ase) {
      System.err.println("Exception was thrown by the service");
    }
    catch(AmazonClientException ace) {
      System.err.println("Exception was thrown by the client");
    }
  }

  private static void displayTextInputStream(InputStream input) throws IOException
  {
    // Read one text line at a time and display.
    BufferedReader reader = new BufferedReader(new InputStreamReader(input));
    while(true)
    {
      String line = reader.readLine();
      if(line == null) break;
      System.out.println( "    " + line );
    }
    System.out.println();
  }
}
```

Open a new file, call it `build.xml`, and add the following lines:

```
<project name="Get Amazon S3 Object" default="run" basedir=".">
  <path id="aws.java.sdk.classpath">
    <fileset dir="./lib" includes="**/*.jar"/>
    <fileset dir="./third-party" includes="**/*.jar"/>
    <pathelement location="lib"/>
    <pathelement location="."/>
  </path>

  <target name="build">
  <javac debug="true"
    includeantruntime="false"
    srcdir="."
    destdir="."
    classpathref="aws.java.sdk.classpath"/>
  </target>

  <target name="run" depends="build">
    <java classname="GetS3Object" classpathref="aws.java.sdk.classpath" fork="true"/>
  </target>
</project>
```

Build and run the modified program. Note that there are no credentials are stored in the program. Therefore, unless you have your AWS credentials specified already, the code will throw `AmazonServiceException`. For example:

```
$ ant
Buildfile: /path/to/my/GetS3ObjectApp/build.xml

build:
  [javac] Compiling 1 source file to /path/to/my/GetS3ObjectApp

run:
   [java] Downloading an object
   [java] AmazonServiceException

BUILD SUCCESSFUL
```

Transfer the Compiled Program to Your EC2 Instance

Transfer the program to your Amazon EC2 instance using secure copy (**`scp`**), along with the AWS SDK for Java libraries. The sequence of commands looks something like the following.

```
scp -p -i {my-key-pair}.pem GetS3Object.class ec2-user@{public_dns}:GetS3Object.class
scp -p -i {my-key-pair}.pem build.xml ec2-user@{public_dns}:build.xml
scp -r -p -i {my-key-pair}.pem lib ec2-user@{public_dns}:lib
scp -r -p -i {my-key-pair}.pem third-party ec2-user@{public_dns}:third-party
```

Note
Depending on the Linux distribution that you used, the *user name* might be "ec2-user", "root", or "ubuntu". To get the public DNS name of your instance, open the EC2 console and look for the **Public DNS** value in the **Description** tab (for example, `ec2-198-51-100-1.compute-1.amazonaws.com`).

In the preceding commands:

- `GetS3Object.class` is your compiled program
- `build.xml` is the ant file used to build and run your program
- the `lib` and `third-party` directories are the corresponding library folders from the AWS SDK for Java.
- The `-r` switch indicates that `scp` should do a recursive copy of all of the contents of the `library` and `third-party` directories in the AWS SDK for Java distribution.
- The `-p` switch indicates that `scp` should preserve the permissions of the source files when it copies them to the destination.

  Note
  The `-p` switch works only on Linux, macOS, or Unix. If you are copying files from Windows, you may need to fix the file permissions on your instance using the following command:

  ```
  chmod -R u+rwx GetS3Object.class build.xml lib third-party
  ```

Run the Sample Program on the EC2 Instance

To run the program, connect to your Amazon EC2 instance. For more information, see Connect to Your Linux Instance in the *Amazon EC2 User Guide for Linux Instances*.

If **`ant`** is not available on your instance, install it using the following command:

```
sudo yum install ant
```

Then, run the program using `ant` as follows:

```
ant run
```

The program will write the contents of your Amazon S3 object to your command window.

Tutorial: Amazon EC2 Spot Instances

Overview

Spot Instances allow you to bid on unused Amazon Elastic Compute Cloud (Amazon EC2) capacity and run the acquired instances for as long as your bid exceeds the current *Spot Price*. Amazon EC2 changes the Spot Price periodically based on supply and demand, and customers whose bids meet or exceed it gain access to the available Spot Instances. Like On-Demand Instances and Reserved Instances, Spot Instances provide you another option for obtaining more compute capacity.

Spot Instances can significantly lower your Amazon EC2 costs for batch processing, scientific research, image processing, video encoding, data and web crawling, financial analysis, and testing. Additionally, Spot Instances give you access to large amounts of additional capacity in situations where the need for that capacity is not urgent.

To use Spot Instances, place a Spot Instance request specifying the maximum price you are willing to pay per instance hour; this is your bid. If your bid exceeds the current Spot Price, your request is fulfilled and your instances will run until either you choose to terminate them or the Spot Price increases above your bid (whichever is sooner).

It's important to note:

- You will often pay less per hour than your bid. Amazon EC2 adjusts the Spot Price periodically as requests come in and available supply changes. Everyone pays the same Spot Price for that period regardless of whether their bid was higher. Therefore, you might pay less than your bid, but you will never pay more than your bid.
- If you're running Spot Instances and your bid no longer meets or exceeds the current Spot Price, your instances will be terminated. This means that you will want to make sure that your workloads and applications are flexible enough to take advantage of this opportunistic capacity.

Spot Instances perform exactly like other Amazon EC2 instances while running, and like other Amazon EC2 instances, Spot Instances can be terminated when you no longer need them. If you terminate your instance, you pay for any partial hour used (as you would for On-Demand or Reserved Instances). However, if the Spot Price goes above your bid and your instance is terminated by Amazon EC2, you will not be charged for any partial hour of usage.

This tutorial shows how to use AWS SDK for Java to do the following.

- Submit a Spot Request
- Determine when the Spot Request becomes fulfilled
- Cancel the Spot Request
- Terminate associated instances

Prerequisites

To use this tutorial you must have the AWS SDK for Java installed, as well as having met its basic installation prerequisites. See Set up the AWS SDK for Java (p. 4) for more information.

Step 1: Setting Up Your Credentials

To begin using this code sample, you need to add AWS credentials to the `AwsCredentials.properties` file as follows:

1. Open the *AwsCredentials.properties* file.
2. Set your access key / secret key id combination in the `AwsCredentials.properties` file.

Note
We recommend that you use the credentials of an IAM user to provide these values. For more information, see Sign Up for AWS and Create an IAM User (p. 3).

Now that you have configured your settings, you can get started using the code in the example.

Step 2: Setting Up a Security Group

A *security group* acts as a firewall that controls the traffic allowed in and out of a group of instances. By default, an instance is started without any security group, which means that all incoming IP traffic, on any TCP port will be denied. So, before submitting our Spot Request, we will set up a security group that allows the necessary network traffic. For the purposes of this tutorial, we will create a new security group called "GettingStarted" that allows Secure Shell (SSH) traffic from the IP address where you are running your application from. To set up a new security group, you need to include or run the following code sample that sets up the security group programmatically.

After we create an `AmazonEC2` client object, we create a `CreateSecurityGroupRequest` object with the name, "GettingStarted" and a description for the security group. Then we call the `ec2.createSecurityGroup` API to create the group.

To enable access to the group, we create an `ipPermission` object with the IP address range set to the CIDR representation of the subnet for the local computer; the "/10" suffix on the IP address indicates the subnet for the specified IP address. We also configure the `ipPermission` object with the TCP protocol and port 22 (SSH). The final step is to call `ec2.authorizeSecurityGroupIngress` with the name of our security group and the `ipPermission` object.

```
<?dbhtml linenumbering.everyNth="1" ?>
// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Create a new security group.
try {
    CreateSecurityGroupRequest securityGroupRequest = new
 CreateSecurityGroupRequest("GettingStartedGroup", "Getting Started Security Group");
    ec2.createSecurityGroup(securityGroupRequest);
} catch (AmazonServiceException ase) {
    // Likely this means that the group is already created, so ignore.
    System.out.println(ase.getMessage());
}

String ipAddr = "0.0.0.0/0";

// Get the IP of the current host, so that we can limit the Security
// Group by default to the ip range associated with your subnet.
try {
    InetAddress addr = InetAddress.getLocalHost();

    // Get IP Address
    ipAddr = addr.getHostAddress()+"/10";
} catch (UnknownHostException e) {
}
```

```
// Create a range that you would like to populate.
ArrayList<String> ipRanges = new ArrayList<String>();
ipRanges.add(ipAddr);

// Open up port 22 for TCP traffic to the associated IP
// from above (e.g. ssh traffic).
ArrayList<IpPermission> ipPermissions = new ArrayList<IpPermission> ();
IpPermission ipPermission = new IpPermission();
ipPermission.setIpProtocol("tcp");
ipPermission.setFromPort(new Integer(22));
ipPermission.setToPort(new Integer(22));
ipPermission.setIpRanges(ipRanges);
ipPermissions.add(ipPermission);

try {
    // Authorize the ports to the used.
    AuthorizeSecurityGroupIngressRequest ingressRequest =
        new AuthorizeSecurityGroupIngressRequest("GettingStartedGroup",ipPermissions);
    ec2.authorizeSecurityGroupIngress(ingressRequest);
} catch (AmazonServiceException ase) {
    // Ignore because this likely means the zone has
    // already been authorized.
    System.out.println(ase.getMessage());
}
```

You can view this entire code sample in the `CreateSecurityGroupApp.java` code sample. Note you only need to run this application once to create a new security group.

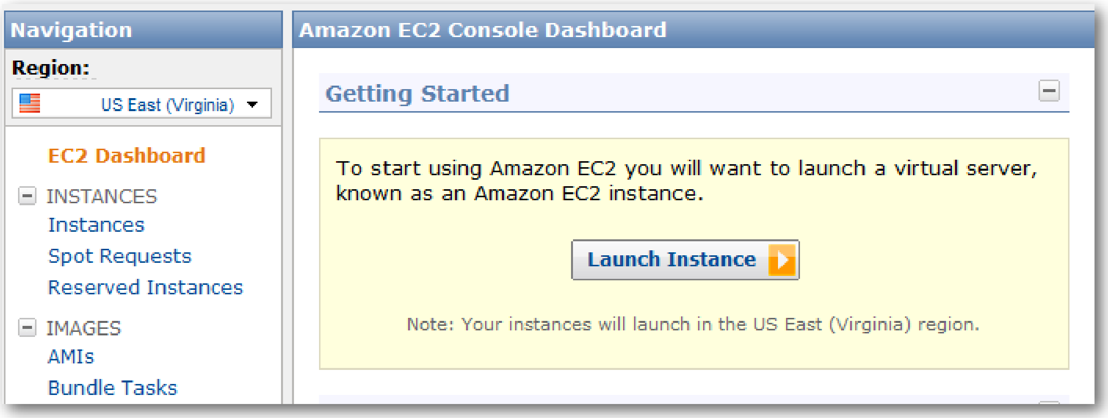
You can also create the security group using the AWS Toolkit for Eclipse. See Managing Security Groups from AWS Explorer for more information.

Step 3: Submitting Your Spot Request

To submit a Spot request, you first need to determine the instance type, Amazon Machine Image (AMI), and maximum bid price you want to use. You must also include the security group we configured previously, so that you can log into the instance if desired.

There are several instance types to choose from; go to Amazon EC2 Instance Types for a complete list. For this tutorial, we will use t1.micro, the cheapest instance type available. Next, we will determine the type of AMI we would like to use. We'll use ami-8c1fece5, the most up-to-date Amazon Linux AMI available when we wrote this tutorial. The latest AMI may change over time, but you can always determine the latest version AMI by following these steps:

1. Log into the AWS Management Console, click the **EC2** tab, and, from the EC2 Console Dashboard, attempt to launch an instance.

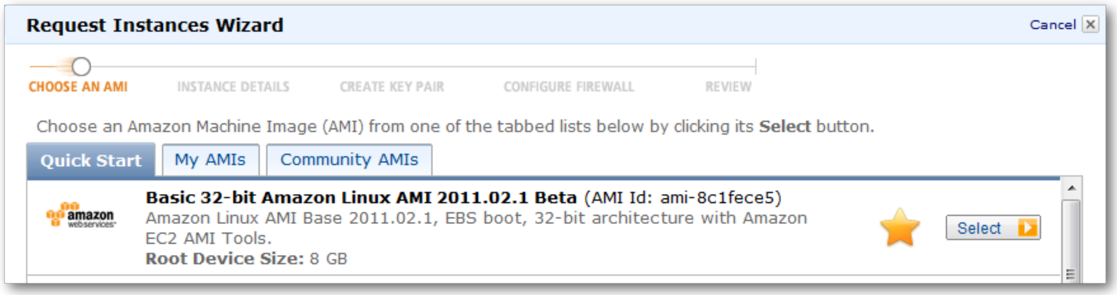


To start using Amazon EC2 you will want to launch a virtual server, known as an Amazon EC2 instance.

Note: Your instances will launch in the US East (Virginia) region.

AWS Management Console to launch an instance

2. In the window that displays AMIs, just use the AMI ID as shown in the following screen shot. Alternatively, you can use the `DescribeImages` API, but leveraging that command is outside the scope of this tutorial.


Request Instances Wizard

Choose an Amazon Machine Image (AMI) from one of the tabbed lists below by clicking its **Select** button.

Basic 32-bit Amazon Linux AMI 2011.02.1 Beta (AMI Id: ami-8c1fece5)
Amazon Linux AMI Base 2011.02.1, EBS boot, 32-bit architecture with Amazon EC2 AMI Tools.
Root Device Size: 8 GB

Identifying the most-recent AMI

There are many ways to approach bidding for Spot instances; to get a broad overview of the various approaches you should view the Bidding for Spot Instances video. However, to get started, we'll describe three common strategies: bid to ensure cost is less than on-demand pricing; bid based on the value of the resulting computation; bid so as to acquire computing capacity as quickly as possible.

- *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 when it completes. You want to see if you can complete it for less cost than with On-Demand Instances. You examine the Spot Price history for instance types using either the AWS Management Console or the Amazon EC2 API. For more information, go to Viewing Spot Price History. After you've analyzed the price history for your desired instance type in a given Availability Zone, you have two alternative approaches for your bid:
  - You could bid at the upper end of the range of Spot Prices (which are still below the On-Demand price), anticipating that your one-time Spot request would most likely be fulfilled and run for enough consecutive compute time to complete the job.
  - Or, you could bid 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. (We will explain how to automate this task later in this tutorial.)
- *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 are worth in terms of computing costs. After you've analyzed the Spot Price history for your instance type, you choose a bid price at which the cost of the computing time is no more than the value of the job's results. You create a persistent bid and allow it to run intermittently as the Spot Price fluctuates at or below your bid.
- *Acquire Computing Capacity Quickly* You have an unanticipated, short-term need for additional capacity that is not available through On-Demand Instances. After you've analyzed the Spot Price history for your instance type, you bid above the highest historical price to provide a high likelihood that your request will be fulfilled quickly and continue computing until it completes.

After you choose your bid price, you are ready to request a Spot Instance. For the purposes of this tutorial, we will bid the On-Demand price ($0.03) to maximize the chances that the bid will be fulfilled. You can determine the types of available instances and the On-Demand prices for instances by going to Amazon EC2 Pricing page. To request a Spot Instance, you simply need to build your request with the parameters you chose earlier. We start by creating a `RequestSpotInstanceRequest` object. The request object requires the number of instances you want to start and the bid price. Additionally, you need to set the `LaunchSpecification` for the request, which includes the instance type, AMI ID, and security group you want to use. Once the request is populated, you call the `requestSpotInstances` method on the `AmazonEC2Client` object. The following example shows how to request a Spot Instance.

```
// Retrieves the credentials from a AWSCrentials.properties file.
AWSCredentials credentials = null;
try {
    credentials = new PropertiesCredentials(
        GettingStartedApp.class.getResourceAsStream("AwsCredentials.properties"));
} catch (IOException e1) {
    System.out.println("Credentials were not properly entered into
 AwsCredentials.properties.");
    System.out.println(e1.getMessage());
    System.exit(-1);
}

// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 1 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(1));

// Setup the specifications of the launch. This includes the
// instance type (e.g. t1.micro) and the latest Amazon Linux
// AMI id available. Note, you should always use the latest
// Amazon Linux AMI id or another of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-8c1fece5");
launchSpecification.setInstanceType("t1.micro");

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Add the launch specifications to the request.
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult = ec2.requestSpotInstances(requestRequest);
```

Running this code will launch a new Spot Instance Request. There are other options you can use to configure your Spot Requests. To learn more, please visit Tutorial: Advanced Amazon EC2 Spot Request Management (p. 70) or the RequestSpotInstances class in the *AWS SDK for Java API Reference*.

Note
You will be charged for any Spot Instances that are actually launched, so make sure that you cancel any requests and terminate any instances you launch to reduce any associated fees.

Step 4: Determining the State of Your Spot Request

Next, we want to create code to wait until the Spot request reaches the "active" state before proceeding to the last step. To determine the state of our Spot request, we poll the describeSpotInstanceRequests method for the state of the Spot request ID we want to monitor.

The request ID created in Step 2 is embedded in the response to our `requestSpotInstances` request. The following example code shows how to gather request IDs from the `requestSpotInstances` response and use them to populate an `ArrayList`.

```
// Call the RequestSpotInstance API.
```

```
RequestSpotInstancesResult requestResult = ec2.requestSpotInstances(requestRequest);
List<SpotInstanceRequest> requestResponses = requestResult.getSpotInstanceRequests();

// Setup an arraylist to collect all of the request ids we want to
// watch hit the running state.
ArrayList<String> spotInstanceRequestIds = new ArrayList<String>();

// Add all of the request ids to the hashset, so we can determine when they hit the
// active state.
for (SpotInstanceRequest requestResponse : requestResponses) {
    System.out.println("Created Spot Request:
 "+requestResponse.getSpotInstanceRequestId());
    spotInstanceRequestIds.add(requestResponse.getSpotInstanceRequestId());
}
```

To monitor your request ID, call the `describeSpotInstanceRequests` method to determine the state of the request. Then loop until the request is not in the "open" state. Note that we monitor for a state of not "open", rather a state of, say, "active", because the request can go straight to "closed" if there is a problem with your request arguments. The following code example provides the details of how to accomplish this task.

```
// Create a variable that will track whether there are any
// requests still in the open state.
boolean anyOpen;

do {
    // Create the describeRequest object with all of the request ids
    // to monitor (e.g. that we started).
    DescribeSpotInstanceRequestsRequest describeRequest = new
 DescribeSpotInstanceRequestsRequest();
    describeRequest.setSpotInstanceRequestIds(spotInstanceRequestIds);

    // Initialize the anyOpen variable to false - which assumes there
    // are no requests open unless we find one that is still open.
    anyOpen=false;

    try {
        // Retrieve all of the requests we want to monitor.
        DescribeSpotInstanceRequestsResult describeResult =
 ec2.describeSpotInstanceRequests(describeRequest);
        List<SpotInstanceRequest> describeResponses =
 describeResult.getSpotInstanceRequests();

        // Look through each request and determine if they are all in
        // the active state.
        for (SpotInstanceRequest describeResponse : describeResponses) {
            // If the state is open, it hasn't changed since we attempted
            // to request it. There is the potential for it to transition
            // almost immediately to closed or cancelled so we compare
            // against open instead of active.
        if (describeResponse.getState().equals("open")) {
            anyOpen = true;
            break;
        }
    }
} catch (AmazonServiceException e) {
      // If we have an exception, ensure we don't break out of
      // the loop. This prevents the scenario where there was
      // blip on the wire.
      anyOpen = true;
    }

    try {
        // Sleep for 60 seconds.
        Thread.sleep(60*1000);
    } catch (Exception e) {
        // Do nothing because it woke up early.
    }
} while (anyOpen);
```

After running this code, your Spot Instance Request will have completed or will have failed with an error that will be output to the screen. In either case, we can proceed to the next step to clean up any active requests and terminate any running instances.

Step 5: Cleaning Up Your Spot Requests and Instances

Lastly, we need to clean up our requests and instances. It is important to both cancel any outstanding requests *and* terminate any instances. Just canceling your requests will not terminate your instances, which means that you will continue to pay for them. If you terminate your instances, your Spot requests may be canceled, but there are some scenarios—such as if you use persistent bids|mdash|where terminating your instances is not sufficient to stop your request from being re-fulfilled. Therefore, it is a best practice to both cancel any active bids and terminate any running instances.

The following code demonstrates how to cancel your requests.

```
try {
    // Cancel requests.
    CancelSpotInstanceRequestsRequest cancelRequest =
       new CancelSpotInstanceRequestsRequest(spotInstanceRequestIds);
    ec2.cancelSpotInstanceRequests(cancelRequest);
} catch (AmazonServiceException e) {
    // Write out any exceptions that may have occurred.
    System.out.println("Error cancelling instances");
    System.out.println("Caught Exception: " + e.getMessage());
    System.out.println("Reponse Status Code: " + e.getStatusCode());
    System.out.println("Error Code: " + e.getErrorCode());
    System.out.println("Request ID: " + e.getRequestId());
}
```

To terminate any outstanding instances, you will need the instance ID associated with the request that started them. The following code example takes our original code for monitoring the instances and adds an `ArrayList` in which we store the instance ID associated with the `describeInstance` response.

```
// Create a variable that will track whether there are any requests
// still in the open state.
boolean anyOpen;
// Initialize variables.
ArrayList<String> instanceIds = new ArrayList<String>();

do {
   // Create the describeRequest with all of the request ids to
   // monitor (e.g. that we started).
   DescribeSpotInstanceRequestsRequest describeRequest = new
 DescribeSpotInstanceRequestsRequest();
   describeRequest.setSpotInstanceRequestIds(spotInstanceRequestIds);

   // Initialize the anyOpen variable to false, which assumes there
   // are no requests open unless we find one that is still open.
   anyOpen = false;

   try {
         // Retrieve all of the requests we want to monitor.
         DescribeSpotInstanceRequestsResult describeResult =
            ec2.describeSpotInstanceRequests(describeRequest);
         List<SpotInstanceRequest> describeResponses =
            describeResult.getSpotInstanceRequests();

         // Look through each request and determine if they are all
         // in the active state.
         for (SpotInstanceRequest describeResponse : describeResponses) {
           // If the state is open, it hasn't changed since we
           // attempted to request it. There is the potential for
           // it to transition almost immediately to closed or
           // cancelled so we compare against open instead of active.
           if (describeResponse.getState().equals("open")) {
              anyOpen = true; break;
           }
           // Add the instance id to the list we will
           // eventually terminate.
           instanceIds.add(describeResponse.getInstanceId());
         }
   } catch (AmazonServiceException e) {
      // If we have an exception, ensure we don't break out
      // of the loop. This prevents the scenario where there
      // was blip on the wire.
      anyOpen = true;
   }

    try {
        // Sleep for 60 seconds.
        Thread.sleep(60*1000);
    } catch (Exception e) {
        // Do nothing because it woke up early.
    }
} while (anyOpen);
```

Using the instance IDs, stored in the `ArrayList`, terminate any running instances using the following code snippet.

```
try {
    // Terminate instances.
    TerminateInstancesRequest terminateRequest = new
 TerminateInstancesRequest(instanceIds);
    ec2.terminateInstances(terminateRequest);
} catch (AmazonServiceException e) {
    // Write out any exceptions that may have occurred.
    System.out.println("Error terminating instances");
    System.out.println("Caught Exception: " + e.getMessage());
    System.out.println("Reponse Status Code: " + e.getStatusCode());
    System.out.println("Error Code: " + e.getErrorCode());
    System.out.println("Request ID: " + e.getRequestId());
}
```

Bringing It All Together

To bring this all together, we provide a more object-oriented approach that combines the preceding steps we showed: initializing the EC2 Client, submitting the Spot Request, determining when the Spot Requests are no longer in the open state, and cleaning up any lingering Spot request and associated instances. We create a class called `Requests` that performs these actions.

We also create a `GettingStartedApp` class, which has a main method where we perform the high level function calls. Specifically, we initialize the `Requests` object described previously. We submit the Spot Instance request. Then we wait for the Spot request to reach the "Active" state. Finally, we clean up the requests and instances.

The complete source code for this example can be viewed or downloaded at GitHub.

Congratulations! You have just completed the getting started tutorial for developing Spot Instance software with the AWS SDK for Java.

Next Steps

Proceed with Tutorial: Advanced Amazon EC2 Spot Request Management (p. 70).

Tutorial: Advanced Amazon EC2 Spot Request Management

Amazon EC2 spot instances allow you to bid on unused Amazon EC2 capacity and run those instances for as long as your bid exceeds the current *spot price*. Amazon EC2 changes the spot price periodically based on supply and demand. For more information about spot instances, see Spot Instances in the *Amazon EC2 User Guide for Linux Instances*.

Prerequisites

To use this tutorial you must have the AWS SDK for Java installed, as well as having met its basic installation prerequisites. See Set up the AWS SDK for Java (p. 4) for more information.

Setting up your credentials

To begin using this code sample, you need to add AWS credentials to the `AwsCredentials.properties` file as follows:

1. Open the *AwsCredentials.properties* file.
2. Set your access key / secret key id combination in the `AwsCredentials.properties` file.

> **Note**
> We recommend that you use the credentials of an IAM user to provide these values. For more information, see Sign Up for AWS and Create an IAM User (p. 3).

Now that you have configured your settings, you can get started using the code in the example.

Setting up a security group

A security group acts as a firewall that controls the traffic allowed in and out of a group of instances. By default, an instance is started without any security group, which means that all incoming IP traffic, on any TCP port will be denied. So, before submitting our Spot Request, we will set up a security group that allows the necessary network traffic. For the purposes of this tutorial, we will create a new security group called "GettingStarted" that allows Secure Shell (SSH) traffic from the IP address where you are running your application from. To set up a new security group, you need to include or run the following code sample that sets up the security group programmatically.

After we create an `AmazonEC2` client object, we create a `CreateSecurityGroupRequest` object with the name, "GettingStarted" and a description for the security group. Then we call the `ec2.createSecurityGroup` API to create the group.

To enable access to the group, we create an `ipPermission` object with the IP address range set to the CIDR representation of the subnet for the local computer; the "/10" suffix on the IP address indicates the subnet for the specified IP address. We also configure the `ipPermission` object with the TCP protocol and port 22 (SSH). The final step is to call `ec2.authorizeSecurityGroupIngress` with the name of our security group and the `ipPermission` object.

(The following code is the same as what we used in the first tutorial.)

```
// Retrieves the credentials from the shared credentials file
AWSCredentialsProvider credentials = new ProfileCredentialsProvider("my-profile");

// Create the AmazonEC2Client object so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.standard()
                    .withCredentials(credentials)
                    .build();

// Create a new security group.
try {
    CreateSecurityGroupRequest securityGroupRequest =
        new CreateSecurityGroupRequest("GettingStartedGroup",
        "Getting Started Security Group");
    ec2.createSecurityGroup(securityGroupRequest);
} catch (AmazonServiceException ase) {
    // Likely this means that the group is already created, so ignore.
    System.out.println(ase.getMessage());
}

String ipAddr = "0.0.0.0/0";

// Get the IP of the current host, so that we can limit the Security Group
// by default to the ip range associated with your subnet.
try {
    // Get IP Address
    InetAddress addr = InetAddress.getLocalHost();
    ipAddr = addr.getHostAddress()+"/10";
}
catch (UnknownHostException e) {
    // Fail here...
}

// Create a range that you would like to populate.
ArrayList<String> ipRanges = new ArrayList<String>();
ipRanges.add(ipAddr);

// Open up port 22 for TCP traffic to the associated IP from
// above (e.g. ssh traffic).
ArrayList<IpPermission> ipPermissions = new ArrayList<IpPermission> ();
IpPermission ipPermission = new IpPermission();
ipPermission.setIpProtocol("tcp");
ipPermission.setFromPort(new Integer(22));
ipPermission.setToPort(new Integer(22));
ipPermission.setIpRanges(ipRanges);
ipPermissions.add(ipPermission);

try {
    // Authorize the ports to the used.
    AuthorizeSecurityGroupIngressRequest ingressRequest =
        new AuthorizeSecurityGroupIngressRequest(
            "GettingStartedGroup",ipPermissions);
    ec2.authorizeSecurityGroupIngress(ingressRequest);
}
catch (AmazonServiceException ase) {
    // Ignore because this likely means the zone has already
    // been authorized.
    System.out.println(ase.getMessage());
}
```

You can view this entire code sample in the `advanced.CreateSecurityGroupApp.java` code sample. Note you only need to run this application once to create a new security group.

> **Note**
> You can also create the security group using the AWS Toolkit for Eclipse. See Managing Security Groups from AWS Explorer in the *AWS Toolkit for Eclipse User Guide* for more information.

Detailed spot instance request creation options

As we explained in Tutorial: Amazon EC2 Spot Instances (p. 62), you need to build your request with an instance type, an Amazon Machine Image (AMI), and maximum bid price.

Let's start by creating a `RequestSpotInstanceRequest` object. The request object requires the number of instances you want and the bid price. Additionally, we need to set the `LaunchSpecification` for the request, which includes the instance type, AMI ID, and security group you want to use. After the request is populated, we call the `requestSpotInstances` method on the `AmazonEC2Client` object. An example of how to request a Spot instance follows.

(The following code is the same as what we used in the first tutorial.)

```
// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 1 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(1));

// Set up the specifications of the launch. This includes the
// instance type (e.g. t1.micro) and the latest Amazon Linux
// AMI id available. Note, you should always use the latest
// Amazon Linux AMI id or another of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-8c1fece5");
launchSpecification.setInstanceType("t1.micro");

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Add the launch specification.
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult =
    ec2.requestSpotInstances(requestRequest);
```

Persistent vs. one-time requests

When building a Spot request, you can specify several optional parameters. The first is whether your request is one-time only or persistent. By default, it is a one-time request. A one-time request can be fulfilled only once, and after the requested instances are terminated, the request will be closed. A persistent request is considered for fulfillment whenever there is no Spot Instance running for the same request. To specify the type of request, you simply need to set the Type on the Spot request. This can be done with the following code.

```
// Retrieves the credentials from an AWSCredentials.properties file.
```

```
AWSCredentials credentials = null;
try {
    credentials = new PropertiesCredentials(
        GettingStartedApp.class.getResourceAsStream("AwsCredentials.properties"));
}
catch (IOException e1) {
    System.out.println(
        "Credentials were not properly entered into AwsCredentials.properties.");
    System.out.println(e1.getMessage());
    System.exit(-1);
}

// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest =
    new RequestSpotInstancesRequest();

// Request 1 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(1));

// Set the type of the bid to persistent.
requestRequest.setType("persistent");

// Set up the specifications of the launch. This includes the
// instance type (e.g. t1.micro) and the latest Amazon Linux
// AMI id available. Note, you should always use the latest
// Amazon Linux AMI id or another of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-8c1fece5");
launchSpecification.setInstanceType("t1.micro");

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Add the launch specification.
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult =
    ec2.requestSpotInstances(requestRequest);
```

Limiting the duration of a request

You can also optionally specify the length of time that your request will remain valid. You can specify both a starting and ending time for this period. By default, a Spot request will be considered for fulfillment from the moment it is created until it is either fulfilled or canceled by you. However you can constrain the validity period if you need to. An example of how to specify this period is shown in the following code.

```
// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 1 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(1));
```

```
// Set the valid start time to be two minutes from now.
Calendar cal = Calendar.getInstance();
cal.add(Calendar.MINUTE, 2);
requestRequest.setValidFrom(cal.getTime());

// Set the valid end time to be two minutes and two hours from now.
cal.add(Calendar.HOUR, 2);
requestRequest.setValidUntil(cal.getTime());

// Set up the specifications of the launch. This includes
// the instance type (e.g. t1.micro)

// and the latest Amazon Linux AMI id available.
// Note, you should always use the latest Amazon
// Linux AMI id or another of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-8c1fece5");
launchSpecification.setInstanceType("t1.micro");

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Add the launch specification.
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult = ec2.requestSpotInstances(requestRequest);
```

Grouping your Amazon EC2 spot instance requests

You have the option of grouping your Spot instance requests in several different ways. We'll look at the benefits of using launch groups, Availability Zone groups, and placement groups.

If you want to ensure your Spot instances are all launched and terminated together, then you have the option to leverage a launch group. A launch group is a label that groups a set of bids together. All instances in a launch group are started and terminated together. Note, if instances in a launch group have already been fulfilled, there is no guarantee that new instances launched with the same launch group will also be fulfilled. An example of how to set a Launch Group is shown in the following code example.

```
// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 5 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(5));

// Set the launch group.
requestRequest.setLaunchGroup("ADVANCED-DEMO-LAUNCH-GROUP");

// Set up the specifications of the launch. This includes
// the instance type (e.g. t1.micro) and the latest Amazon Linux
// AMI id available. Note, you should always use the latest
// Amazon Linux AMI id or another of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-8c1fece5");
launchSpecification.setInstanceType("t1.micro");
```

```
// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Add the launch specification.
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult =
    ec2.requestSpotInstances(requestRequest);
```

If you want to ensure that all instances within a request are launched in the same Availability Zone, and you don't care which one, you can leverage Availability Zone groups. An Availability Zone group is a label that groups a set of instances together in the same Availability Zone. All instances that share an Availability Zone group and are fulfilled at the same time will start in the same Availability Zone. An example of how to set an Availability Zone group follows.

```
// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 5 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(5));

// Set the availability zone group.
requestRequest.setAvailabilityZoneGroup("ADVANCED-DEMO-AZ-GROUP");

// Set up the specifications of the launch.  This includes the instance
// type (e.g.  t1.micro) and the latest Amazon Linux AMI id available.
// Note, you should always use the latest Amazon Linux AMI id or another
// of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-8c1fece5");
launchSpecification.setInstanceType("t1.micro");

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Add the launch specification.
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult =
    ec2.requestSpotInstances(requestRequest);
```

You can specify an Availability Zone that you want for your Spot Instances. The following code example shows you how to set an Availability Zone.

```
// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 1 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(1));

// Set up the specifications of the launch. This includes the instance
// type (e.g. t1.micro) and the latest Amazon Linux AMI id available.
// Note, you should always use the latest Amazon Linux AMI id or another
// of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-8c1fece5");
launchSpecification.setInstanceType("t1.micro");

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Set up the availability zone to use. Note we could retrieve the
// availability zones using the ec2.describeAvailabilityZones() API. For
// this demo we will just use us-east-1a.
SpotPlacement placement = new SpotPlacement("us-east-1b");
launchSpecification.setPlacement(placement);

// Add the launch specification.
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult =
    ec2.requestSpotInstances(requestRequest);
```

Lastly, you can specify a *placement group* if you are using High Performance Computing (HPC) Spot instances, such as cluster compute instances or cluster GPU instances. Placement groups provide you with lower latency and high-bandwidth connectivity between the instances. An example of how to set a placement group follows.

```
// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 1 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(1));

// Set up the specifications of the launch. This includes the instance
// type (e.g. t1.micro) and the latest Amazon Linux AMI id available.
// Note, you should always use the latest Amazon Linux AMI id or another
// of your choosing.

LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-8c1fece5");
launchSpecification.setInstanceType("t1.micro");

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Set up the placement group to use with whatever name you desire.
// For this demo we will just use "ADVANCED-DEMO-PLACEMENT-GROUP".
SpotPlacement placement = new SpotPlacement();
placement.setGroupName("ADVANCED-DEMO-PLACEMENT-GROUP");
launchSpecification.setPlacement(placement);

// Add the launch specification.
```

```
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult =
    ec2.requestSpotInstances(requestRequest);
```

All of the parameters shown in this section are optional. It is also important to realize that most of these parameters—with the exception of whether your bid is one-time or persistent—can reduce the likelihood of bid fulfillment. So, it is important to leverage these options only if you need them. All of the preceding code examples are combined into one long code sample, which can be found in the `com.amazonaws.codesamples.advanced.InlineGettingStartedCodeSampleApp.java` class.

How to persist a root partition after interruption or termination

One of the easiest ways to manage interruption of your Spot instances is to ensure that your data is checkpointed to an Amazon Elastic Block Store (Amazon EBS) volume on a regular cadence. By checkpointing periodically, if there is an interruption you will lose only the data created since the last checkpoint (assuming no other non-idempotent actions are performed in between). To make this process easier, you can configure your Spot Request to ensure that your root partition will not be deleted on interruption or termination. We've inserted new code in the following example that shows how to enable this scenario.

In the added code, we create a `BlockDeviceMapping` object and set its associated Elastic Block Storage (EBS) to an EBS object that we've configured to `not` be deleted if the Spot Instance is terminated. We then add this `BlockDeviceMapping` to the ArrayList of mappings that we include in the launch specification.

```
// Retrieves the credentials from an AWSCredentials.properties file.
AWSCredentials credentials = null;
try {
    credentials = new PropertiesCredentials(
        GettingStartedApp.class.getResourceAsStream("AwsCredentials.properties"));
}
catch (IOException e1) {
    System.out.println(
        "Credentials were not properly entered into AwsCredentials.properties.");
    System.out.println(e1.getMessage());
    System.exit(-1);
}

// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 1 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(1));

// Set up the specifications of the launch. This includes the instance
// type (e.g. t1.micro) and the latest Amazon Linux AMI id available.
// Note, you should always use the latest Amazon Linux AMI id or another
// of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-8c1fece5");
launchSpecification.setInstanceType("t1.micro");

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);
```

```
// Create the block device mapping to describe the root partition.
BlockDeviceMapping blockDeviceMapping = new BlockDeviceMapping();
blockDeviceMapping.setDeviceName("/dev/sda1");

// Set the delete on termination flag to false.
EbsBlockDevice ebs = new EbsBlockDevice();
ebs.setDeleteOnTermination(Boolean.FALSE);
blockDeviceMapping.setEbs(ebs);

// Add the block device mapping to the block list.
ArrayList<BlockDeviceMapping> blockList = new ArrayList<BlockDeviceMapping>();
blockList.add(blockDeviceMapping);

// Set the block device mapping configuration in the launch specifications.
launchSpecification.setBlockDeviceMappings(blockList);

// Add the launch specification.
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult =
    ec2.requestSpotInstances(requestRequest);
```

Assuming you wanted to re-attach this volume to your instance on startup, you can also use the block device mapping settings. Alternatively, if you attached a non-root partition, you can specify the Amazon EBS volumes you want to attach to your Spot instance after it resumes. You do this simply by specifying a snapshot ID in your `EbsBlockDevice` and alternative device name in your `BlockDeviceMapping` objects. By leveraging block device mappings, it can be easier to bootstrap your instance.

Using the root partition to checkpoint your critical data is a great way to manage the potential for interruption of your instances. For more methods on managing the potential of interruption, please visit the Managing Interruption video.

How to tag your spot requests and instances

Adding tags to EC2 resources can simplify the administration of your cloud infrastructure. A form of metadata, tags can be used to create user-friendly names, enhance searchability, and improve coordination between multiple users. You can also use tags to automate scripts and portions of your processes. To read more about tagging Amazon EC2 resources, go to Using Tags in the *Amazon EC2 User Guide for Linux Instances*.

Tagging requests

To add tags to your spot requests, you need to tag them *after* they have been requested. The return value from `requestSpotInstances()` provides you with a RequestSpotInstancesResult object that you can use to get the spot request IDs for tagging:

```
// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult = ec2.requestSpotInstances(requestRequest);
List<SpotInstanceRequest> requestResponses = requestResult.getSpotInstanceRequests();

// A list of request IDs to tag
ArrayList<String> spotInstanceRequestIds = new ArrayList<String>();

// Add the request ids to the hashset, so we can determine when they hit the
// active state.
for (SpotInstanceRequest requestResponse : requestResponses) {
    System.out.println("Created Spot Request:
 "+requestResponse.getSpotInstanceRequestId());
    spotInstanceRequestIds.add(requestResponse.getSpotInstanceRequestId());
}
```

Once you have the IDs, you can tag the requests by adding their IDs to a CreateTagsRequest and calling the EC2 client's `createTags()` method:

```
// The list of tags to create
ArrayList<Tag> requestTags = new ArrayList<Tag>();
requestTags.add(new Tag("keyname1","value1"));

// Create the tag request
CreateTagsRequest createTagsRequest_requests = new CreateTagsRequest();
createTagsRequest_requests.setResources(spotInstanceRequestIds);
createTagsRequest_requests.setTags(requestTags);

// Tag the spot request
try {
    ec2.createTags(createTagsRequest_requests);
}
catch (AmazonServiceException e) {
    System.out.println("Error terminating instances");
    System.out.println("Caught Exception: " + e.getMessage());
    System.out.println("Reponse Status Code: " + e.getStatusCode());
    System.out.println("Error Code: " + e.getErrorCode());
    System.out.println("Request ID: " + e.getRequestId());
}
```

Tagging instances

Similarly to spot requests themselves, you can only tag an instance once it has been created, which will happen once the spot request has been met (it is no longer in the *open* state).

You can check the status of your requests by calling the EC2 client's `describeSpotInstanceRequests()` method with a DescribeSpotInstanceRequestsRequest object. The returned DescribeSpotInstanceRequestsResult object contains a list of SpotInstanceRequest objects that you can use to query the status of your spot requests and obtain their instance IDs once they are no longer in the *open* state.

Once the spot request is no longer open, you can retrieve its instance ID from the `SpotInstanceRequest` object by calling its `getInstanceId()` method.

```
boolean anyOpen; // tracks whether any requests are still open

// a list of instances to tag.
ArrayList<String> instanceIds = new ArrayList<String>();

do {
    DescribeSpotInstanceRequestsRequest describeRequest =
        new DescribeSpotInstanceRequestsRequest();
    describeRequest.setSpotInstanceRequestIds(spotInstanceRequestIds);

    anyOpen=false; // assume no requests are still open

    try {
        // Get the requests to monitor
        DescribeSpotInstanceRequestsResult describeResult =
            ec2.describeSpotInstanceRequests(describeRequest);

        List<SpotInstanceRequest> describeResponses =
            describeResult.getSpotInstanceRequests();

        // are any requests open?
        for (SpotInstanceRequest describeResponse : describeResponses) {
                if (describeResponse.getState().equals("open")) {
                    anyOpen = true;
                    break;
                }
                // get the corresponding instance ID of the spot request
                instanceIds.add(describeResponse.getInstanceId());
            }
        }
        catch (AmazonServiceException e) {
            // Don't break the loop due to an exception (it may be a temporary issue)
            anyOpen = true;
        }

        try {
            Thread.sleep(60*1000); // sleep 60s.
        }
        catch (Exception e) {
            // Do nothing if the thread woke up early.
        }
    } while (anyOpen);
```

Now you can tag the instances that are returned:

```
// Create a list of tags to create
ArrayList<Tag> instanceTags = new ArrayList<Tag>();
instanceTags.add(new Tag("keyname1","value1"));

// Create the tag request
CreateTagsRequest createTagsRequest_instances = new CreateTagsRequest();
createTagsRequest_instances.setResources(instanceIds);
createTagsRequest_instances.setTags(instanceTags);

// Tag the instance
try {
    ec2.createTags(createTagsRequest_instances);
}
catch (AmazonServiceException e) {
    // Write out any exceptions that may have occurred.
    System.out.println("Error terminating instances");
    System.out.println("Caught Exception: " + e.getMessage());
    System.out.println("Reponse Status Code: " + e.getStatusCode());
    System.out.println("Error Code: " + e.getErrorCode());
    System.out.println("Request ID: " + e.getRequestId());
}
```

Canceling spot requests and terminating instances

Canceling a spot request

To cancel a spot instance request, call `cancelSpotInstanceRequests` on the EC2 client with a CancelSpotInstanceRequestsRequest object.

```
try {
    CancelSpotInstanceRequestsRequest cancelRequest = new
 CancelSpotInstanceRequestsRequest(spotInstanceRequestIds);
    ec2.cancelSpotInstanceRequests(cancelRequest);
} catch (AmazonServiceException e) {
    System.out.println("Error cancelling instances");
    System.out.println("Caught Exception: " + e.getMessage());
    System.out.println("Reponse Status Code: " + e.getStatusCode());
    System.out.println("Error Code: " + e.getErrorCode());
    System.out.println("Request ID: " + e.getRequestId());
}
```

Terminating spot instances

You can terminate any spot instances that are running by passing their IDs to the EC2 client's `terminateInstances()` method.

```
try {
    TerminateInstancesRequest terminateRequest = new
 TerminateInstancesRequest(instanceIds);
    ec2.terminateInstances(terminateRequest);
} catch (AmazonServiceException e) {
    System.out.println("Error terminating instances");
    System.out.println("Caught Exception: " + e.getMessage());
    System.out.println("Reponse Status Code: " + e.getStatusCode());
    System.out.println("Error Code: " + e.getErrorCode());
    System.out.println("Request ID: " + e.getRequestId());
}
```

Bringing it all together

To bring this all together, we provide a more object-oriented approach that combines the steps we showed in this tutorial into one easy to use class. We instantiate a class called `Requests` that performs these actions. We also create a `GettingStartedApp` class, which has a main method where we perform the high level function calls.

The complete source code for this example can be viewed or downloaded at GitHub.

Congratulations! You've completed the Advanced Request Features tutorial for developing Spot Instance software with the AWS SDK for Java.

Managing Amazon EC2 Instances

Creating an Instance

Create a new Amazon EC2 instance by calling the AmazonEC2Client's `runInstances` method, providing it with a RunInstancesRequest containing the Amazon Machine Image (AMI) to use and an instance type.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.InstanceType;
import com.amazonaws.services.ec2.model.RunInstancesRequest;
import com.amazonaws.services.ec2.model.RunInstancesResult;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

RunInstancesRequest run_request = new RunInstancesRequest()
    .withImageId(ami_id)
    .withInstanceType(InstanceType.T1Micro)
    .withMaxCount(1)
    .withMinCount(1);

RunInstancesResult run_response = ec2.runInstances(run_request);
```

```
String instance_id = run_response.getReservation().getReservationId();
```

See the complete example.

Starting an Instance

To start an Amazon EC2 instance, call the AmazonEC2Client's `startInstances` method, providing it with a StartInstancesRequest containing the ID of the instance to start.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.StartInstancesRequest;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

StartInstancesRequest request = new StartInstancesRequest()
    .withInstanceIds(instance_id);

ec2.startInstances(request);
```

See the complete example.

Stopping an Instance

To stop an Amazon EC2 instance, call the AmazonEC2Client's `stopInstances` method, providing it with a StopInstancesRequest containing the ID of the instance to stop.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.StopInstancesRequest;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

StopInstancesRequest request = new StopInstancesRequest()
    .withInstanceIds(instance_id);

ec2.stopInstances(request);
```

See the complete example.

Rebooting an Instance

To reboot an Amazon EC2 instance, call the AmazonEC2Client's `rebootInstances` method, providing it with a RebootInstancesRequest containing the ID of the instance to reboot.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.RebootInstancesRequest;
import com.amazonaws.services.ec2.model.RebootInstancesResult;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

RebootInstancesRequest request = new RebootInstancesRequest()
    .withInstanceIds(instance_id);

RebootInstancesResult response = ec2.rebootInstances(request);
```

See the complete example.

Describing Instances

To list your instances, create a DescribeInstancesRequest and call the AmazonEC2Client's `describeInstances` method. It will return a DescribeInstancesResult object that you can use to list the Amazon EC2 instances for your account and region.

Instances are grouped by *reservation*. Each reservation corresponds to the call to `startInstances` that launched the instance. To list your instances, you must first call the `DescribeInstancesResult` class' `getReservations' method, and then call :methodname:`getInstances` on each returned Reservation object.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.DescribeInstancesRequest;
import com.amazonaws.services.ec2.model.DescribeInstancesResult;
import com.amazonaws.services.ec2.model.Instance;
import com.amazonaws.services.ec2.model.Reservation;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();
boolean done = false;

while(!done) {
    DescribeInstancesRequest request = new DescribeInstancesRequest();
    DescribeInstancesResult response = ec2.describeInstances(request);

    for(Reservation reservation : response.getReservations()) {
        for(Instance instance : reservation.getInstances()) {
            System.out.printf(
                "Found reservation with id %s, " +
                "AMI %s, " +
                "type %s, " +
                "state %s " +
                "and monitoring state %s",
                instance.getInstanceId(),
                instance.getImageId(),
                instance.getInstanceType(),
                instance.getState().getName(),
                instance.getMonitoring().getState());
        }
    }

    request.setNextToken(response.getNextToken());

    if(response.getNextToken() == null) {
        done = true;
    }
}
```

Results are paged; you can get further results by passing the value returned from the result object's `getNextToken` method to your original request object's `setNextToken` method, then using the same request object in your next call to `describeInstances`.

See the complete example.

Monitoring an Instance

You can monitor various aspects of your Amazon EC2 instances, such as CPU and network utilization, available memory, and disk space remaining. To learn more about instance monitoring, see Monitoring Amazon EC2 in the *Amazon EC2 User Guide for Linux Instances*.

To start monitoring an instance, you must create a MonitorInstancesRequest with the ID of the instance to monitor, and pass it to the AmazonEC2Client's `monitorInstances` method.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.MonitorInstancesRequest;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

MonitorInstancesRequest request = new MonitorInstancesRequest()
        .withInstanceIds(instance_id);

ec2.monitorInstances(request);
```

See the complete example.

Stopping Instance Monitoring

To stop monitoring an instance, create an UnmonitorInstancesRequest with the ID of the instance to stop monitoring, and pass it to the AmazonEC2Client's `unmonitorInstances` method.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.UnmonitorInstancesRequest;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

UnmonitorInstancesRequest request = new UnmonitorInstancesRequest()
```

```
    .withInstanceIds(instance_id);

ec2.unmonitorInstances(request);
```

See the complete example.

More Information

- RunInstances in the *Amazon EC2 API Reference*
- DescribeInstances in the *Amazon EC2 API Reference*
- StartInstances in the *Amazon EC2 API Reference*
- StopInstances in the *Amazon EC2 API Reference*
- RebootInstances in the *Amazon EC2 API Reference*
- DescribeInstances in the *Amazon EC2 API Reference*
- MonitorInstances in the *Amazon EC2 API Reference*
- UnmonitorInstances in the *Amazon EC2 API Reference*

Using Elastic IP Addresses in Amazon EC2

Allocating an Elastic IP Address

To use an Elastic IP address, you first allocate one to your account, and then associate it with your instance or a network interface.

To allocate an Elastic IP address, call the AmazonEC2Client's `allocateAddress` method with an AllocateAddressRequest object containing the network type (classic EC2 or VPC).

The returned AllocateAddressResult contains an allocation ID that you can use to associate the address with an instance, by passing the allocation ID and instance ID in a AssociateAddressRequest to the AmazonEC2Client's `associateAddress` method.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.AllocateAddressRequest;
import com.amazonaws.services.ec2.model.AllocateAddressResult;
import com.amazonaws.services.ec2.model.AssociateAddressRequest;
import com.amazonaws.services.ec2.model.AssociateAddressResult;
import com.amazonaws.services.ec2.model.DomainType;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

AllocateAddressRequest allocate_request = new AllocateAddressRequest()
    .withDomain(DomainType.Vpc);

AllocateAddressResult allocate_response =
    ec2.allocateAddress(allocate_request);

String allocation_id = allocate_response.getAllocationId();

AssociateAddressRequest associate_request =
    new AssociateAddressRequest()
```

```
        .withInstanceId(instance_id)
        .withAllocationId(allocation_id);

AssociateAddressResult associate_response =
    ec2.associateAddress(associate_request);
```

See the complete example.

Describing Elastic IP Addresses

To list the Elastic IP addresses assigned to your account, call the AmazonEC2Client's `describeAddresses` method. It returns a DescribeAddressesResult which you can use to get a list of Address objects that represent the Elastic IP addresses on your account.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.Address;
import com.amazonaws.services.ec2.model.DescribeAddressesResult;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

DescribeAddressesResult response = ec2.describeAddresses();

for(Address address : response.getAddresses()) {
    System.out.printf(
            "Found address with public IP %s, " +
            "domain %s, " +
            "allocation id %s " +
            "and NIC id %s",
            address.getPublicIp(),
            address.getDomain(),
            address.getAllocationId(),
            address.getNetworkInterfaceId());
}
```

See the complete example.

Releasing an Elastic IP Address

To release an Elastic IP address, call the AmazonEC2Client's `releaseAddress` method, passing it a ReleaseAddressRequest containing the allocation ID of the Elastic IP address you want to release.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.ReleaseAddressRequest;
import com.amazonaws.services.ec2.model.ReleaseAddressResult;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

ReleaseAddressRequest request = new ReleaseAddressRequest()
```

```
    .withAllocationId(alloc_id);

ReleaseAddressResult response = ec2.releaseAddress(request);
```

After you release an Elastic IP address, it is released to the AWS IP address pool and might be unavailable to you afterward. Be sure to update your DNS records and any servers or devices that communicate with the address. If you attempt to release an Elastic IP address that you already released, you'll get an *AuthFailure* error if the address is already allocated to another AWS account.

If you are using *EC2-Classic* or a *default VPC*, then releasing an Elastic IP address automatically disassociates it from any instance that it's associated with. To disassociate an Elastic IP address without releasing it, use the AmazonEC2Client's `disassociateAddress` method.

If you are using a non-default VPC, you *must* use `disassociateAddress` to disassociate the Elastic IP address before you try to release it. Otherwise, Amazon EC2 returns an error (*InvalidIPAddress.InUse*).

See the complete example.

More Information

- Elastic IP Addresses in the *Amazon EC2 User Guide for Linux Instances*
- AllocateAddress in the *Amazon EC2 API Reference*
- DescribeAddresses in the *Amazon EC2 API Reference*
- ReleaseAddress in the *Amazon EC2 API Reference*

Using Regions and Availability Zones

Describing Regions

To list the regions available to your account, call the AmazonEC2Client's `describeRegions` method. It returns a DescribeRegionsResult. Call the returned object's `getRegions` method to get a list of Region objects that represent each region.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.DescribeRegionsResult;
import com.amazonaws.services.ec2.model.Region;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

DescribeRegionsResult regions_response = ec2.describeRegions();

for(Region region : regions_response.getRegions()) {
    System.out.printf(
        "Found region %s " +
        "with endpoint %s",
        region.getRegionName(),
        region.getEndpoint());
}
```

See the complete example.

Describing Availability Zones

To list each availability zone available to your account, call the AmazonEC2Client's `describeAvailabilityZones` method. It returns a DescribeAvailabilityZonesResult. Call its `getAvailabilityZones` method to get a list of AvailabilityZone objects that represent each availability zone.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.DescribeRegionsResult;
import com.amazonaws.services.ec2.model.Region;
```

Code

```
DescribeAvailabilityZonesResult zones_response =
    ec2.describeAvailabilityZones();

for(AvailabilityZone zone : zones_response.getAvailabilityZones()) {
    System.out.printf(
        "Found availability zone %s " +
        "with status %s " +
        "in region %s",
        zone.getZoneName(),
        zone.getState(),
        zone.getRegionName());
}
```

See the complete example.

More Information

- Regions and Availability Zones in the *Amazon EC2 User Guide for Linux Instances*
- DescribeRegions in the *Amazon EC2 API Reference*
- DescribeAvailabilityZones in the *Amazon EC2 API Reference*

Working with Amazon EC2 Key Pairs

Creating a Key Pair

To create a key pair, call the AmazonEC2Client's `createKeyPair` method with a CreateKeyPairRequest that contains the key's name.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.CreateKeyPairRequest;
import com.amazonaws.services.ec2.model.CreateKeyPairResult;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();
```

```
CreateKeyPairRequest request = new CreateKeyPairRequest()
    .withKeyName(key_name);

CreateKeyPairResult response = ec2.createKeyPair(request);
```

See the complete example.

Describing Key Pairs

To list your key pairs or to get information about them, call the AmazonEC2Client's `describeKeyPairs` method. It returns a DescribeKeyPairsRequest that you can use to access the list of key pairs by calling its `getKeyPairs` method, which returns a list of KeyPairInfo objects.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.DescribeKeyPairsResult;
import com.amazonaws.services.ec2.model.KeyPairInfo;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

DescribeKeyPairsResult response = ec2.describeKeyPairs();

for(KeyPairInfo key_pair : response.getKeyPairs()) {
    System.out.printf(
        "Found key pair with name %s " +
        "and fingerprint %s",
        key_pair.getKeyName(),
        key_pair.getKeyFingerprint());
}
```

See the complete example.

Deleting a Key Pair

To delete a key pair, call the AmazonEC2Client's `deleteKeyPair` method, passing it a DeleteKeyPairRequest that contains the name of the key pair to delete.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.DeleteKeyPairRequest;
import com.amazonaws.services.ec2.model.DeleteKeyPairResult;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

DeleteKeyPairRequest request = new DeleteKeyPairRequest()
    .withKeyName(key_name);

DeleteKeyPairResult response = ec2.deleteKeyPair(request);
```

See the complete example.

More Information

- Amazon EC2 Key Pairs in the *Amazon EC2 User Guide for Linux Instances*
- CreateKeyPair in the *Amazon EC2 API Reference*
- DescribeKeyPairs in the *Amazon EC2 API Reference*
- DeleteKeyPair in the *Amazon EC2 API Reference*

Working with Security Groups in Amazon EC2

Creating a Security Group

To create a security group, call the AmazonEC2Client's `createSecurityGroup` method with a CreateSecurityGroupRequest that contains the key's name.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.CreateSecurityGroupRequest;
import com.amazonaws.services.ec2.model.CreateSecurityGroupResult;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

CreateSecurityGroupRequest create_request = new
    CreateSecurityGroupRequest()
        .withGroupName(group_name)
        .withDescription(group_desc)
        .withVpcId(vpc_id);

CreateSecurityGroupResult create_response =
    ec2.createSecurityGroup(create_request);
```

See the complete example.

Configuring a Security Group

A security group can control both inbound (ingress) and outbound (egress) traffic to your Amazon EC2 instances.

To add ingress rules to your security group, use the AmazonEC2Client's `authorizeSecurityGroupIngress` method, providing the name of the security group and the access rules (IpPermission) you want to assign to it within an AuthorizeSecurityGroupIngressRequest object. The following example shows how to add IP permissions to a security group.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.CreateSecurityGroupRequest;
import com.amazonaws.services.ec2.model.CreateSecurityGroupResult;
```

Code

```
IpRange ip_range = new IpRange()
    .withCidrIp("0.0.0.0/0");

IpPermission ip_perm = new IpPermission()
    .withIpProtocol("tcp")
    .withToPort(80)
    .withFromPort(80)
    .withIpv4Ranges(ip_range);

IpPermission ip_perm2 = new IpPermission()
    .withIpProtocol("tcp")
    .withToPort(22)
    .withFromPort(22)
    .withIpv4Ranges(ip_range);

AuthorizeSecurityGroupIngressRequest auth_request = new
    AuthorizeSecurityGroupIngressRequest()
        .withGroupName(group_name)
        .withIpPermissions(ip_perm, ip_perm2);

AuthorizeSecurityGroupIngressResult auth_response =
    ec2.authorizeSecurityGroupIngress(auth_request);
```

To add an egress rule to the security group, provide similar data in an AuthorizeSecurityGroupEgressRequest to the AmazonEC2Client's `authorizeSecurityGroupEgress` method.

See the complete example.

Describing Security Groups

To describe your security groups or get information about them, call the AmazonEC2Client's `describeSecurityGroups` method. It returns a DescribeSecurityGroupsResult that you can use to access the list of security groups by calling its `getSecurityGroups` method, which returns a list of SecurityGroupInfo objects.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.DescribeSecurityGroupsRequest;
import com.amazonaws.services.ec2.model.DescribeSecurityGroupsResult;
```

Code

```

final String USAGE =
    "To run this example, supply a group id\n" +
    "Ex: DescribeSecurityGroups <group-id>\n";

if (args.length != 1) {
    System.out.println(USAGE);
    System.exit(1);
}

String group_id = args[0];
```

See the complete example.

Deleting a Security Group

To delete a security group, call the AmazonEC2Client's `deleteSecurityGroup` method, passing it a DeleteSecurityGroupRequest that contains the ID of the security group to delete.

Imports

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.DeleteSecurityGroupRequest;
import com.amazonaws.services.ec2.model.DeleteSecurityGroupResult;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

DeleteSecurityGroupRequest request = new DeleteSecurityGroupRequest()
    .withGroupId(group_id);

DeleteSecurityGroupResult response = ec2.deleteSecurityGroup(request);
```

See the complete example.

More Information

- Amazon EC2 Security Groups in the *Amazon EC2 User Guide for Linux Instances*
- Authorizing Inbound Traffic for Your Linux Instances in the *Amazon EC2 User Guide for Linux Instances*
- CreateSecurityGroup in the *Amazon EC2 API Reference*
- DescribeSecurityGroups in the *Amazon EC2 API Reference*
- DeleteSecurityGroup in the *Amazon EC2 API Reference*
- AuthorizeSecurityGroupIngress in the *Amazon EC2 API Reference*

IAM Examples

This section provides examples of programming IAM using the AWS SDK for Java.

AWS Identity and Access Management (IAM) enables you to securely control access to AWS services and resources for your users. Using IAM, you can create and manage AWS users and groups, and use permissions to allow and deny their access to AWS resources. For a complete guide to IAM, visit the IAM User Guide.

Note
The examples include only the code needed to demonstrate each technique. The complete example code is available on GitHub. From there, you can download a single source file or clone the repository locally to get all the examples to build and run.

Topics

Managing IAM Access Keys

Creating an Access Key

To create an IAM access key, call the AmazonIdentityManagementClient`createAccessKey` method with an CreateAccessKeyRequest object.

`CreateAccessKeyRequest` has two constructors — one that takes a user name and another with no parameters. If you use the version that takes no parameters, you must set the user name using the `withUserName` setter method before passing it to the `createAccessKey` method.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.CreateAccessKeyRequest;
import com.amazonaws.services.identitymanagement.model.CreateAccessKeyResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

CreateAccessKeyRequest request = new CreateAccessKeyRequest()
    .withUserName(user);

CreateAccessKeyResult response = iam.createAccessKey(request);
```

See the complete example.

Listing Access Keys

To list the access keys for a given user, create a ListAccessKeysRequest object that contains the user name to list keys for, and pass it to the AmazonIdentityManagementClient's `listAccessKeys` method.

> **Note**
> If you do not supply a user name to `listAccessKeys`, it will attempt to list access keys associated with the AWS account that signed the request.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.AccessKeyMetadata;
import com.amazonaws.services.identitymanagement.model.ListAccessKeysRequest;
import com.amazonaws.services.identitymanagement.model.ListAccessKeysResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

boolean done = false;

while (!done) {
    ListAccessKeysRequest request = new ListAccessKeysRequest()
```

```
        .withUserName(username);

    ListAccessKeysResult response = iam.listAccessKeys(request);

    for (AccessKeyMetadata metadata :
            response.getAccessKeyMetadata()) {
        System.out.format("Retrieved access key %s",
                metadata.getAccessKeyId());
    }

    request.setMarker(response.getMarker());

    if (!response.getIsTruncated()) {
        done = true;
    }
}
```

The results of `listAccessKeys` are paged (with a default maximum of 100 records per call). You can call `getIsTruncated` on the returned ListAccessKeysResult object to see if the query returned fewer results then are available. If so, then call `setMarker` on the `ListAccessKeysRequest` and pass it back to the next invocation of `listAccessKeys`.

See the complete example.

Retrieving an Access Key's Last Used Time

To get the time an access key was last used, call the AmazonIdentityManagementClient's `getAccessKeyLastUsed` method with the access key's ID (which can be passed in using a GetAccessKeyLastUsedRequest object, or directly to the overload that takes the access key ID directly.

You can then use the returned GetAccessKeyLastUsedResult object to retrieve the key's last used time.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.GetAccessKeyLastUsedRequest;
import com.amazonaws.services.identitymanagement.model.GetAccessKeyLastUsedResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

GetAccessKeyLastUsedRequest request = new GetAccessKeyLastUsedRequest()
    .withAccessKeyId(access_id);

GetAccessKeyLastUsedResult response = iam.getAccessKeyLastUsed(request);

System.out.println("Access key was last used at: " +
        response.getAccessKeyLastUsed().getLastUsedDate());
```

See the complete example.

Activating or Deactivating Access Keys

You can activate or deactivate an access key by creating an UpdateAccessKeyRequest object, providing the access key ID, optionally the user name, and the desired Status, then passing the request object to the AmazonIdentityManagementClient's `updateAccessKey` method.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.UpdateAccessKeyRequest;
import com.amazonaws.services.identitymanagement.model.UpdateAccessKeyResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

UpdateAccessKeyRequest request = new UpdateAccessKeyRequest()
    .withAccessKeyId(access_id)
    .withUserName(username)
    .withStatus(status);

UpdateAccessKeyResult response = iam.updateAccessKey(request);
```

See the complete example.

Deleting an Access Key

To permanently delete an access key, call the AmazonIdentityManagementClient's `deleteKey` method, providing it with a DeleteAccessKeyRequest containing the access key's ID and username.

> **Note**
> Once deleted, a key can no longer be retrieved or used. To temporarily deactivate a key so that it can be activated again later, use updateAccessKey (p. 94) method instead.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.DeleteAccessKeyRequest;
import com.amazonaws.services.identitymanagement.model.DeleteAccessKeyResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

DeleteAccessKeyRequest request = new DeleteAccessKeyRequest()
    .withAccessKeyId(access_key)
    .withUserName(username);

DeleteAccessKeyResult response = iam.deleteAccessKey(request);
```

See the complete example.

More Information

- CreateAccessKey in the *IAM API Reference*
- ListAccessKeys in the *IAM API Reference*
- GetAccessKeyLastUsed in the *IAM API Reference*
- UpdateAccessKey in the *IAM API Reference*
- DeleteAccessKey in the *IAM API Reference*

Managing IAM Users

Creating a User

Create a new IAM user by providing the user name to the AmazonIdentityManagementClient's `createUser` method, either directly or using a CreateUserRequest object containing the user name.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.CreateUserRequest;
import com.amazonaws.services.identitymanagement.model.CreateUserResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

CreateUserRequest request = new CreateUserRequest()
    .withUserName(username);

CreateUserResult response = iam.createUser(request);
```

See the complete example.

Listing Users

To list the IAM users for your account, create a new ListUsersRequest and pass it to the AmazonIdentityManagementClient's `listUsers` method. You can retrieve the list of users by calling `getUsers` on the returned ListUsersResponse object.

The list of users returned by `listUsers` is paged. You can check to see there are more results to retrieve by calling the response object's `getIsTruncated` method. If it returns `true`, then call the request object's `setMarker()` method, passing it the return value of the response object's `getMarker()` method.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.ListUsersRequest;
import com.amazonaws.services.identitymanagement.model.ListUsersResult;
import com.amazonaws.services.identitymanagement.model.User;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

boolean done = false;

while(!done) {
    ListUsersRequest request = new ListUsersRequest();
    ListUsersResult response = iam.listUsers(request);

    for(User user : response.getUsers()) {
        System.out.format("Retrieved user %s", user.getUserName());
    }
    request.setMarker(response.getMarker());

    if(!response.getIsTruncated()) {
        done = true;
    }
}
```

See the complete example.

Updating a User

To update a user, call the AmazonIdentityManagementClient object's `updateUser` method, which takes a UpdateUserRequest object that you can use to change the user's *name* or *path*.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.UpdateUserRequest;
import com.amazonaws.services.identitymanagement.model.UpdateUserResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

UpdateUserRequest request = new UpdateUserRequest()
    .withUserName(cur_name)
    .withNewUserName(new_name);

UpdateUserResult response = iam.updateUser(request);
```

See the complete example.

Deleting a User

To delete a user, call the AmazonIdentityManagementClient's `deleteUser` request with a UpdateUserRequest object set with the user name to delete.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.DeleteConflictException;
import com.amazonaws.services.identitymanagement.model.DeleteUserRequest;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

DeleteUserRequest request = new DeleteUserRequest()
    .withUserName(username);

try {
    iam.deleteUser(request);
} catch (DeleteConflictException e) {
    System.out.println("Unable to delete user. Verify user is not" +
            " associated with any resources");
    throw e;
}
```

See the complete example.

More Information

- IAM Users in the *IAM User Guide*
- Managing IAM Users in the *IAM User Guide*
- CreateUser in the *IAM API Reference*
- ListUsers in the *IAM API Reference*
- UpdateUser in the *IAM API Reference*
- DeleteUser in the *IAM API Reference*

Using IAM Account Aliases

If you want the URL for your sign-in page to contain your company name or other friendly identifier instead of your AWS account ID, you can create an alias for your AWS account.

Note
AWS supports exactly one account alias per account.

Creating an Account Alias

To create an account alias, call the AmazonIdentityManagementClient's `createAccountAlias` method with a CreateAccountAliasRequest object that contains the alias name.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.CreateAccountAliasRequest;
import com.amazonaws.services.identitymanagement.model.CreateAccountAliasResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

CreateAccountAliasRequest request = new CreateAccountAliasRequest()
    .withAccountAlias(alias);

CreateAccountAliasResult response = iam.createAccountAlias(request);
```

See the complete example.

Listing Account Aliases

To list your account's alias, if any, call the AmazonIdentityManagementClient's `listAccountAliases` method.

Note
The returned ListAccountAliasesResponse supports the same `getIsTruncated` and `getMarker` methods as other AWS SDK for Java*list* methods, but an AWS account can have only *one* account alias.

imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.ListAccountAliasesResult;
```

code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

ListAccountAliasesResult response = iam.listAccountAliases();

for (String alias : response.getAccountAliases()) {
    System.out.printf("Retrieved account alias %s", alias);
}
```

see the complete example.

Deleting an account alias

To delete your account's alias, call the AmazonIdentityManagementClient's `deleteAccountAlias` method. When deleting an account alias, you must supply its name using a DeleteAccountAliasRequest object.

imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.DeleteAccountAliasRequest;
import com.amazonaws.services.identitymanagement.model.DeleteAccountAliasResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

DeleteAccountAliasRequest request = new DeleteAccountAliasRequest()
    .withAccountAlias(alias);

DeleteAccountAliasResult response = iam.deleteAccountAlias(request);
```

See the complete example.

More Information

- Your AWS Account ID and Its Alias in the *IAM User Guide*
- CreateAccountAlias in the *IAM API Reference*
- ListAccountAliases in the *IAM API Reference*
- DeleteAccountAlias in the *IAM API Reference*

Working with IAM Policies

Creating a Policy

To create a new policy, provide the policy's name and a JSON-formatted policy document in a CreatePolicyRequest to the AmazonIdentityManagementClient's `createPolicy` method.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.CreatePolicyRequest;
import com.amazonaws.services.identitymanagement.model.CreatePolicyResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

CreatePolicyRequest request = new CreatePolicyRequest()
    .withPolicyName(policy_name)
    .withPolicyDocument(POLICY_DOCUMENT);

CreatePolicyResult response = iam.createPolicy(request);
```

IAM policy documents are JSON strings with a well-documented syntax. Here is an example that provides access to make particular requests to DynamoDB.

```
public static final String POLICY_DOCUMENT =
    "{" +
    "  \"Version\": \"2012-10-17\"," +
    "  \"Statement\": [" +
    "    {" +
    "        \"Effect\": \"Allow\"," +
    "        \"Action\": \"logs:CreateLogGroup\"," +
    "        \"Resource\": \"%s\"" +
    "    }," +
    "    {" +
    "        \"Effect\": \"Allow\"," +
    "        \"Action\": [" +
    "            \"dynamodb:DeleteItem\"," +
    "            \"dynamodb:GetItem\"," +
    "            \"dynamodb:PutItem\"," +
    "            \"dynamodb:Scan\"," +
    "            \"dynamodb:UpdateItem\"" +
    "       ]," +
    "       \"Resource\": \"RESOURCE_ARN\"" +
    "    }" +
    "   ]" +
    "}";
```

See the complete example.

Getting a Policy

To retrieve an existing policy, call the AmazonIdentityManagementClient's `getPolicy` method, providing the policy's ARN within a GetPolicyRequest object.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.GetPolicyRequest;
import com.amazonaws.services.identitymanagement.model.GetPolicyResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

GetPolicyRequest request = new GetPolicyRequest()
    .withPolicyArn(policy_arn);

GetPolicyResult response = iam.getPolicy(request);
```

See the complete example.

Attaching a Role Policy

You can attach a policy to an IAMrole by calling the AmazonIdentityManagementClient's `attachRolePolicy` method, providing it with the role name and policy ARN in an AttachRolePolicyRequest.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.AttachRolePolicyRequest;
import com.amazonaws.services.identitymanagement.model.AttachedPolicy;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

AttachRolePolicyRequest attach_request =
    new AttachRolePolicyRequest()
        .withRoleName(role_name)
        .withPolicyArn(POLICY_ARN);

iam.attachRolePolicy(attach_request);
```

See the complete example.

Listing Attached Role Policies

List attached policies on a role by calling the AmazonIdentityManagementClient's `listAttachedRolePolicies` method. It takes a ListAttachedRolePoliciesRequest object that contains the role name to list the policies for.

Call `getAttachedPolicies` on the returned ListAttachedRolePoliciesResult object to get the list of attached policies. Results may be truncated; if the `ListAttachedRolePoliciesResult` object's `getIsTruncated` method returns `true`, call the `ListAttachedRolePoliciesRequest` object's `setMarker` method and use it to call `listAttachedRolePolicies` again to get the next batch of results.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.ListAttachedRolePoliciesRequest;
import com.amazonaws.services.identitymanagement.model.ListAttachedRolePoliciesResult;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

ListAttachedRolePoliciesRequest request =
    new ListAttachedRolePoliciesRequest()
        .withRoleName(role_name);

List<AttachedPolicy> matching_policies = new ArrayList<>();

boolean done = false;

while(!done) {
    ListAttachedRolePoliciesResult response =
        iam.listAttachedRolePolicies(request);

    matching_policies.addAll(
            response.getAttachedPolicies()
                    .stream()
                    .filter(p -> p.getPolicyName().equals(role_name))
                    .collect(Collectors.toList()));

    if(!response.getIsTruncated()) {
        done = true;
    }
    request.setMarker(response.getMarker());
}
```

See the complete example.

Detaching a Role Policy

To detach a policy from a role, call the AmazonIdentityManagementClient's `detachRolePolicy` method, providing it with the role name and policy ARN in a DetachRolePolicyRequest.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.DetachRolePolicyRequest;
import com.amazonaws.services.identitymanagement.model.DetachRolePolicyResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

DetachRolePolicyRequest request = new DetachRolePolicyRequest()
    .withRoleName(role_name)
    .withPolicyArn(policy_arn);

DetachRolePolicyResult response = iam.detachRolePolicy(request);
```

See the complete example.

More Information

- Overview of IAM Policies in the *IAM User Guide*.

- AWS IAM Policy Reference in the *IAM User Guide*.
- CreatePolicy in the *IAM API Reference*
- GetPolicy in the *IAM API Reference*
- AttachRolePolicy in the *IAM API Reference*
- ListAttachedRolePolicies in the *IAM API Reference*
- DetachRolePolicy in the *IAM API Reference*

Working with IAM Server Certificates

To enable HTTPS connections to your website or application on AWS, you need an SSL/TLS *server certificate*. You can use a server certificate provided by AWS Certificate Manager or one that you obtained from an external provider.

We recommend that you use ACM to provision, manage, and deploy your server certificates. With ACM you can request a certificate, deploy it to your AWS resources, and let ACM handle certificate renewals for you. Certificates provided by ACM are free. For more information about ACM , see the ACM User Guide.

Getting a Server Certificate

You can retrieve a server certificate by calling the AmazonIdentityManagementClient's `getServerCertificate` method, passing it a GetServerCertificateRequest with the certificate's name.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.GetServerCertificateRequest;
import com.amazonaws.services.identitymanagement.model.GetServerCertificateResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

GetServerCertificateRequest request = new GetServerCertificateRequest()
            .withServerCertificateName(cert_name);

GetServerCertificateResult response = iam.getServerCertificate(request);
```

See the complete example.

Listing Server Certificates

To list your server certificates, call the AmazonIdentityManagementClient's `listServerCertificates` method with a ListServerCertificatesRequest. It returns a ListServerCertificatesResult.

Call the returned `ListServerCertificateResult` object's `getServerCertificateMetadataList` method to get a list of ServerCertificateMetadata objects that you can use to get information about each certificate.

Results may be truncated; if the `ListServerCertificateResult` object's `getIsTruncated` method returns `true`, call the `ListServerCertificatesRequest` object's `setMarker` method and use it to call `listServerCertificates` again to get the next batch of results.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.ListServerCertificatesRequest;
import com.amazonaws.services.identitymanagement.model.ListServerCertificatesResult;
import com.amazonaws.services.identitymanagement.model.ServerCertificateMetadata;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

boolean done = false;

while(!done) {
    ListServerCertificatesRequest request =
        new ListServerCertificatesRequest();

    ListServerCertificatesResult response =
        iam.listServerCertificates(request);

    for(ServerCertificateMetadata metadata :
            response.getServerCertificateMetadataList()) {
        System.out.printf("Retrieved server certificate %s",
                metadata.getServerCertificateName());
    }

    request.setMarker(response.getMarker());

    if(!response.getIsTruncated()) {
        done = true;
    }
}
```

See the complete example.

Updating a Server Certificate

You can update a server certificate's name or path by calling the AmazonIdentityManagementClient's `updateServerCertificate` method. It takes a UpdateServerCertificateRequest object set with the server certificate's current name and either a new name or new path to use.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.UpdateServerCertificateRequest;
import com.amazonaws.services.identitymanagement.model.UpdateServerCertificateResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

UpdateServerCertificateRequest request =
    new UpdateServerCertificateRequest()
        .withServerCertificateName(cur_name)
        .withNewServerCertificateName(new_name);
```

```
UpdateServerCertificateResult response =
    iam.updateServerCertificate(request);
```

See the complete example.

Deleting a Server Certificate

To delete a server certificate, call the AmazonIdentityManagementClient's `deleteServerCertificate` method with a DeleteServerCertificateRequest containing the certificate's name.

Imports

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.DeleteServerCertificateRequest;
import com.amazonaws.services.identitymanagement.model.DeleteServerCertificateResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

DeleteServerCertificateRequest request =
    new DeleteServerCertificateRequest()
        .withServerCertificateName(cert_name);

DeleteServerCertificateResult response =
    iam.deleteServerCertificate(request);
```

See the complete example.

More Information

- Working with Server Certificates in the *IAM User Guide*
- GetServerCertificate in the *IAM API Reference*
- ListServerCertificates in the *IAM API Reference*
- UpdateServerCertificate in the *IAM API Reference*
- DeleteServerCertificate in the *IAM API Reference*
- ACM User Guide

Amazon S3 Examples

This section provides examples of programming Amazon S3 using the AWS SDK for Java.

Note
The examples include only the code needed to demonstrate each technique. The complete example code is available on GitHub. From there, you can download a single source file or clone the repository locally to get all the examples to build and run.

Topics

- Managing Access to Amazon S3 Buckets Using Bucket Policies (p. 116)
- Using TransferManager for Amazon S3 Operations (p. 118)
- Configuring an Amazon S3 Bucket as a Website (p. 125)

Creating, Listing, and Deleting Amazon S3 Buckets

Every object (file) in Amazon S3 must reside within a *bucket*, which represents a collection (container) of objects. Each bucket is known by a *key* (name), which must be unique. For detailed information about buckets and their configuration, see Working with Amazon S3 Buckets in the *Amazon S3 Developer Guide*.
<admonition>
<title>Best Practice</title>

We recommend that you enable the AbortIncompleteMultipartUpload lifecycle rule on your Amazon S3 buckets.

This rule directs Amazon S3 to abort multipart uploads that don't complete within a specified number of days after being initiated. When the set time limit is exceeded, Amazon S3 aborts the upload and then deletes the incomplete upload data.

For more information, see Lifecycle Configuration for a Bucket with Versioning in the *Amazon S3 User Guide*.
</admonition>

Note
These code snippets assume that you understand the material in Using the AWS SDK for Java (p. 11) and have configured default AWS credentials using the information in Set up AWS Credentials and Region for Development (p. 6).

Create a Bucket

Use the AmazonS3 client's `createBucket` method. The new Bucket is returned. The `createBucket` method will raise an exception if the bucket already exists.

Note
To check whether a bucket already exists before attempting to create one with the same name, call the `doesBucketExist` method. It will return `true` if the bucket exists, and `false` otherwise.

Imports

```
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.model.AmazonS3Exception;
import com.amazonaws.services.s3.model.Bucket;
import java.util.List;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
Bucket b = null;
if (s3.doesBucketExist(bucket_name)) {
    System.out.format("Bucket %s already exists.\n", bucket_name);
    b = getBucket(bucket_name);
} else {
    try {
        b = s3.createBucket(bucket_name);
    } catch (AmazonS3Exception e) {
        System.err.println(e.getErrorMessage());
    }
}
return b;
```

See the complete example.

List Buckets

Use the AmazonS3 client's `listBucket` method. If successful, a list of Bucket is returned.

Imports

```
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.Bucket;
import java.util.List;
```

Code

```
inal AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
List<Bucket> buckets = s3.listBuckets();
System.out.println("Your Amazon S3 buckets are:");
for (Bucket b : buckets) {
    System.out.println("* " + b.getName());
}
```

See the complete example.

Delete a Bucket

Before you can delete an Amazon S3 bucket, you must ensure that the bucket is empty or an error will result. If you have a versioned bucket, you must also delete any versioned objects associated with the bucket.

> **Note**
> The complete example includes each of these steps in order, providing a complete solution for deleting an Amazon S3 bucket and its contents.

Topics

Remove Objects from an Unversioned Bucket Before Deleting It

Use the AmazonS3 client's `listObjects` method to retrieve the list of objects and `deleteObject` to delete each one.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.ObjectListing;
import com.amazonaws.services.s3.model.S3ObjectSummary;
import java.util.Iterator;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
    System.out.println(" - removing objects from bucket");
    ObjectListing object_listing = s3.listObjects(bucket_name);
    while (true) {
        for (Iterator<?> iterator =
                object_listing.getObjectSummaries().iterator();
                iterator.hasNext();) {
            S3ObjectSummary summary = (S3ObjectSummary)iterator.next();
            s3.deleteObject(bucket_name, summary.getKey());
        }

        // more object_listing to retrieve?
        if (object_listing.isTruncated()) {
            object_listing = s3.listNextBatchOfObjects(object_listing);
        } else {
            break;
        }
    };
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

See the complete example.

Remove Objects from a Versioned Bucket Before Deleting It

If you're using a versioned bucket, you also need to remove any stored versions of the objects in the bucket before the bucket can be deleted.

Using a pattern similar to the one used when removing objects within a bucket, remove versioned objects by using the AmazonS3 client's `listVersions` method to list any versioned objects, and then `deleteVersion` to delete each one.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.ListVersionsRequest;
import com.amazonaws.services.s3.model.ObjectListing;
import com.amazonaws.services.s3.model.S3ObjectSummary;
import com.amazonaws.services.s3.model.S3VersionSummary;
import com.amazonaws.services.s3.model.VersionListing;
import java.util.Iterator;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
    System.out.println(" - removing objects from bucket");
    ObjectListing object_listing = s3.listObjects(bucket_name);
    while (true) {
        for (Iterator<?> iterator =
                object_listing.getObjectSummaries().iterator();
                iterator.hasNext();) {
            S3ObjectSummary summary = (S3ObjectSummary)iterator.next();
            s3.deleteObject(bucket_name, summary.getKey());
        }

        // more object_listing to retrieve?
        if (object_listing.isTruncated()) {
            object_listing = s3.listNextBatchOfObjects(object_listing);
        } else {
            break;
        }
    };

    System.out.println(" - removing versions from bucket");
    VersionListing version_listing = s3.listVersions(
            new ListVersionsRequest().withBucketName(bucket_name));
    while (true) {
        for (Iterator<?> iterator =
                version_listing.getVersionSummaries().iterator();
                iterator.hasNext();) {
            S3VersionSummary vs = (S3VersionSummary)iterator.next();
            s3.deleteVersion(
                    bucket_name, vs.getKey(), vs.getVersionId());
        }

        if (version_listing.isTruncated()) {
            version_listing = s3.listNextBatchOfVersions(
                    version_listing);
        } else {
            break;
        }
    }
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

See the complete example.

Delete an Empty Bucket

Once you remove the objects from a bucket (including any versioned objects), you can delete the bucket itself by using the AmazonS3 client's `deleteBucket` method.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
    s3.deleteBucket(bucket_name);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

See the complete example.

Performing Operations on Amazon S3 Objects

An Amazon S3 object represents a *file* or collection of data. Every object must reside within a bucket (p. 106).

Note
These code snippets assume that you understand the material in Using the AWS SDK for Java (p. 11) and have configured default AWS credentials using the information in Set up AWS Credentials and Region for Development (p. 6).

Topics

Upload an Object

Use the AmazonS3 client's `putObject` method, supplying a bucket name, key name, and file to upload. *The bucket must exist, or an error will result.*

Imports

```
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.AmazonServiceException;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
    s3.putObject(bucket_name, key_name, file_path);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

See the complete example.

List Objects

To get a list of objects within a bucket, use the AmazonS3 client's `listObjects` method, supplying the name of a bucket.

The `listObjects` method returns an ObjectListing object that provides information about the objects in the bucket. To list the object names (keys), use the `getObjectSummaries` method to get a List of S3ObjectSummary objects, each of which represents a single object in the bucket. Then call its `getKey` method to retrieve the object's name.

Imports

```
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.ObjectListing;
import com.amazonaws.services.s3.model.S3ObjectSummary;
import java.util.List;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
ObjectListing ol = s3.listObjects(bucket_name);
List<S3ObjectSummary> objects = ol.getObjectSummaries();
for (S3ObjectSummary os: objects) {
    System.out.println("* " + os.getKey());
}
```

See the complete example.

Download an Object

Use the AmazonS3 client's `getObject` method, passing it the name of a bucket and object to download. If successful, the method returns an S3Object. *The specified bucket and object key must exist, or an error will result*.

You can get the object's contents by calling `getObjectContent` on the `S3Object`. This returns an S3ObjectInputStream that behaves as a standard Java `InputStream` object.

The following example downloads an object from S3 and saves its contents to a file (using the same name as the object's key).

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.S3Object;
import com.amazonaws.services.s3.model.S3ObjectInputStream;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
    S3Object o = s3.getObject(bucket_name, key_name);
    S3ObjectInputStream s3is = o.getObjectContent();
    FileOutputStream fos = new FileOutputStream(new File(key_name));
    byte[] read_buf = new byte[1024];
    int read_len = 0;
    while ((read_len = s3is.read(read_buf)) > 0) {
        fos.write(read_buf, 0, read_len);
    }
    s3is.close();
    fos.close();
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
} catch (FileNotFoundException e) {
    System.err.println(e.getMessage());
    System.exit(1);
} catch (IOException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
```

See the complete example.

Copy, Move, or Rename Objects

You can copy an object from one bucket to another by using the AmazonS3 client's `copyObject` method. It takes the name of the bucket to copy from, the object to copy, and the destination bucket and name.

Imports

```
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3Client;
import com.amazonaws.AmazonServiceException;
```

Code

```
final AmazonS3 s3 = new AmazonS3Client();
try {
    s3.copyObject(from_bucket, object_key, to_bucket, object_key);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

See the complete example.

Note
You can use `copyObject` with deleteObject (p. 112) to **move** or **rename** an object, by first copying the object to a new name (you can use the same bucket as both the source and destination) and then deleting the object from its old location.

Delete an Object

Use the AmazonS3 client's `deleteObject` method, passing it the name of a bucket and object to delete. *The specified bucket and object key must exist, or an error will result.*

Imports

```
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.AmazonServiceException;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
    s3.deleteObject(bucket_name, object_key);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

See the complete example.

Delete Multiple Objects at Once

Using the AmazonS3 client's `deleteObjects` method, you can delete multiple objects from the same bucket by passing their names to the DeleteObjectRequest `withKeys` method.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.DeleteObjectsRequest;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
    DeleteObjectsRequest dor = new DeleteObjectsRequest(bucket_name)
        .withKeys(object_keys);
    s3.deleteObjects(dor);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

See the complete example.

Managing Amazon S3 Access Permissions for Buckets and Objects

You can use access control lists (ACLs) for Amazon S3 buckets and objects for fine-grained control over your Amazon S3 resources.

> **Note**
> These code snippets assume that you understand the material in Using the AWS SDK for Java (p. 11) and have configured default AWS credentials using the information in Set up AWS Credentials and Region for Development (p. 6).

Get the Access Control List for a Bucket

To get the current ACL for a bucket, call the AmazonS3's `getBucketAcl` method, passing it the *bucket name* to query. This method returns an AccessControlList object. To get each access grant in the list, call its `getGrantsAsList` method, which will return a standard Java list of Grant objects.

Imports

```
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.model.AccessControlList;
import com.amazonaws.services.s3.model.Grant;
import java.util.List;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
    AccessControlList acl = s3.getBucketAcl(bucket_name);
    List<Grant> grants = acl.getGrantsAsList();
    for (Grant grant : grants) {
        System.out.format("  %s: %s\n", grant.getGrantee().getIdentifier(),
                grant.getPermission().toString());
    }
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

See the complete example.

Set the Access Control List for a Bucket

To add or modify permissions to an ACL for a bucket, call the AmazonS3's `setBucketAcl` method. It takes an AccessControlList object that contains a list of grantees and access levels to set.

Imports

```
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.model.AccessControlList;
import com.amazonaws.services.s3.model.EmailAddressGrantee;
import com.amazonaws.services.s3.model.Permission;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
    // get the current ACL
    AccessControlList acl = s3.getBucketAcl(bucket_name);
    // set access for the grantee
    EmailAddressGrantee grantee = new EmailAddressGrantee(email);
    Permission permission = Permission.valueOf(access);
    acl.grantPermission(grantee, permission);
    s3.setBucketAcl(bucket_name, acl);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

Note
You can provide the grantee's unique identifier directly using the Grantee class, or use the EmailAddressGrantee class to set the grantee by email, as we've done here.

See the complete example.

Get the Access Control List for an Object

To get the current ACL for an object, call the AmazonS3's `getObjectAcl` method, passing it the *bucket name* and *object name* to query. Like `getBucketAcl`, this method returns an AccessControlList object that you can use to examine each Grant.

Imports

```
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.model.AccessControlList;
import com.amazonaws.services.s3.model.Grant;
import java.util.List;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
    AccessControlList acl = s3.getObjectAcl(bucket_name, object_key);
    List<Grant> grants = acl.getGrantsAsList();
    for (Grant grant : grants) {
        System.out.format("  %s: %s\n", grant.getGrantee().getIdentifier(),
                grant.getPermission().toString());
    }
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

See the complete example.

Set the Access Control List for an Object

To add or modify permissions to an ACL for an object, call the AmazonS3's `setObjectAcl` method. It takes an AccessControlList object that contains a list of grantees and access levels to set.

Imports

```
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.model.AccessControlList;
import com.amazonaws.services.s3.model.EmailAddressGrantee;
import com.amazonaws.services.s3.model.Permission;
```

Code

```
    final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
    try {
        // get the current ACL
        AccessControlList acl = s3.getObjectAcl(bucket_name, object_key);
        // set access for the grantee
        EmailAddressGrantee grantee = new EmailAddressGrantee(email);
        Permission permission = Permission.valueOf(access);
        acl.grantPermission(grantee, permission);
        s3.setObjectAcl(bucket_name, object_key, acl);
    } catch (AmazonServiceException e) {
        System.err.println(e.getErrorMessage());
        System.exit(1);
    }
}
```

Note

You can provide the grantee's unique identifier directly using the Grantee class, or use the EmailAddressGrantee class to set the grantee by email, as we've done here.

See the complete example.

More Information

- GET Bucket acl in the *Amazon S3 API Reference*
- PUT Bucket acl in the *Amazon S3 API Reference*
- GET Object acl in the *Amazon S3 API Reference*
- PUT Object acl in the *Amazon S3 API Reference*

Managing Access to Amazon S3 Buckets Using Bucket Policies

You can set, get, or delete a *bucket policy* to manage access to your Amazon S3 buckets.

Set a Bucket Policy

You can set the bucket policy for a particular S3 bucket by:

- Calling the AmazonS3 client's `setBucketPolicy` and providing it with a SetBucketPolicyRequest
- Setting the policy directly by using the `setBucketPolicy` overload that takes a bucket name and policy text (in JSON format)

Imports

```
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.AmazonServiceException;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
    s3.setBucketPolicy(bucket_name, policy_text);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

Use the Policy Class to Generate or Validate a Policy

When providing a bucket policy to `setBucketPolicy`, you can do the following:

- Specify the policy directly as a string of JSON-formatted text
- Build the policy using the Policy class

By using the `Policy` class, you don't have to be concerned about correctly formatting your text string. To get the JSON policy text from the `Policy` class, use its `toJson` method.

Imports

```
import com.amazonaws.auth.policy.Policy;
import com.amazonaws.auth.policy.Principal;
import com.amazonaws.auth.policy.Resource;
import com.amazonaws.auth.policy.Statement;
import com.amazonaws.auth.policy.actions.S3Actions;
import java.io.IOException;
```

Code

```
Policy bucket_policy = new Policy().withStatements(
    new Statement(Statement.Effect.Allow)
        .withPrincipals(Principal.AllUsers)
        .withActions(S3Actions.GetObject)
        .withResources(new Resource(
            "arn:aws:s3:::" + bucket_name + "/*")));
return bucket_policy.toJson();
```

The `Policy` class also provides a `fromJson` method that can attempt to build a policy using a passed-in JSON string. The method validates it to ensure that the text can be transformed into a valid policy structure, and will fail with an `IllegalArgumentException` if the policy text is invalid.

```
Policy bucket_policy = null;
try {
    bucket_policy = Policy.fromJson(file_text.toString());
} catch (IllegalArgumentException e) {
    System.out.format("Invalid policy text in file: \"%s\"",
            policy_file);
    System.out.println(e.getMessage());
}
```

You can use this technique to prevalidate a policy that you read in from a file or other means.

See the complete example.

Get a Bucket Policy

To retrieve the policy for an Amazon S3 bucket, call the AmazonS3 client's `getBucketPolicy` method, passing it the name of the bucket to get the policy from.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.BucketPolicy;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
   BucketPolicy bucket_policy = s3.getBucketPolicy(bucket_name);
   policy_text = bucket_policy.getPolicyText();
} catch (AmazonServiceException e) {
   System.err.println(e.getErrorMessage());
   System.exit(1);
}
```

If the named bucket doesn't exist, if you don't have access to it, or if it has no bucket policy, an `AmazonServiceException` is thrown.

See the complete example.

Delete a Bucket Policy

To delete a bucket policy, call the AmazonS3 client's `deleteBucketPolicy`, providing it with the bucket name.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
   s3.deleteBucketPolicy(bucket_name);
} catch (AmazonServiceException e) {
   System.err.println(e.getErrorMessage());
   System.exit(1);
}
```

This method succeeds even if the bucket doesn't already have a policy. If you specify a bucket name that doesn't exist or if you don't have access to the bucket, an `AmazonServiceException` is thrown.

See the complete example.

More Info

- Access Policy Language Overview in the *Amazon S3 Developer Guide*
- Bucket Policy Examples in the *Amazon S3 Developer Guide*

Using TransferManager for Amazon S3 Operations

You can use the AWS SDK for JavaTransferManager class to reliably transfer files from the local environment to Amazon S3 and to copy objects from one S3 location to another. `TransferManager` can get the progress of a transfer and pause or resume uploads and downloads.
<admonition>
<title>Best Practice</title>

We recommend that you enable the AbortIncompleteMultipartUpload lifecycle rule on your Amazon S3 buckets.

This rule directs Amazon S3 to abort multipart uploads that don't complete within a specified number of days after being initiated. When the set time limit is exceeded, Amazon S3 aborts the upload and then deletes the incomplete upload data.

For more information, see Lifecycle Configuration for a Bucket with Versioning in the *Amazon S3 User Guide*.
</admonition>

Note
These code snippets assume that you understand the material in Using the AWS SDK for Java (p. 11) and have configured default AWS credentials using the information in Set up AWS Credentials and Region for Development (p. 6).

Upload Files and Directories

TransferManager can upload files, file lists, and directories to any Amazon S3 buckets that you've previously created (p. 106).

Topics

Upload a Single File

Call the TransferManager`upload` method, providing an Amazon S3 bucket name, a key (object) name, and a standard Java File object that represents the file to upload.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.Upload;
import java.io.File;
```

Code

```
File f = new File(file_path);
TransferManager xfer_mgr = new TransferManager();
try {
    Upload xfer = xfer_mgr.upload(bucket_name, key_name, f);
    // loop with Transfer.isDone()
    //  or block with Transfer.waitForCompletion()
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

The `upload` method returns *immediately*, providing an `Upload` object to use to check the transfer state or to wait for it to complete.

See Wait for a Transfer to Complete (p. 123) for information about using `waitForCompletion` to successfully complete a transfer before calling TransferManager's `shutdownNow` method. While waiting for the transfer to complete, you can poll or listen for updates about its status and progress. See Get Transfer Status and Progress (p. 123) for more information.

See the complete example.

Upload a List of Files

To upload multiple files in one operation, call the TransferManager`uploadFileList` method, providing the following:

- An Amazon S3 bucket name
- A *key prefix* to prepend to the names of the created objects (the path within the bucket in which to place the objects)
- A File object that represents the relative directory from which to create file paths

- A List object containing a set of File objects to upload

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.MultipleFileUpload;
import java.io.File;
import java.util.ArrayList;
```

Code

```
ArrayList<File> files = new ArrayList<File>();
for (String path : file_paths) {
    files.add(new File(path));
}

TransferManager xfer_mgr = new TransferManager();
try {
    MultipleFileUpload xfer = xfer_mgr.uploadFileList(bucket_name,
            key_prefix, new File("."), files);
    // loop with Transfer.isDone()
    // or block with Transfer.waitForCompletion()
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

See Wait for a Transfer to Complete (p. 123) for information about using `waitForCompletion` to successfully complete a transfer before calling TransferManager's `shutdownNow` method. While waiting for the transfer to complete, you can poll or listen for updates about its status and progress. See Get Transfer Status and Progress (p. 123) for more information.

The MultipleFileUpload object returned by `uploadFileList` can be used to query the transfer state or progress. See Poll the Current Progress of a Transfer (p. 124) and Get Transfer Progress with a ProgressListener (p. 124) for more information.

You can also use `MultipleFileUpload`'s `getSubTransfers` method to get the individual `Upload` objects for each file being transferred. For more information, see Get the Progress of Subtransfers (p. 125).

See the complete example.

Upload a Directory

You can use TransferManager's `uploadDirectory` method to upload an entire directory of files, with the option to copy files in subdirectories recursively. You provide an Amazon S3 bucket name, an S3 key prefix, a File object representing the local directory to copy, and a `boolean` value indicating whether you want to copy subdirectories recursively (*true* or *false*).

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.Upload;
import com.amazonaws.services.s3.transfer.MultipleFileUpload;
import java.io.File;
```

Code

```
TransferManager xfer_mgr = new TransferManager();
try {
    MultipleFileUpload xfer = xfer_mgr.uploadDirectory(bucket_name,
            key_prefix, new File(dir_path), recursive);
    // loop with Transfer.isDone()
    // or block with Transfer.waitForCompletion()
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

See Wait for a Transfer to Complete (p. 123) for information about using `waitForCompletion` to successfully complete a transfer before calling TransferManager's `shutdownNow` method. While waiting for the transfer to complete, you can poll or listen for updates about its status and progress. See Get Transfer Status and Progress (p. 123) for more information.

The MultipleFileUpload object returned by `uploadFileList` can be used to query the transfer state or progress. See Poll the Current Progress of a Transfer (p. 124) and Get Transfer Progress with a ProgressListener (p. 124) for more information.

You can also use `MultipleFileUpload`'s `getSubTransfers` method to get the individual `Upload` objects for each file being transferred. For more information, see Get the Progress of Subtransfers (p. 125).

See the complete example.

Download Files or Directories

Use the TransferManager class to download either a single file (Amazon S3 object) or a directory (an Amazon S3 bucket name followed by an object prefix) from Amazon S3.

Topics

Download a Single File

Use the TransferManager's `download` method, providing the Amazon S3 bucket name containing the object you want to download, the key (object) name, and a File object that represents the file to create on your local system.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.Download;
import java.io.File;
```

Code

```
File f = new File(file_path);
TransferManager xfer_mgr = new TransferManager();
try {
    Download xfer = xfer_mgr.download(bucket_name, key_name, f);
    // loop with Transfer.isDone()
    // or block with Transfer.waitForCompletion()
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

See Wait for a Transfer to Complete (p. 123) for information about using `waitForCompletion` to successfully complete a transfer before calling TransferManager's `shutdownNow` method. While waiting for the transfer to complete, you can poll or listen for updates about its status and progress. See Get Transfer Status and Progress (p. 123) for more information.

See the complete example.

Download a Directory

To download a set of files that share a common key prefix (analagous to a directory on a file system) from Amazon S3, use the TransferManager`downloadDirectory` method. The method takes the Amazon S3 bucket name containing the objects you want to download, the object prefix shared by all of the objects, and a File object that represents the directory to download the files into on your local system. If the named directory doesn't exist yet, it will be created.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.MultipleFileDownload;
import java.io.File;
```

Code

```
TransferManager xfer_mgr = new TransferManager();
try {
    MultipleFileDownload xfer = xfer_mgr.downloadDirectory(
            bucket_name, key_prefix, new File(dir_path));
    // loop with Transfer.isDone()
    // or block with Transfer.waitForCompletion()
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

See Wait for a Transfer to Complete (p. 123) for information about using `waitForCompletion` to successfully complete a transfer before calling TransferManager's `shutdownNow` method. While waiting for the transfer to complete, you can poll or listen for updates about its status and progress. See Get Transfer Status and Progress (p. 123) for more information.

See the complete example.

Copy Objects

To copy an object from one S3 bucket to another, use the TransferManager`copy` method.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.Copy;
import com.amazonaws.services.s3.transfer.TransferManager;
```

Code

```
TransferManager xfer_mgr = new TransferManager();
try {
    Copy xfer = xfer_mgr.copy(from_bucket, from_key, to_bucket, to_key);
    // loop with Transfer.isDone()
    // or block with Transfer.waitForCompletion()
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

See the complete example.

Wait for a Transfer to Complete

If your application (or thread) can block until the transfer completes, you can use the Transfer interface's `waitForCompletion` method to block until the transfer is complete or an exception occurs.

```
try {
    xfer.waitForCompletion();
} catch (AmazonServiceException e) {
    System.err.println("Amazon service error: " + e.getMessage());
    System.exit(1);
} catch (AmazonClientException e) {
    System.err.println("Amazon client error: " + e.getMessage());
    System.exit(1);
} catch (InterruptedException e) {
    System.err.println("Transfer interrupted: " + e.getMessage());
    System.exit(1);
}
```

You get progress of transfers if you poll for events *before* calling `waitForCompletion`, implement a polling mechanism on a separate thread, or receive progress updates asynchronously using a ProgressListener.

See the complete example.

Get Transfer Status and Progress

Each of the classes returned by the TransferManager`upload*`, `download*`, and `copy` methods returns an instance of one of the following classes, depending on whether it's a single-file or multiple-file operation.

Class	Returned by
Copy	`copy`
Download	`download`
MultipleFileDownload	`downloadDirectory`
Upload	`upload`
MultipleFileUpload	`uploadFileList`, `uploadDirectory`

All of these classes implement the Transfer interface. `Transfer` provides useful methods to get the progress of a transfer, pause or resume the transfer, and get the transfer's current or final status.

Topics

Poll the Current Progress of a Transfer

This loop prints the progress of a transfer, examines its current progress while running and, when complete, prints its final state.

Imports

```
import com.amazonaws.services.s3.transfer.TransferProgress;
```

Code

```
do {
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        return;
    }
    TransferProgress progress = xfer.getProgress();
    long so_far = progress.getBytesTransferred();
    long total = progress.getTotalBytesToTransfer();
    double pct = progress.getPercentTransferred();
} while (xfer.isDone() == false);
```

See the complete example.

Get Transfer Progress with a ProgressListener

You can attach a ProgressListener to any transfer by using the Transfer interface's `addProgressListener` method.

A ProgressListener requires only one method, `progressChanged`, which takes a ProgressEvent object. You can use the object to get the total bytes of the operation by calling its `getBytes` method, and the number of bytes transferred so far by calling `getBytesTransferred`.

Imports

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.event.ProgressEvent;
import com.amazonaws.event.ProgressListener;
import com.amazonaws.services.s3.transfer.Upload;
import java.io.File;
```

Code

```
File f = new File(file_path);
TransferManager xfer_mgr = new TransferManager();
try {
    Upload u = xfer_mgr.upload(bucket_name, key_name, f);
    u.addProgressListener(new ProgressListener() {
        public void progressChanged(ProgressEvent e) {
            double pct = e.getBytesTransferred() * 100.0 / e.getBytes();
        }
    });
    // block with Transfer.waitForCompletion()
    TransferState xfer_state = u.getState();
    System.out.println(": " + xfer_state);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

See the complete example.

Get the Progress of Subtransfers

The MultipleFileUpload class can return information about its subtransfers by calling its `getSubTransfers` method. It returns an unmodifiable Collection of Upload objects that provide the individual transfer status and progress of each subtransfer.

Imports

```
import com.amazonaws.services.s3.transfer.Upload;
import com.amazonaws.services.s3.transfer.MultipleFileUpload;
import java.util.ArrayList;
import java.util.Collection;
```

Code

```
Collection<? extends Upload> sub_xfers = new ArrayList<Upload>();
sub_xfers = multi_upload.getSubTransfers();
```

See the complete example.

More Info

- Object Keys in the *Amazon S3 Developer Guide*

Configuring an Amazon S3 Bucket as a Website

You can configure an Amazon S3 bucket to behave as a website. To do this, you need to set its website configuration.

> **Note**
> These code snippets assume that you understand the material in Using the AWS SDK for Java (p. 11) and have configured default AWS credentials using the information in Set up AWS Credentials and Region for Development (p. 6).

Set a Bucket's Website Configuration

To set an Amazon S3 bucket's website configuration, call the AmazonS3's `setWebsiteConfiguration` method with the bucket name to set the configuration for, and a BucketWebsiteConfiguration object containing the bucket's website configuration.

Setting an index document is *required*; all other parameters are optional.

Imports

```
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.BucketWebsiteConfiguration;
import com.amazonaws.AmazonServiceException;
```

Code

```
BucketWebsiteConfiguration website_config = null;

if (index_doc == null) {
    website_config = new BucketWebsiteConfiguration();
} else if (error_doc == null) {
    website_config = new BucketWebsiteConfiguration(index_doc);
} else {
    website_config = new BucketWebsiteConfiguration(index_doc, error_doc);
}

final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
    s3.setBucketWebsiteConfiguration(bucket_name, website_config);
} catch (AmazonServiceException e) {
    System.out.format(
        "Failed to set website configuration for bucket '%s'!\n",
        bucket_name);
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

Note
Setting a website configuration does not modify the access permissions for your bucket. To make your files visible on the web, you will also need to set a *bucket policy* that allows public read access to the files in the bucket. For more information, see Managing Access to Amazon S3 Buckets Using Bucket Policies (p. 116).

See the complete example.

Get a Bucket's Website Configuration

To get an Amazon S3 bucket's website configuration, call the AmazonS3's `getWebsiteConfiguration` method with the name of the bucket to retrieve the configuration for.

The configuration will be returned as a BucketWebsiteConfiguration object. If there is no website configuration for the bucket, then `null` will be returned.

Imports

```
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.BucketWebsiteConfiguration;
import com.amazonaws.AmazonServiceException;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
    BucketWebsiteConfiguration config =
        s3.getBucketWebsiteConfiguration(bucket_name);
    if (config == null) {
        System.out.println("No website configuration found!");
    } else {
        System.out.format("Index document: %s\n",
            config.getIndexDocumentSuffix());
        System.out.format("Error document: %s\n",
            config.getErrorDocument());
    }
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.out.println("Failed to get website configuration!");
    System.exit(1);
}
```

See the complete example.

Delete a Bucket's Website Configuration

To delete an Amazon S3 bucket's website configuration, call the AmazonS3's `deleteWebsiteConfiguration` method with the name of the bucket to delete the configuration from.

Imports

```
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.AmazonServiceException;
```

Code

```
final AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
try {
    s3.deleteBucketWebsiteConfiguration(bucket_name);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.out.println("Failed to delete website configuration!");
    System.exit(1);
}
```

See the complete example.

More Information

- PUT Bucket website in the *Amazon S3 API Reference*
- GET Bucket website in the *Amazon S3 API Reference*
- DELETE Bucket website in the *Amazon S3 API Reference*

Amazon SQS Examples

This section provides examples of programming Amazon SQS using the AWS SDK for Java.

> **Note**
> The examples include only the code needed to demonstrate each technique. The complete example code is available on GitHub. From there, you can download a single source file or clone the repository locally to get all the examples to build and run.

Topics

Working with Amazon SQS Message Queues

A *message queue* is the logical container used for sending messages reliably in Amazon SQS. There are two types of queues: *standard* and *first-in, first-out* (FIFO). To learn more about queues and the differences between these types, see the Amazon SQS Developer Guide.

This topic describes how to create, list, delete, and get the URL of an Amazon SQS queue by using the AWS SDK for Java.

Create a Queue

Use the AmazonSQS client's `createQueue` method, providing a CreateQueueRequest object that describes the queue parameters.

Imports

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.AmazonSQSException;
import com.amazonaws.services.sqs.model.CreateQueueRequest;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
CreateQueueRequest create_request = new CreateQueueRequest(QUEUE_NAME)
        .addAttributesEntry("DelaySeconds", "60")
        .addAttributesEntry("MessageRetentionPeriod", "86400");

try {
    sqs.createQueue(create_request);
} catch (AmazonSQSException e) {
    if (!e.getErrorCode().equals("QueueAlreadyExists")) {
        throw e;
    }
}
```

You can use the simplified form of `createQueue`, which needs only a queue name, to create a standard queue.

```
sqs.createQueue("MyQueue" + new Date().getTime());
```

See the complete sample.

Listing Queues

To list the Amazon SQS queues for your account, call the AmazonSQS client's `listQueues` method.

Imports

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.ListQueuesResult;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
ListQueuesResult lq_result = sqs.listQueues();
System.out.println("Your SQS Queue URLs:");
for (String url : lq_result.getQueueUrls()) {
    System.out.println(url);
}
```

Using the `listQueues` overload without any parameters returns *all queues*. You can filter the returned results by passing it a `ListQueuesRequest` object.

Imports

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.ListQueuesRequest;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
String name_prefix = "Queue";
lq_result = sqs.listQueues(new ListQueuesRequest(name_prefix));
System.out.println("Queue URLs with prefix: " + name_prefix);
for (String url : lq_result.getQueueUrls()) {
    System.out.println(url);
}
```

See the complete sample.

Get the URL for a Queue

Call the AmazonSQS client's `getQueueUrl` method.

Imports

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
String queue_url = sqs.getQueueUrl(QUEUE_NAME).getQueueUrl();
```

See the complete sample.

Delete a Queue

Provide the queue's URL (p. 129) to the AmazonSQS client's `deleteQueue` method.

Imports

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
sqs.deleteQueue(queue_url);
```

See the complete sample.

More Info

- How Amazon SQS Queues Work in the *Amazon SQS Developer Guide*
- CreateQueue in the *Amazon SQS API Reference*
- GetQueueUrl in the *Amazon SQS API Reference*
- ListQueues in the *Amazon SQS API Reference*
- DeleteQueues in the *Amazon SQS API Reference*

Sending, Receiving, and Deleting Amazon SQS Messages

This topic describes how to send, receive and delete Amazon SQS messages. Messages are always delivered using an SQS Queue (p. 128).

Send a Message

Add a single message to an Amazon SQS queue by calling the AmazonSQS client's `sendMessage` method. Provide a SendMessageRequest object that contains the queue's URL (p. 129), message body, and optional delay value (in seconds).

Imports

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.SendMessageRequest;
```

Code

```
SendMessageRequest send_msg_request = new SendMessageRequest()
        .withQueueUrl(queueUrl)
        .withMessageBody("hello world")
        .withDelaySeconds(5);
sqs.sendMessage(send_msg_request);
```

Send Multiple Messages at Once

You can send more than one message in a single request. To send multiple messages, use the AmazonSQS client's `sendMessageBatch` method, which takes a SendMessageBatchRequest containing the queue URL and a list of messages (each one a SendMessageBatchRequestEntry) to send. You can also set an optional delay value per message.

Imports

```
import com.amazonaws.services.sqs.model.SendMessageBatchRequest;
import com.amazonaws.services.sqs.model.SendMessageBatchRequestEntry;
```

Code

```
SendMessageBatchRequest send_batch_request = new SendMessageBatchRequest()
        .withQueueUrl(queueUrl)
        .withEntries(
                new SendMessageBatchRequestEntry(
                        "msg_1", "Hello from message 1"),
                new SendMessageBatchRequestEntry(
                        "msg_2", "Hello from message 2")
                        .withDelaySeconds(10));
sqs.sendMessageBatch(send_batch_request);
```

See the complete sample.

Receive Messages

Retrieve any messages that are currently in the queue by calling the AmazonSQS client's `receiveMessage` method, passing it the queue's URL. Messages are returned as a list of Message objects.

Imports

```
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.AmazonSQSException;
import com.amazonaws.services.sqs.model.SendMessageBatchRequest;
```

Code

```
List<Message> messages = sqs.receiveMessage(queueUrl).getMessages();
```

Delete Messages after Receipt

After receiving a message and processing its contents, delete the message from the queue by sending the message's receipt handle and queue URL to the AmazonSQS client's `deleteMessage` method.

Code

```
for (Message m : messages) {
    sqs.deleteMessage(queueUrl, m.getReceiptHandle());
}
```

See the complete sample.

More Info

- How Amazon SQS Queues Work in the *Amazon SQS Developer Guide*
- SendMessage in the *Amazon SQS API Reference*

- SendMessageBatch in the *Amazon SQS API Reference*
- ReceiveMessage in the *Amazon SQS API Reference*
- DeleteMessage in the *Amazon SQS API Reference*

Enabling Long Polling for Amazon SQS Message Queues

Amazon SQS uses *short polling* by default, querying only a subset of the servers—based on a weighted random distribution—to determine whether any messages are available for inclusion in the response.

Long polling helps reduce your cost of using Amazon SQS by reducing the number of empty responses when there are no messages available to return in reply to a ReceiveMessage request sent to an Amazon SQS queue and eliminating false empty responses.

Note
You can set a long polling frequency from *1–20 seconds*.

Enabling Long Polling when Creating a Queue

To enable long polling when creating an Amazon SQS queue, set the `ReceiveMessageWaitTimeSeconds` attribute on the CreateQueueRequest object before calling the AmazonSQS class' `createQueue` method.

Imports

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.AmazonSQSException;
import com.amazonaws.services.sqs.model.CreateQueueRequest;
```

Code

```
final AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();

// Enable long polling when creating a queue
CreateQueueRequest create_request = new CreateQueueRequest()
        .withQueueName(queue_name)
        .addAttributesEntry("ReceiveMessageWaitTimeSeconds", "20");

try {
    sqs.createQueue(create_request);
} catch (AmazonSQSException e) {
    if (!e.getErrorCode().equals("QueueAlreadyExists")) {
        throw e;
    }
}
```

See the complete example.

Enabling Long Polling on an Existing Queue

In addition to enabling long polling when creating a queue, you can also enable it on an existing queue by setting `ReceiveMessageWaitTimeSeconds` on the SetQueueAttributesRequest before calling the AmazonSQS class' `setQueueAttributes` method.

Imports

```
import com.amazonaws.services.sqs.model.SetQueueAttributesRequest;
```

Code

```
SetQueueAttributesRequest set_attrs_request = new SetQueueAttributesRequest()
        .withQueueUrl(queue_url)
        .addAttributesEntry("ReceiveMessageWaitTimeSeconds", "20");
sqs.setQueueAttributes(set_attrs_request);
```

See the complete example.

Enabling Long Polling on Message Receipt

You can enable long polling when receiving a message by setting the wait time in seconds on the ReceiveMessageRequest that you supply to the AmazonSQS class' `receiveMessage` method.

> **Note**
> You should make sure that the AWS client's request timeout is larger than the maximum long poll time (20s) so that your `receiveMessage` requests don't time out while waiting for the next poll event!

Imports

```
import com.amazonaws.services.sqs.model.ReceiveMessageRequest;
```

Code

```
ReceiveMessageRequest receive_request = new ReceiveMessageRequest()
        .withQueueUrl(queue_url)
        .withWaitTimeSeconds(20);
sqs.receiveMessage(receive_request);
```

See the complete example.

More Info

- Amazon SQS Long Polling in the *Amazon SQS Developer Guide*
- CreateQueue in the *Amazon SQS API Reference*
- ReceiveMessage in the *Amazon SQS API Reference*
- SetQueueAttributes in the *Amazon SQS API Reference*

Setting Visibility Timeout in Amazon SQS

When a message is received in Amazon SQS, it remains on the queue until it's deleted in order to ensure receipt. A message that was received, but not deleted, will be available in subsequent requests after a given *visibility timeout* to help prevent the message from being received more than once before it can be processed and deleted.

> **Note**
> When using standard queues, visibility timeout isn't a guarantee against receiving a message twice. If you are using a standard queue, be sure that your code can handle the case where the same message has been delivered more than once.

Setting the Message Visibility Timeout for a Single Message

When you have received a message, you can modify its visibility timeout by passing its receipt handle in a ChangeMessageVisibilityRequest that you pass to the AmazonSQS class' `changeMessageVisibility` method.

Imports

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();

// Get the receipt handle for the first message in the queue.
String receipt = sqs.receiveMessage(queue_url)
                    .getMessages()
                    .get(0)
                    .getReceiptHandle();

sqs.changeMessageVisibility(queue_url, receipt, timeout);
```

See the complete example.

Setting the Message Visibility Timeout for Multiple Messages at Once

To set the message visibility timeout for multiple messages at once, create a list of ChangeMessageVisibilityBatchRequestEntry objects, each containing a unique ID string and a receipt handle. Then, pass the list to the Amazon SQS client class' `changeMessageVisibilityBatch` method.

Imports

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.ChangeMessageVisibilityBatchRequestEntry;
import java.util.ArrayList;
import java.util.List;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();

List<ChangeMessageVisibilityBatchRequestEntry> entries =
    new ArrayList<ChangeMessageVisibilityBatchRequestEntry>();

entries.add(new ChangeMessageVisibilityBatchRequestEntry(
            "unique_id_msg1",
            sqs.receiveMessage(queue_url)
               .getMessages()
               .get(0)
               .getReceiptHandle())
        .withVisibilityTimeout(timeout));

entries.add(new ChangeMessageVisibilityBatchRequestEntry(
            "unique_id_msg2",
            sqs.receiveMessage(queue_url)
               .getMessages()
```

```
                .get(0)
                .getReceiptHandle())
        .withVisibilityTimeout(timeout + 200));

sqs.changeMessageVisibilityBatch(queue_url, entries);
```

See the complete example.

More Info

- Visibility Timeout in the *Amazon SQS Developer Guide*
- SetQueueAttributes in the *Amazon SQS API Reference*
- GetQueueAttributes in the *Amazon SQS API Reference*
- ReceiveMessage in the *Amazon SQS API Reference*
- ChangeMessageVisibility in the *Amazon SQS API Reference*
- ChangeMessageVisibilityBatch in the *Amazon SQS API Reference*

Using Dead Letter Queues in Amazon SQS

Amazon SQS provides support for *dead letter queues*. A dead letter queue is a queue that other (source) queues can target for messages that can't be processed successfully. You can set aside and isolate these messages in the dead letter queue to determine why their processing did not succeed.

Creating a Dead Letter Queue

A dead letter queue is created the same way as a regular queue, but it has the following restrictions:

- A dead letter queue must be the same type of queue (FIFO or standard) as the source queue.
- A dead letter queue must be created using the same AWS account and region as the source queue.

Here we create two identical Amazon SQS queues, one of which will serve as the dead letter queue:

Imports

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.AmazonSQSException;
```

Code

```
final AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();

// Create source queue
try {
    sqs.createQueue(src_queue_name);
} catch (AmazonSQSException e) {
    if (!e.getErrorCode().equals("QueueAlreadyExists")) {
        throw e;
    }
}

// Create dead-letter queue
try {
    sqs.createQueue(dl_queue_name);
} catch (AmazonSQSException e) {
    if (!e.getErrorCode().equals("QueueAlreadyExists")) {
        throw e;
    }
}
```

See the complete example.

Designating a Dead Letter Queue for a Source Queue

To designate a dead letter queue, you must first create a *redrive policy*, and then set the policy in the queue's attributes. A redrive policy is specified in JSON, and specifies the ARN of the dead letter queue and the maximum number of times the message can be received and not processed before it's sent to the dead letter queue.

To set the redrive policy for your source queue, call the AmazonSQS class' `setQueueAttributes` method with a SetQueueAttributesRequest object for which you've set the `RedrivePolicy` attribute with your JSON redrive policy.

Imports

```
import com.amazonaws.services.sqs.model.GetQueueAttributesRequest;
import com.amazonaws.services.sqs.model.GetQueueAttributesResult;
import com.amazonaws.services.sqs.model.SetQueueAttributesRequest;
```

Code

```
String dl_queue_url = sqs.getQueueUrl(dl_queue_name)
                         .getQueueUrl();

GetQueueAttributesResult queue_attrs = sqs.getQueueAttributes(
        new GetQueueAttributesRequest(dl_queue_url)
            .withAttributeNames("QueueArn"));

String dl_queue_arn = queue_attrs.getAttributes().get("QueueArn");

// Set dead letter queue with redrive policy on source queue.
String src_queue_url = sqs.getQueueUrl(src_queue_name)
                          .getQueueUrl();

SetQueueAttributesRequest request = new SetQueueAttributesRequest()
        .withQueueUrl(src_queue_url)
        .addAttributesEntry("RedrivePolicy",
                "{\"maxReceiveCount\":\"5\", \"deadLetterTargetArn\":\""
                + dl_queue_arn + "\"}");

sqs.setQueueAttributes(request);
```

See the complete example.

More Info

- Using Amazon SQS Dead Letter Queues in the *Amazon SQS Developer Guide*
- SetQueueAttributes in the *Amazon SQS API Reference*

Getting Temporary Credentials with AWS STS

You can use AWS Security Token Service (AWS STS) to get temporary, limited-privilege credentials that can be used to access AWS services.

There are three steps involved in using AWS STS:

1. Activate a region (optional).
2. Retrieve temporary security credentials from AWS STS.
3. Use the credentials to access AWS resources.

Note
Activating a region is *optional*; by default, temporary security credentials are obtained from the global endpoint *sts.amazonaws.com*. However, to reduce latency and to enable you to build redundancy into your requests by using additional endpoints if an AWS STS request to the first endpoint fails, you can activate regions that are geographically closer to your services or applications that use the credentials.

(Optional) Activate and use an AWS STS region

To activate a region for use with AWS STS, use the AWS Management Console to select and activate the region.

To activate additional STS regions

1. Sign in as an IAM user with permissions to perform IAM administration tasks `"iam:*"` for the account for which you want to activate AWS STS in a new region.
2. Open the IAM console and in the navigation pane click **Account Settings**.
3. Expand the **STS Regions** list, find the region that you want to use, and then click **Activate**.

After this, you can direct calls to the STS endpoint that is associated with that region.

Note
For more information about activating STS regions and for a list of the available AWS STS endpoints, see Activating and Deactivating AWS STS in an AWS Region in the *IAM User Guide*.

Retrieve temporary security credentials from AWS STS

To retrieve temporary security credentials using the AWS SDK for Java

1. Create an AWSSecurityTokenServiceClient object:

```
AWSSecurityTokenServiceClient sts_client = new AWSSecurityTokenServiceClient();
```

   When creating the client with no arguments, the default credential provider chain is used to retrieve credentials. You can provide a specific credential provider if you want. For more information, see Providing AWS Credentials in the AWS SDK for Java.

2. *Optional*; requires that you have activated the region) Set the endpoint for the STS client:

```
sts_client.setEndpoint("sts-endpoint.amazonaws.com");
```

   where *sts-endpoint* represents the STS endpoint for your region.

   Important
   Do not use the `setRegion` method to set a regional endpoint—for backwards compatibility, that method continues to use the single global endpoint of sts.amazonaws.com.

3. Create a GetSessionTokenRequest object, and optionally set the duration in seconds for which the temporary credentials are valid:

   ```
   GetSessionTokenRequest session_token_request = new GetSessionTokenRequest();
   session_token_request.setDurationSeconds(7200); // optional.
   ```

   The duration of temporary credentials can range from 900 seconds (15 minutes) to 129600 seconds (36 hours) for IAM users. If a duration isn't specified, then 43200 seconds (12 hours) is used by default.

   For a root AWS account, the valid range of temporary credentials is from 900 to 3600 seconds (1 hour), with a default value of 3600 seconds if no duration is specified.

   Important
   It is *strongly recommended*, from a security standpoint, that you *use IAM users* instead of the root account for AWS access. For more information, see IAM Best Practices in the *IAM User Guide*.
4. Call getSessionToken on the STS client to get a session token, using the `GetSessionTokenRequest` object:

   ```
   GetSessionTokenResult session_token_result =
       sts_client.getSessionToken(session_token_request);
   ```
5. Get session credentials using the result of the call to `getSessionToken`:

   ```
   Credentials session_creds = session_token_result.getCredentials();
   ```

The session credentials provide access only for the duration that was specified by the `GetSessionTokenRequest` object. Once the credentials expire, you will need to call `getSessionToken` again to obtain a new session token for continued access to AWS.

Use the temporary credentials to access AWS resources

Once you have temporary security credentials, you can use them to initialize an AWS service client to use its resources, using the technique described in Explicitly Specifying Credentials (p. 17).

For example, to create an S3 client using temporary service credentials:

```
BasicSessionCredentials sessionCredentials = new BasicSessionCredentials(
   session_creds.getAccessKeyId(),
   session_creds.getSecretAccessKey(),
   session_creds.getSessionToken());

AmazonS3 s3 = AmazonS3ClientBuilder.standard()
                        .withCredentials(new
 AWSStaticCredentialsProvider(sessionCredentials)
                        .build();
```

You can now use the `AmazonS3` object to make Amazon S3 requests.

For more information

For more information about how to use temporary security credentials to access AWS resources, visit the following sections in the *IAM User Guide*:

- Requesting Temporary Security Credentials
- Controlling Permissions for Temporary Security Credentials
- Using Temporary Security Credentials to Request Access to AWS Resources
- Activating and Deactivating AWS STS in an AWS Region

Amazon SWF

Amazon SWF is a workflow-management service that helps developers build and scale distributed workflows that can have parallel or sequential steps consisting of activities, child workflows or even Lambda tasks.

There are two ways to work with Amazon SWF using the AWS SDK for Java, by using the SWF *client* object, or by using the AWS Flow Framework for Java. The AWS Flow Framework for Java is more difficult to configure initially, since it makes heavy use of annotations and relies on additional libraries such as AspectJ and the Spring Framework. However, for large or complex projects, you will save coding time by using the AWS Flow Framework for Java. For more information, see the AWS Flow Framework for Java Developer Guide.

This section provides examples of programming Amazon SWF by using the AWS SDK for Java client directly.

Topics

Amazon SWF Basics

These are general patterns for working with Amazon SWF using the AWS SDK for Java. It is meant primarily for reference. For a more complete introductory tutorial, see Building a Simple Amazon SWF Application (p. 140).

Dependencies

Basic Amazon SWF applications will require the following dependencies, which are included with the AWS SDK for Java:

- aws-java-sdk-1.11.*.jar
- commons-logging-1.1.*.jar
- httpclient-4.3.*.jar
- httpcore-4.3.*.jar
- jackson-annotations-2.5.*.jar
- jackson-core-2.5.*.jar
- jackson-databind-2.5.*.jar
- joda-time-2.8.*.jar

Note
the version numbers of these packages will differ depending on the version of the SDK that you have, but the versions that are supplied with the SDK have been tested for compatibility, and are the ones you should use.

AWS Flow Framework for Java applications require additonal setup, *and* additional dependencies. See the AWS Flow Framework for Java Developer Guide for more information about using the framework.

Imports

In general, you can use the following imports for code development:

```
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
```

It's a good practice to import only the classes you require, however. You will likely end up specifying particular classes in the `com.amazonaws.services.simpleworkflow.model` workspace:

```
import com.amazonaws.services.simpleworkflow.model.PollForActivityTaskRequest;
import com.amazonaws.services.simpleworkflow.model.RespondActivityTaskCompletedRequest;
import com.amazonaws.services.simpleworkflow.model.RespondActivityTaskFailedRequest;
import com.amazonaws.services.simpleworkflow.model.TaskList;
```

If you are using the AWS Flow Framework for Java, you will import classes from the `com.amazonaws.services.simpleworkflow.flow` workspace. For example:

```
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.flow.ActivityWorker;
```

Note
The AWS Flow Framework for Java has additional requirements beyond those of the base AWS SDK for Java. For more information, see the AWS Flow Framework for Java Developer Guide.

Using the SWF client class

Your basic interface to Amazon SWF is through either the AmazonSimpleWorkflowClient or AmazonSimpleWorkflowAsyncClient classes. The main difference between these is that the `*AsyncClient` class return Future objects for concurrent (asynchronous) programming.

```
AmazonSimpleWorkflowClient swf = AmazonSimpleWorkflowClientBuilder.defaultClient();
```

Building a Simple Amazon SWF Application

This topic will introduce you to programming Amazon SWF applications with the AWS SDK for Java, while presenting a few important concepts along the way.

About the example

The example project will create a workflow with a single activity that accepts workflow data passed through the AWS cloud (In the tradition of HelloWorld, it'll be the name of someone to greet) and then prints a greeting in response.

While this seems very simple on the surface, Amazon SWF applications consist of a number of parts working together:

- A **domain**, used as a logical container for your workflow execution data.
- One or more **workflows** which represent code components that define logical order of execution of your workflow's activities and child workflows.
- A **workflow worker**, also known as a *decider*, that polls for decision tasks and schedules activities or child workflows in response.
- One or more **activities**, each of which represents a unit of work in the workflow.
- An **activity worker** that polls for activity tasks and runs activity methods in response.
- One or more **task lists**, which are queues maintained by Amazon SWF used to issue requests to the workflow and activity workers. Tasks on a task list meant for workflow workers are called *decision tasks*. Those meant for activity workers are called *activity tasks*.
- A **workflow starter** that begins your workflow execution.

Behind the scenes, Amazon SWF orchestrates the operation of these components, coordinating their flow from the AWS cloud, passing data between them, handling timeouts and heartbeat notifications, and logging workflow execution history.

Prerequisites

Development environment

The development environment used in this tutorial consists of:

- The AWS SDK for Java.
- Apache Maven (3.3.1).
- JDK 1.7 or later. This tutorial was developed and tested using JDK 1.8.0.
- A good Java text editor (your choice).

Note
If you use a different build system than Maven, you can still create a project using the appropriate steps for your environment and use the the concepts provided here to follow along. More information about configuring and using the AWS SDK for Java with various build systems is provided in Getting Started (p. 3).
Likewise, but with more effort, the steps shown here can be implemented using any of the AWS SDKs with support for Amazon SWF.

All of the necessary external dependencies are included with the AWS SDK for Java, so there's nothing additional to download.

AWS access

To access Amazon Web Services (AWS), you must have an active AWS account. For information about signing up for AWS and creating an IAM user (recommended over using root account credentials), see Sign Up for AWS and Create an IAM User (p. 3).

This tutorial uses the terminal (command-line) to run the example code, and expects that you have your AWS credentials and configuration accessible to the SDK. The easiest way to do this is to use the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`. You should also set the `AWS_REGION` to the region you want to use.

For example, on Linux, macOS, or Unix, set the variables this way:

```
export AWS_ACCESS_KEY_ID=your_access_key_id
export AWS_SECRET_ACCESS_KEY=your_secret_access_key
export AWS_REGION=us-east-1
```

To set these variables on Windows, use these commands:

```
set AWS_ACCESS_KEY_ID=your_access_key_id
set AWS_SECRET_ACCESS_KEY=your_secret_access_key
set AWS_REGION=us-east-1
```

Important
Substitute your own access key, secret access key and region information for the example values shown here.
For more information about configuring your credentials for the SDK, see Set up AWS Credentials and Region for Development (p. 6).

Create a SWF project

1. Start a new project with Maven:

   ```
   mvn archetype:generate -DartifactId=helloswf \
   -DgroupId=example.swf.hello -DinteractiveMode=false
   ```

   This will create a new project with a standard maven project structure:

   ```
   helloswf
   ### pom.xml
   ### src
       ### main
       #   ### java
       #       ### example
       #           ### swf
       #               ### hello
       #                   ### App.java
       ### test
           ### ...
   ```

   You can ignore or delete the `test` directory and all it contains, we won't be using it for this tutorial. You can also delete `App.java`, since we'll be replacing it with new classes.
2. Edit the project's `pom.xml` file and add the **aws-java-sdk-simpleworkflow** module by adding a dependency for it within the `<dependencies>` block.

   ```
   ependencies>
   <dependency>
     <groupId>com.amazonaws</groupId>
     <artifactId>aws-java-sdk-simpleworkflow</artifactId>
     <version>1.11.78</version>
   </dependency>
   dependencies>
   ```
3. *Make sure that Maven builds your project with JDK 1.7+ support*. Add the following to your project (either before or after the `<dependencies>` block) in `pom.xml`:

   ```
   <build>
     <plugins>
       <plugin>
         <artifactId>maven-compiler-plugin</artifactId>
         <configuration>
             <source>1.7</source>
             <target>1.7</target>
   ```

```
      </configuration>
    </plugin>
  </plugins>
</build>
```

Code the project

The example project will consist of four separate applications, which we'll visit one by one:

- **HelloTypes.java**—contains the project's domain, activity and workflow type data, shared with the other components. It also handles registering these types with SWF.
- **ActivityWorker.java**—contains the activity worker, which polls for activity tasks and runs activities in response.
- **WorkflowWorker.java**—contains the workflow worker (decider), which polls for decision tasks and schedules new activities.
- **WorkflowStarter.java**—contains the workflow starter, which starts a new workflow execution, which will cause SWF to start generating decision and workflow tasks for your workers to consume.

Common steps for all source files

All of the files that you create to house your Java classes will have a few things in common. In the interest of time, these steps *will be implied every time you add a new file to the project*:

1. Create the file in the in the project's `src/main/java/example/swf/hello/` directory.
2. Add a `package` declaration to the beginning of each file to declare its namespace. The example project uses:

```
package aws.example.helloswf;
```

3. Add `import` declarations for the AmazonSimpleWorkflowClient class and for multiple classes in the `com.amazonaws.services.simpleworkflow.model` namespace. To simplify things, we'll use:

```
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
import com.amazonaws.services.simpleworkflow.model.*;
```

Register a domain, workflow and activity types

We'll begin by creating a new executeable class, `HelloTypes.java`. This file will contain shared data that different parts of your workflow will need to know about, such as the name and version of your activity and workflow types, the domain name and the task list name.

1. Open your text editor and create the file `HelloTypes.java`, adding a package declaration and imports according to the common steps (p. 143).
2. Declare the `HelloTypes` class and provide it with values to use for your registered activity and workflow types:

```
public class HelloTypes {
    public static final String DOMAIN = "HelloDomain";
    public static final String TASKLIST = "HelloTasklist";
    public static final String WORKFLOW = "HelloWorkflow";
    public static final String WORKFLOW_VERSION = "1.0";
    public static final String ACTIVITY = "HelloActivity";
    public static final String ACTIVITY_VERSION = "1.0";
}
```

These values will be used throughout the code.

3. After the String declarations, create an instance of the AmazonSimpleWorkflowClient class. This is the basic interface to the Amazon SWF methods provided by the AWS SDK for Java.

```
private static final AmazonSimpleWorkflow swf =
    AmazonSimpleWorkflowClientBuilder.defaultClient();
```

4. Add a new function to register a SWF domain. A *domain* is a logical container for a number of related SWF activity and workflow types. SWF components can only communicate with each other if they exist within the same domain.

```
public static void registerDomain() {
    try {
        System.out.println("** Registering the domain '" + DOMAIN + "'.");
        swf.registerDomain(new RegisterDomainRequest()
            .withName(DOMAIN)
            .withWorkflowExecutionRetentionPeriodInDays("1"));
    } catch (DomainAlreadyExistsException e) {
        System.out.println("** Domain already exists!");
    }
}
```

When you register a domain, you provide it with a *name* (any set of 1 – 256 characters excluding `:`, `/`, `|`, control characters or the literal string 'arn') and a *retention period*, which is the number of days that Amazon SWF will keep your workflow's execution history data after a workflow execution has completed. The maximum workflow execution retention period is 90 days. See RegisterDomainRequest for more information.

If a domain with that name already exists, a DomainAlreadyExistsException is raised. Because we're unconcerned if the domain has already been created, we can ignore the exception.

Note
This code demonstrates a common pattern when working with AWS SDK for Java methods, data for the method is supplied by a class in the `simpleworkflow.model` namespace, which you instantiate and populate using the chainable `.with*` methods.

5. Add a function to register a new activity type. An *activity* represents a unit of work in your workflow.

```
public static void registerActivityType() {
    try {
        System.out.println("** Registering the activity type '" + ACTIVITY +
            "-" + ACTIVITY_VERSION + "'.");
        swf.registerActivityType(new RegisterActivityTypeRequest()
            .withDomain(DOMAIN)
            .withName(ACTIVITY)
            .withVersion(ACTIVITY_VERSION)
            .withDefaultTaskList(new TaskList().withName(TASKLIST))
            .withDefaultTaskScheduleToStartTimeout("30")
            .withDefaultTaskStartToCloseTimeout("600")
            .withDefaultTaskScheduleToCloseTimeout("630")
            .withDefaultTaskHeartbeatTimeout("10"));
    } catch (TypeAlreadyExistsException e) {
        System.out.println("** Activity type already exists!");
    }
}
```

An activity type is identified by a *name* and a *version*, which are used to uniquely identify the activity from any others in the domain that it's registered in. Activities also contain a number of optional parameters, such as the default task-list used to receive tasks and data from SWF and a number of different timeouts that you can use to place constraints upon how long different parts of the activity execution can take. See RegisterActivityTypeRequest for more information.

> **Note**
> All timeout values are specified in *seconds*. See Amazon SWF Timeout Types for a full description of how timeouts affect your workflow executions.

If the activity type that you're trying to register already exists, an TypeAlreadyExistsException is raised.

6. Add a function to register a new workflow type. A *workflow*, also known as a *decider* represents the logic of your workflow's execution.

```
public static void registerWorkflowType() {
    try {
        System.out.println("** Registering the workflow type '" + WORKFLOW +
            "-" + WORKFLOW_VERSION + "'.");
        swf.registerWorkflowType(new RegisterWorkflowTypeRequest()
            .withDomain(DOMAIN)
            .withName(WORKFLOW)
            .withVersion(WORKFLOW_VERSION)
            .withDefaultChildPolicy(ChildPolicy.TERMINATE)
            .withDefaultTaskList(new TaskList().withName(TASKLIST))
            .withDefaultTaskStartToCloseTimeout("30"));
    } catch (TypeAlreadyExistsException e) {
        System.out.println("** Workflow type already exists!");
    }
}
```

Similar to activity types, workflow types are identified by a *name* and a *version* and also have configurable timeouts. See RegisterWorkflowTypeRequest for more information.

If the workflow type that you're trying to register already exists, an TypeAlreadyExistsException is raised.

7. Finally, make the class executable by providing it a `main` method, which will register the domain, the activity type, and the workflow type in turn:

```
public static void main(String[] args) {
    registerDomain();
    registerWorkflowType();
    registerActivityType();
}
```

You can build (p. 151) and run (p. 152) the application now to run the registration script, or continue with coding the activity and workflow workers. Once the domain, workflow and activity have been registered, you won't need to run this again—these types persist until you deprecate them yourself.

Implement the activity worker

An *activity* is the basic unit of work in a workflow. A workflow provides the logic, scheduling activities to be run (or other actions to be taken) in response to decision tasks. A typical workflow usually consists of a number of activities that can run synchronously, asynchronously, or a combination of both.

The *activity worker* is the bit of code that polls for activity tasks that are generated by Amazon SWF in response to workflow decisions. When it receives an activity task, it runs the corresponding activity and returns a success/failure response back to the workflow.

We'll implement a simple activity worker that drives a single activity.

1. Open your text editor and create the file `ActivityWorker.java`, adding a package declaration and imports according to the common steps (p. 143).
2. Add the `ActivityWorker` class to the file, and give it a data member to hold a SWF client that we'll use to interact with Amazon SWF:

```
public class ActivityWorker {
    private static final AmazonSimpleWorkflow swf =
        AmazonSimpleWorkflowClientBuilder.defaultClient();
}
```

3. Add the method that we'll use as an activity:

```
private static String sayHello(String input) throws Throwable {
    return "Hello, " + input + "!";
}
```

   The activity simply takes a string, combines it into a greeting and returns the result. Although there is little chance that this activity will raise an exception, it's a good idea to design activities that can raise an error if something goes wrong.
4. Add a `main` method that we'll use as the activity task polling method. We'll start it by adding some code to poll the task list for activity tasks:

```
public static void main(String[] args) {
    while (true) {
        System.out.println("Polling for an activity task from the tasklist '"
                + HelloTypes.TASKLIST + "' in the domain '" +
                HelloTypes.DOMAIN + "'.");

        ActivityTask task = swf.pollForActivityTask(
            new PollForActivityTaskRequest()
                .withDomain(HelloTypes.DOMAIN)
                .withTaskList(
                    new TaskList().withName(HelloTypes.TASKLIST)));

        String task_token = task.getTaskToken();
}
```

   The activity receives tasks from Amazon SWF by calling the SWF client's `pollForActivityTask` method, specifying the domain and task list to use in the passed-in PollForActivityTaskRequest.

   Once a task is received, we retrieve a unique identifier for it by calling the task's `getTaskToken` method.
5. Next, write some code to process the tasks that come in. Add the following to your `main` method, right after the code that polls for the task and retrieves its task token.

```
if (task_token != null) {
    String result = null;
    Throwable error = null;

    try {
        System.out.println("Executing the activity task with input '" +
                task.getInput() + "'.");
        result = sayHello(task.getInput());
    } catch (Throwable th) {
        error = th;
    }

    if (error == null) {
        System.out.println("The activity task succeeded with result '"
                + result + "'.");
        swf.respondActivityTaskCompleted(
            new RespondActivityTaskCompletedRequest()
                .withTaskToken(task_token)
                .withResult(result));
    } else {
        System.out.println("The activity task failed with the error '"
                + error.getClass().getSimpleName() + "'.");
        swf.respondActivityTaskFailed(
            new RespondActivityTaskFailedRequest()
                .withTaskToken(task_token)
                .withReason(error.getClass().getSimpleName())
                .withDetails(error.getMessage()));
    }
}
```

If the task token is not `null`, then we can start running the activity method (`sayHello`), providing it with the input data that was sent with the task.

If the task *succeeded* (no error was generated), then the worker responds to SWF by calling the SWF client's `respondActivityTaskCompleted` method with a RespondActivityTaskCompletedRequest object containing the task token and the activity's result data.

On the other hand, if the task *failed*, then we respond by calling the `respondActivityTaskFailed` method with a RespondActivityTaskFailedRequest object, passing it the task token and information about the error.

Note
This activity will not shut down gracefully if killed. Although it is beyond the scope of this tutorial, an alternative implementation of this activity worker is provided in the accompanying topic, Shutting Down Activity and Workflow Workers Gracefully (p. 156).

Implement the workflow worker

Your workflow logic resides in a piece of code known as a **workflow worker**. The workflow worker polls for decision tasks that are sent by Amazon SWF in the domain, and on the default tasklist, that the workflow type was registered with.

When the workflow worker receives a task, it makes some sort of decision (usually whether to schedule a new activity or not) and takes an appropriate action (such as scheduling the activity).

1. Open your text editor and create the file `WorkflowWorker.java`, adding a package declaration and imports according to the common steps (p. 143).
2. Add a few additional imports to the file:

```
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;
```

3. Declare the `WorkflowWorker` class, and create an instance of the AmazonSimpleWorkflowClient class used to access SWF methods.

```
public class WorkflowWorker {
    private static final AmazonSimpleWorkflow swf =
        AmazonSimpleWorkflowClientBuilder.defaultClient();
}
```

4. Add the `main` method. The method loops continuously, polling for decision tasks using the SWF client's `pollForDecisionTask` method. The PollForDecisionTaskRequest provides the details.

```
public static void main(String[] args) {
    PollForDecisionTaskRequest task_request =
        new PollForDecisionTaskRequest()
            .withDomain(HelloTypes.DOMAIN)
            .withTaskList(new TaskList().withName(HelloTypes.TASKLIST));

    while (true) {
        System.out.println(
                "Polling for a decision task from the tasklist '" +
                HelloTypes.TASKLIST + "' in the domain '" +
                HelloTypes.DOMAIN + "'.");

        DecisionTask task = swf.pollForDecisionTask(task_request);

        String taskToken = task.getTaskToken();
        if (taskToken != null) {
            try {
                executeDecisionTask(taskToken, task.getEvents());
            } catch (Throwable th) {
                th.printStackTrace();
            }
        }
    }
}
```

Once a task is received, we call its `getTaskToken` method, which returns a string that can be used to identify the task. If the returned token is not `null`, then we process it further in the `executeDecisionTask` method, passing it the task token and the list of HistoryEvent objects sent with the task.

5. Add the `executeDecisionTask` method, taking the task token (a `String`) and the `HistoryEvent` list.

```
private static void executeDecisionTask(String taskToken, List<HistoryEvent> events)
        throws Throwable {
    List<Decision> decisions = new ArrayList<Decision>();
    String workflow_input = null;
    int scheduled_activities = 0;
    int open_activities = 0;
    boolean activity_completed = false;
    String result = null;
}
```

We also set up some data members to keep track of things such as:

- A list of Decision objects used to report the results of processing the task.
- A String to hold workflow input provided by the "WorkflowExecutionStarted" event
- a count of the scheduled and open (running) activities to avoid scheduling the same activity when it has already been scheduled or is currently running.
- a boolean to indicate that the activity has completed.
- A String to hold the activity results, for returning it as our workflow result.

6. Next, add some code to `executeDecisionTask` to process the `HistoryEvent` objects that were sent with the task, based on the event type reported by the `getEventType` method.

```
System.out.println("Executing the decision task for the history events: [");
for (HistoryEvent event : events) {
    System.out.println("  " + event);
    switch(event.getEventType()) {
        case "WorkflowExecutionStarted":
            workflow_input =
                event.getWorkflowExecutionStartedEventAttributes()
```

```
                      .getInput();
            break;
        case "ActivityTaskScheduled":
            scheduled_activities++;
            break;
        case "ScheduleActivityTaskFailed":
            scheduled_activities--;
            break;
        case "ActivityTaskStarted":
            scheduled_activities--;
            open_activities++;
            break;
        case "ActivityTaskCompleted":
            open_activities--;
            activity_completed = true;
            result = event.getActivityTaskCompletedEventAttributes()
                          .getResult();
            break;
        case "ActivityTaskFailed":
            open_activities--;
            break;
        case "ActivityTaskTimedOut":
            open_activities--;
            break;
    }
}
System.out.println("]");
```

For the purposes of our workflow, we are most interested in:

- the "WorkflowExecutionStarted" event, which indicates that the workflow execution has started (typically meaning that you should run the first activity in the workflow), and that provides the initial input provided to the workflow. In this case, it's the name portion of our greeting, so it's saved in a String for use when scheduling the activity to run.
- the "ActivityTaskCompleted" event, which is sent once the scheduled activity is complete. The event data also includes the return value of the completed activity. Since we have only one activity, we'll use that value as the result of the entire workflow.

The other event types can be used if your workflow requires them. See the HistoryEvent class description for information about each event type.

Note
Strings in `switch` statements were introduced in Java 7. If you're using an earlier version of Java, you can make use of the EventType class to convert the `String` returned by `history_event.getType()` to an enum value and then back to a `String` if necessary:

```
EventType et = EventType.fromValue(event.getEventType());
```

7. After the `switch` statement, add more code to respond with an appropriate *decision* based on the task that was received.

```
if (activity_completed) {
    decisions.add(
        new Decision()
            .withDecisionType(DecisionType.CompleteWorkflowExecution)
            .withCompleteWorkflowExecutionDecisionAttributes(
                new CompleteWorkflowExecutionDecisionAttributes()
                    .withResult(result)));
} else {
    if (open_activities == 0 && scheduled_activities == 0) {

        ScheduleActivityTaskDecisionAttributes attrs =
            new ScheduleActivityTaskDecisionAttributes()
                .withActivityType(new ActivityType()
                    .withName(HelloTypes.ACTIVITY)
                    .withVersion(HelloTypes.ACTIVITY_VERSION))
                .withActivityId(UUID.randomUUID().toString())
                .withInput(workflow_input);

        decisions.add(
                new Decision()
                    .withDecisionType(DecisionType.ScheduleActivityTask)
                    .withScheduleActivityTaskDecisionAttributes(attrs));
    } else {
        // an instance of HelloActivity is already scheduled or running. Do nothing,
 another
        // task will be scheduled once the activity completes, fails or times out
    }
}

System.out.println("Exiting the decision task with the decisions " + decisions);
```

- If the activity hasn't been scheduled yet, we respond with a `ScheduleActivityTask` decision, which provides information in a ScheduleActivityTaskDecisionAttributes structure about the activity that Amazon SWF should schedule next, also including any data that Amazon SWF should send to the activity.
- If the activity was completed, then we consider the entire workflow completed and respond with a `CompletedWorkflowExecution` decision, filling in a CompleteWorkflowExecutionDecisionAttributes structure to provide details about the completed workflow. In this case, we return the result of the activity.

In either case, the decision information is added to the `Decision` list that was declared at the top of the method.

8. Complete the decision task by returning the list of `Decision` objects collected while processing the task. Add this code at the end of the `executeDecisionTask` method that we've been writing:

```
swf.respondDecisionTaskCompleted(
    new RespondDecisionTaskCompletedRequest()
        .withTaskToken(taskToken)
        .withDecisions(decisions));
```

The SWF client's `respondDecisionTaskCompleted` method takes the task token that identifies the task as well as the list of `Decision` objects.

Implement the workflow starter

Finally, we'll write some code to start the workflow execution.

1. Open your text editor and create the file `WorkflowStarter.java`, adding a package declaration and imports according to the common steps (p. 143).
2. Add the `WorkflowStarter` class:

```
public class WorkflowStarter {
    private static final AmazonSimpleWorkflow swf =
        AmazonSimpleWorkflowClientBuilder.defaultClient();
    public static final String WORKFLOW_EXECUTION = "HelloWorldWorkflowExecution";

    public static void main(String[] args) {
        String workflow_input = "Amazon SWF";
        if (args.length > 0) {
            workflow_input = args[0];
        }

        System.out.println("Starting the workflow execution '" + WORKFLOW_EXECUTION +
                "' with input '" + workflow_input + "'.");

        WorkflowType wf_type = new WorkflowType()
            .withName(HelloTypes.WORKFLOW)
            .withVersion(HelloTypes.WORKFLOW_VERSION);

        Run run = swf.startWorkflowExecution(new StartWorkflowExecutionRequest()
            .withDomain(HelloTypes.DOMAIN)
            .withWorkflowType(wf_type)
            .withWorkflowId(WORKFLOW_EXECUTION)
            .withInput(workflow_input)
            .withExecutionStartToCloseTimeout("90"));

        System.out.println("Workflow execution started with the run id '" +
                run.getRunId() + "'.");
    }
}
```

The `WorkflowStarter` class consists of a single method, `main`, which takes an optional argument passed on the command-line as input data for the workflow.

The SWF client method, `startWorkflowExecution`, takes a StartWorkflowExecutionRequest object as input. Here, in addition to specifying the domain and workflow type to run, we provide it with:

- a human-readable workflow execution name
- workflow input data (provided on the command-line in our example)
- a timeout value that represents how long, in seconds, that the entire workflow should take to run.

The Run object that `startWorkflowExecution` returns provides a *run ID*, a value that can be used to identify this particular workflow execution in Amazon SWF's history of your workflow executions.

Note
The run ID is generated by Amazon SWF, and is *not* the same as the workflow execution name that you pass in when starting the workflow execution.

Build the example

To build the example project with Maven, go to the `helloswf` directory and type:

```
mvn package
```

The resulting `helloswf-1.0.jar` will be generated in the `target` directory.

Run the example

The example consists of four separate executable classes, which are run independently of each other.

Note
If you are using a Linux, macOS, or Unix system, you can run all of them, one after another, in a single terminal window. If you are running Windows, you should open two additional command-line instances and navigate to the `helloswf` directory in each.

Setting the Java classpath

Although Maven has handled the dependencies for you, to run the example, you'll need to provide the AWS SDK library and its dependencies on your Java classpath. You can either set the `CLASSPATH` environment variable to the location of your AWS SDK libraries and the `third-party/lib` directory in the SDK, which includes necessary dependencies:

```
export CLASSPATH='target/helloswf-1.0.jar:/path/to/sdk/lib/*:/path/to/sdk/third-party/lib/
*'
java example.swf.hello.HelloTypes
```

or use the **`java`** command's `-cp` option to set the classpath while running each applications.

```
java -cp target/helloswf-1.0.jar:/path/to/sdk/lib/*:/path/to/sdk/third-party/lib/* \
  example.swf.hello.HelloTypes
```

The style that you use is up to you. If you had no trouble building the code, buth then try to run the examples and get a series of "NoClassDefFound" errors, it is likely because the classpath is set incorrectly.

Register the domain, workflow and activity types

Before running your workers and the workflow starter, you'll need to register the domain and your workflow and activity types. The code to do this was implemented in Register a domain, workflow and activity types (p. 143).

After building, and if you've set the CLASSPATH (p. 151), you can run the registration code by executing the command:

```
java aws.example.helloswf.HelloTypes
```

Start the activity and workflow workers

Now that the types have been registered, you can start the activity and workflow workers. These will continue to run and poll for tasks until they are killed, so you should either run them in separate terminal windows, or, if you're running on Linux, macOS, or Unix you can use the `&` operator to cause each of them to spawn a separate process when run.

```
java aws.example.helloswf.ActivityWorker &
java aws.example.helloswf.WorkflowWorker &
```

If you're running these commands in separate windows, omit the final `&` operator from each line.

Start the workflow execution

Now that your activity and workflow workers are polling, you can start the workflow execution. This process will run until the workflow returns a completed status. You should run it in a new terminal window (unless you ran your workers as new spawned processes by using the `&` operator).

```
java aws.example.helloswf.WorkflowStarter
```

Note
If you want to provide your own input data, which will be passed first to the workflow and then to the activity, add it to the command-line. For example:

```
java aws.example.helloswf.WorkflowStarter "Thelonious"
```

Once you begin the workflow execution, you should start seeing output delivered by both workers and by the workflow execution itself. When the workflow finally completes, its output will be printed to the screen.

Complete source for this example

You can browse the complete source for this example on Github in the *aws-java-developer-guide* repository.

For more information

- The workers presented here can result in lost tasks if they are shutdown while a workflow poll is still going on. To find out how to shut down workers gracefully, see Shutting Down Activity and Workflow Workers Gracefully (p. 156).
- To learn more about Amazon SWF, visit the Amazon SWF home page or view the Amazon SWF Developer Guide.
- You can use the AWS Flow Framework for Java to write more complex workflows in an elegant Java style using annotations. To learn more, see the AWS Flow Framework for Java Developer Guide.

Lambda Tasks

As an alternative to, or in conjunction with, Amazon SWF activities, you can use Lambda functions to represent units of work in your workflows, and schedule them similarly to activities.

This topic focuses on how to implement Amazon SWF Lambda tasks using the AWS SDK for Java. For more information about Lambda tasks in general, see AWS Lambda Tasks in the *Amazon SWF Developer Guide*.

Set up a cross-service IAM role to run your Lambda function

Before Amazon SWF can run your Lambda function, you need to set up an IAM role to give Amazon SWF permission to run Lambda functions on your behalf. For complete information about how to do this, see AWS Lambda Tasks.

You will need the Amazon Resource Name (ARN) of this IAM role when you register a workflow that will use Lambda tasks.

Create a Lambda function

You can write Lambda functions in a number of different languages, including Java. For complete information about how to author, deploy and use Lambda functions, see the AWS Lambda Developer Guide.

Note
It doesn't matter what language you use to write your Lambda function, it can be scheduled and run by *any*Amazon SWF workflow, regardless of the language that your workflow code is written in. Amazon SWF handles the details of running the function and passing data to and from it.

Here's a simple Lambda function that could be used in place of the activity in Building a Simple Amazon SWF Application (p. 140).

- This version is written in JavaScript, which can be entered directly using the AWS Management Console:

```
exports.handler = function(event, context) {
    context.succeed("Hello, " + event.who + "!");
};
```

- Here is the same function written in Java, which you could also deploy and run on Lambda:

```
package example.swf.hellolambda;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.util.json.JSONException;
import com.amazonaws.util.json.JSONObject;

public class SwfHelloLambdaFunction implements RequestHandler<Object, Object> {
    @Override
    public Object handleRequest(Object input, Context context) {
        String who = "Amazon SWF";
        if (input != null) {
            JSONObject jso = null;
            try {
                jso = new JSONObject(input.toString());
                who = jso.getString("who");
            } catch (JSONException e) {
                e.printStackTrace();
            }
        }
        return ("Hello, " + who + "!");
    }
}
```

Note
To learn more about deploying Java functions to Lambda, see Creating a Deployment Package (Java) in the *AWS Lambda Developer Guide*. You will also want to look at the section titled Programming Model for Authoring Lambda Functions in Java.

Lambda functions take an *event* or *input* object as the first parameter, and a *context* object as the second, which provides information about the request to run the Lambda function. This particular function expects input to be in JSON, with a `who` field set to the name used to create the greeting.

Register a workflow for use with Lambda

For a workflow to schedule a Lambda function, you must provide the name of the IAM role that provides Amazon SWF with permission to invoke Lambda functions. You can set this during workflow registration by using the `withDefaultLambdaRole` or `setDefaultLambdaRole` methods of RegisterWorkflowTypeRequest.

```
System.out.println("** Registering the workflow type '" + WORKFLOW + "-" + WORKFLOW_VERSION
        + "'.");
try {
    swf.registerWorkflowType(new RegisterWorkflowTypeRequest()
        .withDomain(DOMAIN)
        .withName(WORKFLOW)
        .withDefaultLambdaRole(lambda_role_arn)
        .withVersion(WORKFLOW_VERSION)
        .withDefaultChildPolicy(ChildPolicy.TERMINATE)
        .withDefaultTaskList(new TaskList().withName(TASKLIST))
        .withDefaultTaskStartToCloseTimeout("30"));
}
catch (TypeAlreadyExistsException e) {
```

Schedule a Lambda task

Schedule a Lambda task is similar to scheduling an activity. You provide a Decision with a `ScheduleLambdaFunction` DecisionType and with ScheduleLambdaFunctionDecisionAttributes.

```
running_functions == 0 && scheduled_functions == 0) {
AWSLambda lam = AWSLambdaClientBuilder.defaultClient();
GetFunctionConfigurationResult function_config =
    lam.getFunctionConfiguration(
            new GetFunctionConfigurationRequest()
                .withFunctionName("HelloFunction"));
String function_arn = function_config.getFunctionArn();

ScheduleLambdaFunctionDecisionAttributes attrs =
    new ScheduleLambdaFunctionDecisionAttributes()
        .withId("HelloFunction (Lambda task example)")
        .withName(function_arn)
        .withInput(workflow_input);

decisions.add(
```

In the `ScheduleLambdaFuntionDecisionAttributes`, you must supply a *name*, which is the ARN of the Lambda function to call, and an *id*, which is the name that Amazon SWF will use to identify the Lambda function in history logs.

You can also provide optional *input* for the Lambda function and set its *start to close timeout* value, which is the number of seconds that the Lambda function is allowed to run before generating a `LambdaFunctionTimedOut` event.

Note
This code uses the AWSLambdaClient to retrieve the ARN of the Lambda function, given the function name. You can use this technique to avoid hard-coding the full ARN (which includes your AWS account ID) in your code.

Handle Lambda function events in your decider

Lambda tasks will generate a number of events that you can take action on when polling for decision tasks in your workflow worker, corresponding to the lifecycle of your Lambda task, with EventType values such as `LambdaFunctionScheduled`, `LambdaFunctionStarted`, and `LambdaFunctionCompleted`. If the Lambda function fails, or takes longer to run than its set timeout value, you will receive either a `LambdaFunctionFailed` or `LambdaFunctionTimedOut` event type, respectively.

```
boolean function_completed = false;
String result = null;

System.out.println("Executing the decision task for the history events: [");
for (HistoryEvent event : events) {
    System.out.println("  " + event);
    EventType event_type = EventType.fromValue(event.getEventType());
    switch(event_type) {
    case WorkflowExecutionStarted:
        workflow_input =
            event.getWorkflowExecutionStartedEventAttributes()
                 .getInput();
        break;
    case LambdaFunctionScheduled:
        scheduled_functions++;
        break;
    case ScheduleLambdaFunctionFailed:
        scheduled_functions--;
        break;
    case LambdaFunctionStarted:
        scheduled_functions--;
        running_functions++;
        break;
    case LambdaFunctionCompleted:
        running_functions--;
        function_completed = true;
        result = event.getLambdaFunctionCompletedEventAttributes()
                      .getResult();
        break;
    case LambdaFunctionFailed:
        running_functions--;
        break;
    case LambdaFunctionTimedOut:
        running_functions--;
        break;
```

Receive output from your Lambda function

When you receive a `LambdaFunctionCompleted` EventType, you can retrieve your Lambda function's return value by first calling `getLambdaFunctionCompletedEventAttributes` on the HistoryEvent to get a LambdaFunctionCompletedEventAttributes object, and then calling its `getResult` method to retrieve the output of the Lambda function:

```
 LambdaFunctionCompleted:
running_functions--;
```

Complete source for this example

You can browse the *complete source :github:`<awsdocs/aws-java-developer-guide/tree/master/doc_source/snippets/helloswf_lambda/>* for this example on Github in the *aws-java-developer-guide* repository.

Shutting Down Activity and Workflow Workers Gracefully

The Building a Simple Amazon SWF Application (p. 140) topic provided a complete implementation of a simple workflow application consisting of a registration application, an activity and workflow worker, and a workflow starter.

Worker classes are designed to run continuously, polling for tasks sent by Amazon SWF in order to run activities or return decisions. Once a poll request is made, Amazon SWF records the poller and will attempt to assign a task to it.

If the workflow worker is terminated during a long poll, Amazon SWF may still try to send a task to the terminated worker, resulting in a lost task (until the task times out).

One way to handle this situation is to wait for all long poll requests to return before the worker terminates.

In this topic, we'll rewrite the activity worker from `helloswf`, using Java's shutdown hooks to attempt a graceful shutdown of the activity worker.

Here is the complete code:

```
package aws.example.helloswf;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
import com.amazonaws.services.simpleworkflow.model.ActivityTask;
import com.amazonaws.services.simpleworkflow.model.PollForActivityTaskRequest;
import com.amazonaws.services.simpleworkflow.model.RespondActivityTaskCompletedRequest;
import com.amazonaws.services.simpleworkflow.model.RespondActivityTaskFailedRequest;
import com.amazonaws.services.simpleworkflow.model.TaskList;

public class ActivityWorkerWithGracefulShutdown {

    private static final AmazonSimpleWorkflow swf =
        AmazonSimpleWorkflowClientBuilder.defaultClient();
    private static CountDownLatch waitForTermination = new CountDownLatch(1);
    private static volatile boolean terminate = false;

    private static String executeActivityTask(String input) throws Throwable {
        return "Hello, " + input + "!";
    }

    public static void main(String[] args) {
        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                try {
                    terminate = true;
                    System.out.println("Waiting for the current poll request" +
                            " to return before shutting down.");
                    waitForTermination.await(60, TimeUnit.SECONDS);
                }
                catch (InterruptedException e) {
                    // ignore
                }
            }
        });
        try {
            pollAndExecute();
        }
        finally {
            waitForTermination.countDown();
        }
    }

    public static void pollAndExecute() {
        while (!terminate) {
            System.out.println("Polling for an activity task from the tasklist '"
                    + HelloTypes.TASKLIST + "' in the domain '" +
                    HelloTypes.DOMAIN + "'.");

            ActivityTask task = swf.pollForActivityTask(new PollForActivityTaskRequest()
                .withDomain(HelloTypes.DOMAIN)
                .withTaskList(new TaskList().withName(HelloTypes.TASKLIST)));

            String taskToken = task.getTaskToken();

            if (taskToken != null) {
                String result = null;
                Throwable error = null;

                try {
                    System.out.println("Executing the activity task with input '"
                            + task.getInput() + "'.");
                    result = executeActivityTask(task.getInput());
                }
                catch (Throwable th) {
                    error = th;
                }

                if (error == null) {
                    System.out.println("The activity task succeeded with result '"
                            + result + "'.");
                    swf.respondActivityTaskCompleted(
                        new RespondActivityTaskCompletedRequest()
                            .withTaskToken(taskToken)
                            .withResult(result));
                }
                else {
                    System.out.println("The activity task failed with the error '"
                            + error.getClass().getSimpleName() + "'.");
                    swf.respondActivityTaskFailed(
                        new RespondActivityTaskFailedRequest()
                            .withTaskToken(taskToken)
                            .withReason(error.getClass().getSimpleName())
                            .withDetails(error.getMessage()));
                }
            }
        }
    }
}
```

In this version, the polling code that was in the `main` function in the original version has been moved into its own method, `pollAndExecute`.

The `main` function now uses a CountDownLatch in conjunction with a shutdown hook to cause the thread to wait for up to 60 seconds after its termination is requested before letting the thread shut down.

Registering Domains

Every workflow and activity in Amazon SWF needs a *domain* to run in.

To register an Amazon SWF domain

1. Create a new RegisterDomainRequest object, providing it with at least the domain name and workflow execution retention period (these parameters are both required).
2. Call the AmazonSimpleWorkflowClient.registerDomain method with the *RegisterDomainRequest* object.
3. Catch the DomainAlreadyExistsException if the domain you're requesting already exists (in which case, no action is usually required).

The following code demonstrates this procedure:

```
public void register_swf_domain(AmazonSimpleWorkflowClient swf, String name)
{
    RegisterDomainRequest request = new RegisterDomainRequest().withName(name);
    request.setWorkflowExecutionRetentionPeriodInDays("10");
    try
    {
        swf.registerDomain(request);
    }
    catch (DomainAlreadyExistsException e)
    {
        System.out.println("Domain already exists!");
    }
}
```

Listing Domains

You can list the Amazon SWF domains associated with your account and AWS region by registration type.

To list Amazon SWF domains

1. Create a ListDomainsRequest object, and specify the registration status of the domains that you're interested in—this is required.
2. Call AmazonSimpleWorkflowClient.listDomains with the *ListDomainRequest* object. Results are provided in a DomainInfos object.
3. Call getDomainInfos on the returned object to get a list of DomainInfo objects.
4. Call getName on each *DomainInfo* object to get its name.

The following code demonstrates this procedure:

```
public void list_swf_domains(AmazonSimpleWorkflowClient swf)
{
    ListDomainsRequest request = new ListDomainsRequest();
    request.setRegistrationStatus("REGISTERED");
    DomainInfos domains = swf.listDomains(request);
    System.out.println("Current Domains:");
    for (DomainInfo di : domains.getDomainInfos())
    {
        System.out.println(" * " + di.getName());
    }
}
```

Document History

This topic describes important changes to the *AWS SDK for Java Developer Guide* over the course of its history.

This documentation was built on: Sep 24, 2017

Apr 14, 2017

Made a number of updates to the Amazon S3 Examples (p. 105) section, including new topics: Managing Amazon S3 Access Permissions for Buckets and Objects (p. 113) and Configuring an Amazon S3 Bucket as a Website (p. 125).

Apr 04, 2017

A new topic, Enabling Metrics for the AWS SDK for Java (p. 29) describes how to generate application and SDK performance metrics for the AWS SDK for Java.

Apr 03, 2017

Added new CloudWatch examples to the CloudWatch Examples (p. 36) section: Getting Metrics from CloudWatch (p. 37), Publishing Custom Metric Data (p. 38), Working with CloudWatch Alarms (p. 39), Using Alarm Actions in CloudWatch (p. 41), and Sending Events to CloudWatch (p. 42)

Mar 27, 2017

Added more Amazon EC2 examples to the Amazon EC2 Examples (p. 54) section: Managing Amazon EC2 Instances (p. 81), Using Elastic IP Addresses in Amazon EC2 (p. 85), Using Regions and Availability Zones (p. 87), Working with Amazon EC2 Key Pairs (p. 88), and Working with Security Groups in Amazon EC2 (p. 90).

Mar 21, 2017

Added a new set of IAM examples to the IAM Examples (p. 92) section: Managing IAM Access Keys (p. 93), Managing IAM Users (p. 96), Using IAM Account Aliases (p. 98), Working with IAM Policies (p. 99), and Working with IAM Server Certificates (p. 103)

Mar 13, 2017

Added three new topics to the Amazon SQS section: Enabling Long Polling for Amazon SQS Message Queues (p. 132), Setting Visibility Timeout in Amazon SQS (p. 133), and Using Dead Letter Queues in Amazon SQS (p. 135).

Jan 26, 2017

Added a new Amazon S3 topic, Using TransferManager for Amazon S3 Operations (p. 118), and a new Best Practices for AWS Development with the AWS SDK for Java (p. 11) topic in the Using the AWS SDK for Java (p. 11) section.

Jan 16, 2017

Added a new Amazon S3 topic, Managing Access to Amazon S3 Buckets Using Bucket Policies (p. 116), and two new Amazon SQS topics, Working with Amazon SQS Message Queues (p. 128) and Sending, Receiving, and Deleting Amazon SQS Messages (p. 130).

Dec 16, 2016

Added new example topics for DynamoDB: Working with Tables in DynamoDB (p. 44) and Working with Items in DynamoDB (p. 49).

Sep 26, 2016

The topics in the **Advanced** section have been moved into Using the AWS SDK for Java (p. 11), since they really are central to using the SDK.

Aug 25, 2016

A new topic, Creating Service Clients (p. 12), has been added to Using the AWS SDK for Java (p. 11), which demonstrates how to use *client builders* to simplify the creation of AWS service clients.

The Programming Examples (p. 34) section has been updated with new examples for S3 (p. 105) which are backed by a repository on GitHub that contains the complete example code.

May 02, 2016

A new topic, Asynchronous Programming (p. 21), has been added to the Using the AWS SDK for Java (p. 11) section, describing how to work with asynchronous client methods that return `Future` objects or that take an `AsyncHandler`.

Apr 26, 2016

The *SSL Certificate Requirements* topic has been removed, since it is no longer relevant. Support for SHA-1 signed certificates was deprecated in 2015 and the site that housed the test scripts has been removed.

Mar 14, 2016

Added a new topic to the Amazon SWF section: Lambda Tasks (p. 153), which describes how to implement a Amazon SWF workflow that calls Lambda functions as tasks as an alternative to using traditional Amazon SWF activities.

Mar 04, 2016

The Amazon SWF (p. 139) section has been updated with new content:

- Amazon SWF Basics (p. 139)– Provides basic information about how to include SWF in your projects.
- Building a Simple Amazon SWF Application (p. 140)– A new tutorial that provides step-by-step guidance for Java developers new to Amazon SWF.
- Shutting Down Activity and Workflow Workers Gracefully (p. 156)– Describes how you can gracefully shut down Amazon SWF worker classes using Java's concurrency classes.

Feb 23, 2016

The source for the *AWS SDK for Java Developer Guide* has been moved to aws-java-developer-guide.

Dec 28, 2015

Setting the JVM TTL for DNS Name Lookups (p. 29) has been moved from **Advanced** into Using the AWS SDK for Java (p. 11), and has been rewritten for clarity.

Using the SDK with Apache Maven (p. 8) has been updated with information about how to include the SDK's bill of materials (BOM) in your project.

Aug 04, 2015

SSL Certificate Requirements is a new topic in the Getting Started (p. 3) section that describes AWS' move to SHA256-signed certificates for SSL connections, and how to fix early 1.6 and previous Java environments to use these certificates, which are *required* for AWS access after September 30, 2015.

> **Note**
> Java 1.7+ is already capable of working with SHA256-signed certificates.

May 14, 2014

The introduction (p. 1) and getting started (p. 3) material has been heavily revised to support the new guide structure and now includes guidance about how to Set up AWS Credentials and Region for Development (p. 6).

The discussion of code samples (p. 34) has been moved into its own topic in the Additional Documentation and Resources (p. 1) section.

Information about how to view the SDK revision history (p. 2) has been moved into the introduction.

May 9, 2014

The overall structure of the AWS SDK for Java documentation has been simplified, and the Getting Started (p. 3) and Additional Documentation and Resources (p. 1) topics have been updated.

New topics have been added:

- Working with AWS Credentials (p. 14)– discusses the various ways that you can specify credentials for use with the AWS SDK for Java.
- Using IAM Roles to Grant Access to AWS Resources on Amazon EC2 (p. 58)– provides information about how to securely specify credentials for applications running on EC2 instances.

Sep 9, 2013

This topic, *Document History*, tracks changes to the *AWS SDK for Java Developer Guide*. It is intended as a companion to the release notes history.