AWS SDK for Java Developer Guide

Release 1.0

Amazon Web Services

Dec 19, 2016

Contents

CHAPTER 1

AWS 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.

1.1 Additional documentation and resources

In addition to this guide, there are a number of valuable online resources available for AWS SDK for Java developers:

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

1.2 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 Java project. The toolkit also supports the creation and uploading of Lambda functions, launching and monitoring Amazon EC2 instances, management of IAM users and security groups, a CloudFormation template editor and more.

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

1.3 Developing AWS applications for Android

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

1.4 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.

1.5 Building Java reference documentation for old SDK versions

The AWS SDK for Java Reference represents the most recent version of the SDK. If you're using an older SDK version, you may wish 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 old 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, OS X, 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:

```
mvn javadoc:javadoc
```

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

CHAPTER 2

Getting Started

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

2.1 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.

Tip: 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, 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. You can, however, delete its associated access key ID and create a new one.

Next, you should *set your credentials* in the AWS shared credentials file or in the environment.

Tip: 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.

2.2 Set up the AWS SDK for Java

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

2.2.1 Prerequisites

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

- a suitable *Java Development Environment*.
- An AWS account and access keys. For instructions, see *Sign Up for AWS and Create an IAM User*.
- 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 for Development*.

2.2.2 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* 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* 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.

2.2.3 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*.

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 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.

2.2.4 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.

2.2.5 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.

2.3 Set up AWS Credentials 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*.
- 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 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, OS X, 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, OS X, 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*.

2.4 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.

2.4.1 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 *com.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
```

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

2.4.2 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* or the *entire SDK*.

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>1.11.22</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

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>1.11.22</version>
  </dependency>
</dependencies>
```

2.4.3 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.

2.4.4 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
```

2.5 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:0.5.4.
↪RELEASE"
    }
}

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:1.10.47'
    }
}
```

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*.

CHAPTER 3

Using the SDK

This section provides important general information about programming with the AWS SDK for Java. Information in this section applies to all services that you might be using with the AWS SDK for Java.

For information that is specific to a particular service (EC2, SWF, etc.), see the *Programming Examples* section.

3.1 Creating Service Clients

To make requests to Amazon Web Services, you first create a service client object. The preferred way to do this is to use the service client builder. Each AWS service has a service interface that has 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.

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 obtain a builder, you can customize properties of the client 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 more readable code. After you configure all the properties you want, you can call the `build` method to create the client. Once a client is created, it is 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', 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();
```

3.1.1 Creating Async Clients

The AWS SDK for Java also has asynchronous (or async) clients for every service, except for Amazon Simple Storage Service. There is also 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 allows 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();
```

3.1.2 Default Client

Both the sync and async client builders have another factory method called `defaultClient`. This method creates a service client with the default configuration, using the default provider chain to load credentials and the AWS region. If either credentials or the region cannot be determined from the environment that the application is running in, the call to `defaultClient` will fail. See *Working with AWS Credentials* and *AWS Region Selection* for more information on how credentials and region are determined.

To create a default service client

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

3.1.3 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, you can 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
```

3.2 Working with AWS Credentials

To make requests to Amazon Web Services, you will need to supply AWS credentials to the AWS SDK for Java. There are a number of ways to do this:

- 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 either root account credentials, IAM credentials or temporary credentials retrieved from AWS STS.

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.

This topic provides information about how to load credentials for AWS using the AWS SDK for Java.

3.2.1 Using the Default Credential Provider Chain

When you initialize a new service client without supplying any arguments, the AWS SDK for Java will attempt to find AWS credentials 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` (this location may vary per platform), this credentials file is 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 hand-editing the file with a text editor. For information about the credentials file format, see *AWS Credentials File Format*.
4. **Amazon ECS container credentials** – loaded from the Amazon EC2 Container Service 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

AWS credentials must be set in *at least one* of the preceding locations in order to be used. For information about setting credentials, visit one of the following topics:

- For information about specifying credentials in the *environment* or in the default *credential profiles file*, see *Set up AWS Credentials for Development*.

- For information about setting Java *system properties*, see the System Properties tutorial on the official *Java Tutorials* website.
- For information about how to set up and use *instance profile credentials* for use with your EC2 instances, see *Using IAM Roles to Grant Access to AWS Resources on Amazon EC2*.

Setting an Alternate Credentials Profile

The SDK for Java will use the *default* profile by default but there are a couple of ways to customize which profile is sourced from the credentials file.

The AWS Profile environment variable can be used to change the profile loaded by the SDK.

For example, on Linux, OS X, 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 will affect credential loading for all other officially supported AWS SDKs and Tools (including the AWS CLI and the AWS CLI for PowerShell). If you want to only change the profile for a Java application, you can use the system property *aws.profile* instead. Please note that the environment variable takes precedence over the system property.

Setting an Alternate Credentials File Location

Although the SDK for Java will load AWS credentials automatically from the default credentials file location, you can also specify the location yourself by setting the `AWS_CREDENTIAL_PROFILES_FILE` environment variable with the full pathname to the credentials file.

This feature can be used to temporarily change the location where the SDK for Java looks for your credentials file (by setting this variable with the command-line, for example), or you can set the environment variable in your user or system environment to change it for the user or system-wide.

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, OS X, 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 create an AWS credentials file using the `aws configure` command, it 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 wish 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

Once credentials have been set, you can load them using the default credential provider chain.

To load credentials using the default credential provider chain

- Instantiate an AWS Service client without explicitly providing credentials to the builder. For example:

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

3.2.2 Specifying a Credential Provider or Provider Chain

If you want to specify a different credential provider than the *default* credential provider chain, you can specify it via the client builder.

To specify a specific credentials provider

- Provide an instance of a credentials provider or provider chain to a client builder that takes an AWSCredentialsProvider interface as input. For example, 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 the list of "All known implementing classes" in the reference topic for AWSCredentialsProvider.

Tip: You can use this technique to supply credential providers or provider chains that you create, by implementing your own credential provider that implements the *AWSCredentialsProvider* interface, or by sub-classing the AWSCredentialsProviderChain class.

3.2.3 Explicitly Specifying Credentials

If neither the default credential chain or a specific or custom provider or provider chain works for your code, you can set credentials explicitly by supplying them yourself. If you have 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, supplying it with the AWS access key and secret key you will use for the connection.
2. Create a AWSStaticCredentialsProvider with the AWSCredentials object.
3. Configure the client builder with the AWSStaticCredentialsProvider and build the client.

For 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*, 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();
```

3.2.4 See Also

- *Sign Up for AWS and Create an IAM User*
- *Set up AWS Credentials for Development*
- *Using IAM Roles to Grant Access to AWS Resources on Amazon EC2*

3.3 AWS Region Selection

Regions enable you to access AWS services that reside physically 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.

3.3.1 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. For example:

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

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

3.3.2 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. If you want to choose the endpoint yourself, see *Choosing a Specific Endpoint*.

To explicitly set a region, it is recommended to use the Regions enum which is a 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 is not in the `Regions` enum, you can set the region using a *string* that represents the name of the region. For example:

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

Note: Once a client has been built with the builder it is *immutable* and the region *cannot be changed*. If you are working with multiple AWS Regions for the same service then you should create multiple clients—one per region.

3.3.3 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");
```

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

3.3.4 Automatically Determining the AWS Region from the Environment

Important: This section applies only when using a *client builder* to access AWS services. AWS clients created 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, it's often desirable 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 will consult the default region provider chain to try and determine the region to use.

Default Region Provider Chain

The region lookup process is as follows:

1. Any explicit region set 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, then that region will be used to configure the client.

   Note: This environment variable is set by the Lambda container

3. The SDK will look at the AWS shared config file (usually located at `~/.aws/config`). If the *region* property is present, then the SDK will use 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 which profile is loaded by the SDK.
4. The SDK will attempt 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 will fail with an exception.

A common approach when developing AWS applications is to use the *shared config file* (described in *Using the Default Credential Provider Chain*) 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.

3.4 Exception Handling

Understanding how and when the AWS SDK for Java throws exceptions is important in order to build 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.

3.4.1 Why Unchecked Exceptions?

The AWS Java SDK uses runtime (or unchecked) exceptions instead of checked exceptions for a few 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 the following articles:

- 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)

3.4.2 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 thrown `AmazonServiceException`. For some cases, a subclass of `AmazonServiceException` will be 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 could not be successfully processed either 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 (i.e., a request with illegal values) or the AWS service's fault (i.e., an internal service error).

3.4.3 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. `AmazonClientException` exceptions are generally more severe than `AmazonServiceException` exceptions and indicate a major problem that is preventing the client from being able to make service calls to AWS services. For example, the AWS Java SDK will throw an `AmazonClientException` if no network connection is available when you try to call an operation on one of the clients.

3.5 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.

Since 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 methods: *Futures* and *callback methods*.

3.5.1 Java Futures

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

Call the Future's `isDone()` method to see if the service has provided a response object yet. Once the response is ready, you can get the response object by calling the Future's `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);
    }
}
```

3.5.2 Asynchronous Callbacks

In addition to using Java Futures to monitor the status of asynchronous requests, the SDK also allows you to implement a class that uses the AsyncHandler interface, which 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(".");
        }
    }
}
```

3.5.3 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 being able to execute 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 SDK 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 if you want more control over how the thread pools are managed.

For example, you could provide a `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 Simple Storage Service (Amazon S3). `TransferManager` manages asynchronous uploads and downloads, provides detailed progress reporting on transfers, and supports callbacks into different events.

3.6 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 a number of logging systems at runtime. Supported logging systems include the Java Logging Framework and Apache Log4j, among others. This topic explains how to use Log4j. You can learn more about Log4j on the Log4j page at the Apache website. You can use the SDK's logging functionality without making any changes to your application code.

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.

In order to use Log4j with the SDK, you need to download the Log4j jar from the Apache website. The jar is not included in the SDK. 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 do not have to be in the same directory.

The log4j.properties configuration file specifies properties such as logging level, where logging output is sent (such as 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 SDK.

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

3.6.1 Setting the Classpath

Both the Log4j jar and the log4j.properties file must be located on your classpath. If you are using Apache Ant, set the classpath in the `path` element in your Ant file. Here is an example 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 are using the Eclipse IDE, you can set the classpath by opening the menu and navigating to *Project* | *Properties* | *Java Build Path*.

3.6.2 Service-Specific Errors and Warnings

We recommend that you always leave the "com.amazonaws" logger hierarchy set to "WARN" in order 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, it will report it through a warning message to the logs. This will also ensure 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 will cause 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
```

3.6.3 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
```

3.6.4 Verbose Wire Logging

In some cases, it may be useful to see the exact requests and responses being sent and received by the AWS SDK for Java. This logging should not be enabled in production systems since 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 since 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
```

3.7 Client Networking Configuration

The AWS SDK for Java allows 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

3.7.1 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.

3.7.2 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 that can be retrieved or set, see the AWS SDK for Java 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 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 may 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 50,000ms.

  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**

  You can set the maximum retry count for retriable errors by using the ClientConfiguration.setMaxErrorRetry method.

3.7.3 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 OS configuration and capabilities. For example, most modern operating systems provide auto-tuning logic for TCP buffer sizes, which 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 OS to buffer more data in memory without requiring the remote server to acknowledge receipt of that information, so can be particularly useful when the network has high latency.

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

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

- TCP Tuning and Network Troubleshooting
- Host Tuning

3.8 Access Control Policies

AWS *access control policies* allow 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.

3.8.1 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());
```

3.8.2 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));
```

3.8.3 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()));
```

3.9 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 will cache 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 re-querying 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 is vitally important 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 60s. If you are using such a JVM and not using a security manager, then you can ignore the remainder of this topic.

3.9.1 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");
```

CHAPTER 4

Programming Examples

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

Tip: See *Additional documentation and resources* for more examples and additional resources available for AWS SDK for Java developers!

4.1 SDK Code Samples

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

4.1.1 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*, then 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.

4.1.2 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.

Tip: If you are browsing the sample code on GitHub, click the *Raw* button in the source code display when viewing the README.html file for a sample. In raw mode, the HTML will render as intended in your browser.

Prerequisites

Before running any of the AWS SDK for Java samples, you will need to set your AWS credentials in the environment or with the AWS CLI as specified in *Set up AWS Credentials for Development*. The samples use the default credential provider chain whenever possible, so by setting your credentials 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 are 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.
```

4.1.3 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 selecting *Preferences* from the *Window* menu in Eclipse, and then

selecting 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, point to *New*, and then click *Project*. The *New Project* wizard opens.
3. Expand the *AWS* category, then select *AWS Java Project*.
4. Click *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. Click *Finish*. The project is created and added to the *Project Explorer*.

To run the project

1. Select the sample `.java` file you want to run. For example, for the Amazon S3 sample, select `S3Sample.java`.
2. Select *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 click *Add Libraries*.
2. Select *AWS Java SDK*, and then click *Next* and follow the remaining on-screen instructions.

4.2 DynamoDB

This section provides examples of programming Amazon S3 using the AWS SDK for Java.

Note: Only the code that is necessary to demonstrate each technique is supplied here, but complete example code is available on GitHub, where you can download a single source file or you can clone the repository locally to get all examples, build and run them.

4.2.1 Working with Tables

Tables are the container 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*, which must be 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 both a partition and a sort (RANGE) key.

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

Tip: Amazon DynamoDB pricing is based on the provisioned throughput values that you set on your tables, so you should only reserve as much capacity as you expect you will need for your table.

Provisioned throughput can be modified at any time for a table, 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 will need to construct table attributes and a table schema, both of which serve to identify the primary key of your table. You must also supply initial provisioned throughput values and give the table a name.

Note: If a table already exists with the name that you've chosen, then an AmazonServiceException will be thrown.

Imports:

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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;
import com.amazonaws.services.dynamodbv2.model.ScalarAttributeType;
```

Creating 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 AmazonDynamoDBClient ddb = new AmazonDynamoDBClient();

try {
    CreateTableResult result = ddb.createTable(request);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

See the complete sample.

Creating a table with a composite primary key

To create a table with a composite primary key, add an additional AttributeDefinition and KeySchemaElement to the 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)))
    .withTableName(table_name);
```

See the complete sample.

Listing 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, then a ResourceNotFoundException exception will result.

Imports:

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.model.ListTablesResult;
import java.util.List;
```

Code:

```
final AmazonDynamoDBClient ddb = new AmazonDynamoDBClient();

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

        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);
    }
}
```

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

See the complete sample.

Describe a Table

You can describe (get information about) a table by calling the DynamoDB client's `describeTable` method.

Note: If the named table doesn't exist for your account and region, then a ResourceNotFoundException exception will result.

Imports:

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.
↪ProvisionedThroughputDescription;
import com.amazonaws.services.dynamodbv2.model.TableDescription;
import java.util.List;
```

Code:

```
final AmazonDynamoDBClient ddb = new AmazonDynamoDBClient();

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 update the provisioned throughput values for your table at any time by calling the DynamoDB client's `updateTable` method.

Note: If the named table doesn't exist for your account and region, then a ResourceNotFoundException exception will result.

Imports:

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.AmazonServiceException;

/**
 * Update a DynamoDB table (change provisioned throughput).
 *
 * Takes the name of the table to update, the read capacity and the write
```

Code:

```
ProvisionedThroughput table_throughput = new ProvisionedThroughput(
      read_capacity, write_capacity);

final AmazonDynamoDBClient ddb = new AmazonDynamoDBClient();

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

To delete a table, call the DynamoDB client's `deleteTable` method, passing it the table's name.

Note: If the named table doesn't exist for your account and region, then a ResourceNotFoundException exception will result.

Imports:

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

Code:

```
final AmazonDynamoDBClient ddb = new AmazonDynamoDBClient();

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

See the complete sample.

See Also

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

4.2.2 Working with Items

In DynamoDB, an item is a collection of *attributes*, which consist of 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 DynamoDB Developer Guide.

Retrieve (get) an item from a table

To get an item from a table, call the DynamoDB client's `getItem` method, passing it a GetItemRequest object with the table name and primary key value of the desired item. 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 associated with the item.

Imports:

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
```

```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
```

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 AmazonDynamoDBClient ddb = new AmazonDynamoDBClient();

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

To add a new item to a table, create a Map of key/value pairs that represent the attributes of the item. These must include values for the primary key field(s) of the table. If the item identified by the primary key already exists, then its fields will be *updated* by the request.

Note: If the named table doesn't exist for your account and region, then a ResourceNotFoundException

exception will result.

Imports:

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.ResourceNotFoundException;
import java.util.ArrayList;
import java.util.HashMap;
```

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 AmazonDynamoDBClient ddb = new AmazonDynamoDBClient();

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 DynamoDB client'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 passed-in primary key doesn't exist, then a ResourceNotFoundException exception will result.

Imports:

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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;
import java.util.HashMap;
```

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 AmazonDynamoDBClient ddb = new AmazonDynamoDBClient();

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.

See Also

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

4.2.3 Manage Tomcat Session State with DynamoDB

Tomcat applications often store session-state data in memory. However, this approach doesn't scale well; once the application grows beyond a single web server, the session state must be shared between servers. A common solution is to set up a dedicated session-state server with MySQL. 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, a NoSQL database store from Amazon Web Services (AWS), avoids these drawbacks by providing an effective solution for sharing session state across web servers.

Downloading the Session Manager

You can download the session manager from the aws/aws-dynamodb-session-tomcat project on GitHub. That project also hosts the session manager source code if you want to 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 need to 1) configure the Tomcat server to use the provider, and 2) set the security credentials of the provider so that it can access AWS.

Configuring 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, containing 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>
```

Configuring Your AWS Security Credentials

You can specify AWS security credentials for the session manager in multiple ways, and 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 no credentials are specified through the `Manager` element, `DefaultAWSCredentialsProviderChain` will keep 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).

Configuring with Elastic Beanstalk

If you're using the session manager in Elastic Beanstalk, you need to ensure your project has a `.ebextensions` directory at the top level of your output artifact structure. In that directory, place the following files:

- The `AmazonDynamoDBSessionManagerForTomcat-1.x.x.jar` file.
- A `context.xml` file as described previously to configure the session manager.
- A config 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 Elastic Beanstalk Developer Guide.

If you deploy to Elastic Beanstalk with the AWS Toolkit for Eclipse, then the session manager is set up for you if you go through the *New AWS Java Web Project* wizard and select DynamoDB for session management. The AWS Toolkit for Eclipse configures all the needed files, and puts them in the `.ebextensions` directory inside the `WebContent` directory of your project. If you have problems finding this directory, make 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 do not 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 an DynamoDB table called, 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 first running your application. 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.

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

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

Options Specified in context.xml

Below are the configuration attributes that you can use 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; if present, this option overrides any value set for the `Region` option. The regional endpoint of the DynamoDB service to use. For a list of available AWS regions, see Regions and Endpoints in the *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 does not 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.

4.3 Amazon EC2

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

4.3.1 Tutorial: Starting an EC2 Instance

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

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*.

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(createSecurityGroupRequest);
```

   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();

ipPermission.withIpRanges("111.111.111.111/32", "150.150.150.150/32")
            .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*.

withSecurityGroups One or more security groups. For more information, see *Create an Amazon EC2 Security Group*.

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.

4.3.2 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())
              .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*.

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 `GetS3Objbect.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.

Tip: The `-p` switch works only on Linux, OS X, 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.

4.3.3 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* 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*.

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.

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.

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* or the RequestSpotInstances class in the AWS SDK for Java 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*.

4.3.4 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* 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*.

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*, 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.

4.4 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.

4.4.1 (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.

4.4.2 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.

4.4.3 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*.

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.

4.4.4 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

4.5 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.

4.5.1 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*.

Dependencies

Basic Amazon SWF applications will require the following dependencies, which are included with the AWS SDK for Java:

- aws-java-sdk-1.10.*.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();
```

4.5.2 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 exectuion history.

Prerequisites

Development environment

The development environment used in this tutorial consists of:

- The AWS SDK for Java (v. 1.10.56 at the time of writing).
- 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*.

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*.

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, OS X, 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 for Development*.

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 to it by adding the following section within the `<dependencies>` block.

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-java-sdk-simpleworkflow</artifactId>
  <version>1.10.56</version>
</dependency>
```

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 example.swf.hello;
```

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;
```

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*.
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";
    }
```

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 =
```

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.

Tip: 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.

Tip: 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* and *run* 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*.

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)));
```

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.

Tip: 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*.

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*.
2. Add a few additional imports to the file:

```
import com.amazonaws.services.simpleworkflow.model.*;
import java.util.ArrayList;
import java.util.List;
```

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 =
                .withDecisions(decisions));
```

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.

```
    AmazonSimpleWorkflowClientBuilder.defaultClient();

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.

```
 * as the output of the workflow.
 */
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;
            .withTaskToken(taskToken)
```

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.

```
String result = null;

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;
    }
```

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.

```
System.out.println("]");

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
```

- 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:

```
System.out.println("Exiting the decision task with the decisions " +
↪decisions);

swf.respondDecisionTaskCompleted(
    new RespondDecisionTaskCompletedRequest()
```

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*.
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, OS X, 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*.

After building, and if you've *set the CLASSPATH*, you can run the registration code by executing the command:

```
java example.swf.hello.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, OS X, or Unix you can use the `&` operator to cause each of them to spawn a separate process when run.

```
java example.swf.hello.ActivityWorker &
java example.swf.hello.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 example.swf.hello.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 example.swf.hello.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*.
- 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.

4.5.3 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 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*.

- 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 + "!");
    }
}
```

Tip: To learn more about deploying Java functions to Lambda, see Creating a Deployment Package (Java) in the 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.

Tip: 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.

4.5.4 Shutting Down Activity and Workflow Workers Gracefully

The *Building a Simple Amazon SWF Application* 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 example.swf.hello;

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.

4.5.5 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!");
    }
}
```

4.5.6 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());
    }
}
```

4.6 Amazon S3

This section provides examples of programming Amazon S3 using the AWS SDK for Java.

Note: Only the code that is necessary to demonstrate each technique is supplied here, but complete example code is available on GitHub, where you can download a single source file or you can clone the repository locally to get all examples, build and run them.

4.6.1 Creating, Listing and Deleting 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 the Amazon S3 Developer Guide.

Best Practice

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.

Note: These code snippets assume that you understand the material in *Using the SDK* and have configured default AWS credentials using the information in *Set up AWS Credentials for Development*.

- *Create a bucket*

- *List buckets*
- *Delete a bucket*

Create a bucket

Use the AmazonS3 client's `createBucket` method. The new Bucket is returned.

Imports:

```
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.model.Bucket;
import com.amazonaws.services.s3.AmazonS3Client;
import com.amazonaws.AmazonServiceException;
```

Code:

```
final AmazonS3 s3 = new AmazonS3Client();
try {
    Bucket b = s3.createBucket(bucket_name);
}
catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
System.out.println("Done!");
```

See the complete example.

List buckets

Use the AmazonS3 client's `listBucket` method. If successful, a List of Bucket objects will be returned.

Imports:

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3Client;
import com.amazonaws.services.s3.model.Bucket;
import java.util.List;
```

Code:

```
final AmazonS3 s3 = new AmazonS3Client();
List<Bucket> buckets = s3.listBuckets();
System.out.println("Your Amazon S3 buckets:");
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 first, or an error will result. If you have a versioned bucket, then 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.

Removing objects from an unversioned bucket prior to deletion

To remove objects from an unversioned bucket prior to deletion, you can 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.AmazonS3Client;
import com.amazonaws.services.s3.model.ObjectListing;
import com.amazonaws.services.s3.model.S3ObjectSummary;
import java.util.Iterator;
```

Code:

```
final AmazonS3 s3 = new AmazonS3Client();
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.

Removing objects from a versioned bucket prior to deletion

If you are using a versioned bucket, you will 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.AmazonS3Client;
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 = new AmazonS3Client();
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.

Deleting an empty bucket

Once you've removed the objects from a bucket (including any versioned objects), you can delete the bucket itself, use the AmazonS3 client's `deleteBucket` method.

Imports:

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3Client;
```

Code:

```
final AmazonS3 s3 = new AmazonS3Client();
try {
    s3.deleteBucket(bucket_name);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

See the complete example.

4.6.2 Operations on Objects

An Amazon S3 object represents a *file*, or collection of data. Every object must reside within a *bucket*.

Note: These code snippets assume that you understand the material in *Using the SDK* and have configured default AWS credentials using the information in *Set up AWS Credentials for Development*.

- *Upload an object*
- *List objects*
- *Download an object*
- *Copying, moving or renaming objects*
- *Delete an object*
- *Deleting multiple objects at once*

Upload an object

Use the AmazonS3 client's `putObject` method, supplying it with 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.AmazonS3Client;
import com.amazonaws.AmazonServiceException;
```

Code:

```
final AmazonS3 s3 = new AmazonS3Client();
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 it with 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.AmazonS3;
import com.amazonaws.services.s3.AmazonS3Client;
import com.amazonaws.services.s3.model.ObjectListing;
```

```
import com.amazonaws.services.s3.model.S3ObjectSummary;
import java.util.List;
```

Code:

```
final AmazonS3 s3 = new AmazonS3Client();
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 will return 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.AmazonS3Client;
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 = new AmazonS3Client();
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.

Copying, moving or renaming 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* 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.AmazonS3Client;
import com.amazonaws.AmazonServiceException;
```

Code:

```
final AmazonS3 s3 = new AmazonS3Client();
try {
    s3.deleteObject(bucket_name, object_key);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

See the complete example.

Deleting 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.AmazonS3Client;
import com.amazonaws.services.s3.model.DeleteObjectsRequest;
```

Code:

```
final AmazonS3 s3 = new AmazonS3Client();
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.

CHAPTER 5

Document History

This topic describes important changes to the AWS Java Developer Guide over the course of its history.

Last documentation update: Dec 19, 2016

Dec 16, 2016 Added new example topics for DynamoDB: *Working with Tables* and *Working with Items*.

Sep 26, 2016 The topics in the **Advanced** section have been moved into *Using the SDK*, since they really are central to using the SDK.

Aug 25, 2016 A new topic, *Creating Service Clients*, has been added to *Using the SDK*, which demonstrates how to use *client builders* to simplify the creation of AWS service clients.

The *Programming Examples* section has been updated with *new examples for S3* which are backed by a repository on GitHub that contains the complete example code.

May 02, 2016 A new topic, *Asynchronous Programming*, has been added to the *Using the SDK* 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*, 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* section has been updated with new content:

- *Amazon SWF Basics* – Provides basic information about how to include SWF in your projects.
- *Building a Simple Amazon SWF Application* – A new tutorial that provides step-by-step guidance for Java developers new to Amazon SWF.
- *Shutting Down Activity and Workflow Workers Gracefully* – Describes how you can gracefully shut down Amazon SWF worker classes using Java's concurrency classes.

Feb 23, 2016 The source for the AWS Java Developer Guide has been moved to aws-java-developer-guide.

Dec 28, 2015 *Setting the JVM TTL for DNS Name Lookups* has been moved from **Advanced** into *Using the SDK*, and has been rewritten for clarity.

Using the SDK with Apache Maven 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* 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* and *getting started* material has been heavily revised to support the new guide structure and now includes guidance about how to *Set up AWS Credentials for Development*.

The discussion of *code samples* has been moved into its own topic in the *Additional documentation and resources* section.

Information about how to *view the SDK revision history* 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* and *Additional documentation and resources* topics have been updated.

New topics have been added:

- *Working with AWS Credentials* – 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* – 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 Java Developer Guide. It is intended as a companion to the release notes history.

CHAPTER 6

About Amazon Web Services

Amazon Web Services (AWS) is a collection of digital infrastructure services that developers can leverage when developing their applications. The services include computing, storage, database, and application synchronization (messaging and queuing). AWS uses a pay-as-you-go service model: you are charged only for the services that you—or your applications—use. For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see Use the AWS Free Tier. To obtain an AWS account, visit the AWS home page and click **Create a Free Account**.