AWS SDK for Java

Developer Guide

Version v1.0.0

AWS SDK for Java: Developer Guide

Table of Contents

What is the AWS SDK for Java?

The AWS SDK for Java provides a Java API for AWS infrastructure services. Using the SDK for Java, you can build applications that work with Amazon Simple Storage Service (Amazon S3), Amazon Elastic Compute Cloud (Amazon EC2), Amazon SimpleDB, and more.

New AWS services are occasionally added to the AWS SDK for Java. For a complete list of the services that are supported by the SDK for Java, see Supported Services on the AWS SDK for Java home page.

Topics

About the AWS SDK for Java

The AWS SDK for Java includes:

AWS Java Library
Build Java applications for AWS using an API that takes the complexity out of coding directly against a web service interface. The library (`aws-java-sdk-`*`version`*`.jar`) provides an API that hides much of the lower-level plumbing including authentication, request retries, and error handling.

Code Samples
The SDK for Java provides a growing number of practical code samples that demonstrate how to use the SDK to build AWS applications. For more information, see SDK for Java Code Samples (p. 66).

Eclipse IDE support
You can use the AWS Toolkit for Eclipse to add the AWS SDK for Java to an existing project or to create a new SDK for Java project.

Browse the Source at GitHub

The AWS SDK for Java is open source, provided to you under the Apache License. You can find the full source code for the AWS SDK for Java and its packaged samples at: http://github.com/aws/aws-sdk-java.

Developing AWS Applications for Android

If you are an Android developer, Amazon Web Services publishes a separate SDK specifically for Android development. For more information, go to the documentation for the AWS SDK for Android.

What's in this Guide?

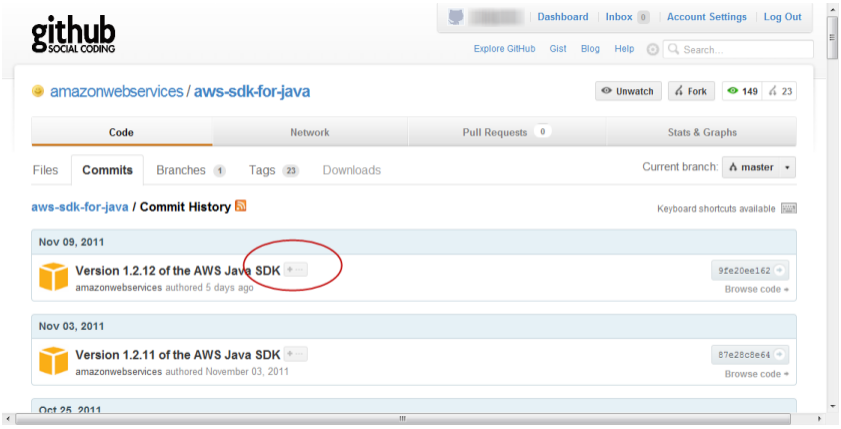
This is the AWS SDK for Java Developer Guide, which aims to provide you with information about how to install, set up, and use the SDK for Java to program applications in Java that can make full use of the services offered by Amazon Web Services.

Here is a guide to the contents:

Viewing the Revision History for the SDK for Java

The AWS SDK for Java is regularly updated to support new services and new service features. To see what changed with a given release, you can check the release notes history.

Each release of the AWS SDK for Java is also published to GitHub. The comments in the commit history provide information about what changed in each commit. To view the comments associated with a commit, click the plus sign next to that commit.



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. Also, to make AWS more approachable as a platform for prototyping and experimentation, AWS offers a free usage tier. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, go to Test-Driving AWS in the Free Usage Tier. To obtain an AWS account, go to the AWS home page and click Sign Up.

Getting Started

This section provides information about how to install, set up and use the SDK for Java. If you have never used the SDK for Java before, you should start here.

Topics

Get an AWS Account and Your AWS Credentials

To access AWS, you will need to sign up for an AWS account.

To sign up for an AWS account

1. Go to http://aws.amazon.com, and then 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 the phone keypad.

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

To get your access key ID and secret access key

Access keys consist of an access key ID and secret access key, which are used to sign programmatic requests that you make to AWS. If you don't have access keys, you can create them by using the AWS Management Console. We recommend that you use IAM access keys instead of AWS root account access keys. IAM lets you securely control access to AWS services and resources in your AWS account.

Note
To create access keys, you must have permissions to perform the required IAM actions. For more information, see Granting IAM User Permission to Manage Password Policy and Credentials in *Using IAM*.

1. Go to the IAM console.
2. From the navigation menu, click **Users**.
3. Select your IAM user name.
4. Click **User Actions**, and then click **Manage Access Keys**.
5. Click **Create Access Key**.

   Your keys will look something like this:

   - Access key ID example: AKIAIOSFODNN7EXAMPLE
   - Secret access key example: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY

6. Click **Download Credentials**, and store the keys in a secure location.

   Your secret key will no longer be available through the AWS Management Console; you will have the only copy. Keep it confidential in order to protect your account, and never email it. Do not share it outside your organization, even if an inquiry appears to come from AWS or Amazon.com. No one who legitimately represents Amazon will ever ask you for your secret key.

Related topics

- What Is IAM? in *Using IAM*
- AWS Security Credentials in *AWS General Reference*

Install 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://developers.sun.com/downloads/. The SDK also uses the Apache Commons (Code, HTTP Client, and Logging), and Saxon HE third-party packages, which are included in the third-party directory of the SDK.

If you use Eclipse, the AWS Toolkit for Eclipse provides support for the AWS SDK for Java as well as additional management features. For more information on installing the AWS Toolkit for Eclipse, see http://aws.amazon.com/eclipse/. For more information about using the Toolkit for AWS development, see the AWS Toolkit for Eclipse Getting Started Guide.

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.

Install the AWS SDK for Java

Once you have set up Java, you should download and install the the AWS SDK for Java.

If you intend to use the SDK for Java with the Eclipse IDE, you should install the AWS Toolkit for Eclipse, which automatically includes the AWS SDK for Java. For information about how to obtain, install and set up the Toolkit for Eclipse, see Setting Up the AWS Toolkit for Eclipse in the *AWS Toolkit for Eclipse Getting Started Guide*.

If you intend to build using Ant and the command-line, or will be using an IDE other than Eclipse, you can set up the Java SDK as shown here.

To download and install the AWS SDK for Java

1. Download the AWS SDK for Java from the SDK web page at http://aws.amazon.com/sdkforjava.
2. After downloading the SDK, extract the contents into a directory.

Set up your AWS Credentials for Use with the SDK for Java

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

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

- To set credentials for your local user (login), use the AWS CLI to set credentials with the following command:

```
aws cli configure
```

  The CLI will store credentials that are specified this way in a local file, typically `~/.aws/config`.
- To set credentials for a particular terminal (command-line) session, set the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables.

  To set these variables in 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 in Windows, use **set**:

```
set AWS_ACCESS_KEY_ID=your_access_key_id
set AWS_SECRET_ACCESS_KEY=your_secret_access_key
```

- To set credentials for an EC2 instance, you should specify an IAM role and then give your EC2 instance access to that role as shown in Using IAM Roles for EC2 Instances (p. 29).

Once you have set your AWS credentials using one of these methods, they can be loaded automatically by the SDK for Java by using the default credential provider chain. For complete information about how to load and use AWS credentials in your SDK for Java applications, see Providing Credentials (p. 8).

Note
You can override the default location of the AWS credential file by setting the AWS_CREDENTIAL_FILE environment variable. For more information, see Providing Credentials (p. 8).

Explore the SDK for Java Code Samples

The SDK for Java is packaged with a number of code samples, which you can browse on your machine or view on GitHub. For more information about the samples that are provided, see SDK for Java Code Samples (p. 66).

Where to Go from Here

To learn more about the SDK for Java, you should start with AWS SDK for Java Basics (p. 8), which provides essential information about how to use the SDK for Java with any AWS service.

The Using AWS Services (p. 20) section provides instructions about how to use the SDK for Java to perform common actions with various AWS services. Many of these are accompanied by code examples that demonstrate the techniques that are discussed.

The Additional Resources (p. 65) section has pointers to other resources to assist you with programming AWS using the SDK for Java.

AWS SDK for Java Basics

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 SDK for Java.

For information that is specific to a particular service (EC2, SWF, etc.), see the Using AWS Services (p. 20) section.

Topics

Providing AWS Credentials in the AWS SDK for Java

To make requests to Amazon Web Services, you will need to supply AWS credentials to the 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.

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

Topics

Using the Default Credential Provider Chain

When you initialize a new service client without supplying any arguments, the 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 SDK for Java uses the EnvironmentVariableCredentialsProvider class to load these credentials.
2. **Java System Properties** – `aws.accessKeyId` and `aws.secretKey`. The SDK for Java uses the SystemPropertiesCredentialsProvider to load these credentials.
3. **The default credential profiles file** – typically located at `~/.aws/config` (this location may vary per platform), this credentials file is shared by many of the AWS SDKs and by the AWS CLI. The SDK for Java uses the ProfileCredentialsProvider to load these credentials.
4. **Instance profile credentials** – these credentials can be used on EC2 instances, and are delivered through the Amazon EC2 metadata service. The SDK for Java uses the InstanceProfileCredentialsProvider to load these credentials.

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 your AWS Credentials for Use with the SDK for Java (p. 6).
- 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 for EC2 Instances (p. 29).

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_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_FILE environment variable to the location of your AWS credentials file.

  On Linux, OS X, or Unix, use `export`:

  ```
  export AWS_CREDENTIAL_FILE=path/to/credentials_file
  ```

  On Window, use `set`:

  ```
  set AWS_CREDENTIAL_FILE=path/to/credentials_file
  ```

Loading Credentials

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

To load credentials using the default credential provider chain

- Instantiate an AWS Service client using the default constructor. For example:

```
AmazonS3 s3Client = new AmazonS3Client();
```

- Alternately, you can specify a new DefaultAWSCredentialsProviderChain in the client's constructor. The following line of code is effectively equivalent to the preceding example:

```
AmazonS3 s3Client = new AmazonS3Client(new DefaultAWSCredentialsProviderCha
in());
```

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 in the client constructor.

To specify a specific credentials provider

- Provide an instance of a credentials provider or provider chain to a service client constructor that takes an AWSCredentialsProvider interface as input. For example, to use *environment* credentials specifically:

```
AmazonS3 s3Client = new AmazonS3Client(new EnvironmentVariableCredentialsPro
vider());
```

For the full list of 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.

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.

In most cases, it's better to create your own credential provider and to use that (as shown in Specifying a Credential Provider or Provider Chain (p. 10)), but there may be some situations in which this isn't feasible.

To explicitly supply credentials to an AWS client:

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.

Provide the class instance to a service client constructor that takes an AWSCredentials interface as input.

For example:

```
BasicAWSCredentials awsCreds = new BasicAWSCredentials(access_key_id,
secret_access_key)
AmazonS3 s3Client = new AmazonS3Client(awsCreds);
```

See Also

- Get an AWS Account and Your AWS Credentials (p. 4)
- Set up your AWS Credentials for Use with the SDK for Java (p. 6)
- Using IAM Roles for EC2 Instances (p. 29)

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.

Each AWS client can be configured to use a specific endpoint by calling the `setEndpoint(String endpointUrl)` 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");
```

Be aware that regions are logically isolated from each other, so for example, you won't be able to access US East resources when communicating with the EU West endpoint. If your code accesses multiple AWS regions, we recommend that you instantiate a specific client for each region, as the following example shows.

```
AmazonEC2 ec2_euro = new AmazonEC2(myCredentials);
ec2_euro.setEndpoint("https://ec2.eu-west-1.amazonaws.com");

AmazonEC2 ec2_us = new AmazonEC2(myCredentials);
ec2_us.setEndpoint("https://ec2.us-east-1.amazonaws.com");
```

Using a specific client for each endpoint also protects against the (unfortunate) scenario in which, in a multithreaded environment, one thread sets the endpoint for a client and then later a different thread changes the endpoint for that client.

The AWS SDK for Java uses the US East (Northern Virginia) Region as the default region if you do not specify a region in your code. However, the AWS Management Console uses US West (Oregon) Region as its default. Therefore, when using the AWS Management Console in conjunction with your development, be sure to specify the same region in both your code and the console.

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

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

Proxy Configuration

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

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

HTTP Transport Configuration

Several HTTP transport options can be configured through the `ClientConfiguration` object. Default values will suffice for the majority of users, but users who want more control can configure the following:

- Socket timeout
- Connection timeout
- Maximum retry attempts for retry-able errors
- Maximum open HTTP connections

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

Setting the JVM TTL for DNS Name Lookups

For Java applications that access Amazon Web Services (AWS), we recommend that you configure your Java virtual machine (JVM) with a time-to-live (TTL) of 60 seconds for DNS name lookups.

The JVM caches DNS name lookups. That is, when the JVM resolves a DNS name to an IP address, it caches the IP address for a period of time. During this time period, the JVM uses the cached IP address rather than querying a DNS server. This time period is known as the time-to-live or TTL. The default TTL varies with the version of the JVM and also depends on whether a security manager is installed.

In some cases, the JVM default TTL is set to *never* re-resolve DNS names to IP addresses. This means that when the IP address for an AWS resource changes, the application will be unable to connect to that resource until someone manually restarts the JVM so that the new IP addresses can be picked up. In these cases, it is vital that the TTL be configured to a shorter time period.

A TTL of 60 seconds ensures that if there is a change in the IP address that corresponds to an AWS resource, the JVM will refresh the cached IP value after a relatively brief period of time. If the TTL value is too large, Java applications may fail to connect to AWS resources because the cached IP has become invalid.

You can configure the TTL in the file java.security, which is located in the directory %JRE%\lib\security. The configured value specifies the number of seconds that the JVM should cache a successful DNS name lookup. Here is an example that shows how to configure the TTL to 60 seconds.

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

You can also configure the TTL programmatically using the following code

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

Note that the JVM is a shared resource; multiple Java applications could be using the same JVM. The TTL value affects all Java applications that use the JVM.

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.

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)

`AmazonServiceException` (and Subclasses)

This is the main type of exception that you'll need to deal with when using the AWS SDK for Java. This exception represents an error response from an AWS service. For example, if you request 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` has several useful fields in it, including:

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

`AmazonServiceException` also includes a field that describes 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).

`AmazonClientException`

This exception 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 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.

`IllegalArgumentException`

When calling operations, if you pass an illegal argument, the AWS SDK for Java will throw an `IllegalArgumentException`. For example, if you call an operation and pass a null value in for one of the required parameters, the SDK will throw an `IllegalArgumentException` describing the illegal argument.

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.

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

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 navigating to:

Project | **Properties** | **Java Build Path**

Service-Specific Errors and Warnings

We recommend that you always leave the "com.amazonaws" logger hierarchy set to "WARN" 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
```

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: Descri
beEngineDefaultParameters, SignatureMethod: HmacSHA256, AWSAccessKeyId: ACCESS
KEYID, Version: 2009-10-16, SignatureVersion: 2, Engine: mysql5.1, Timestamp:
2009-12-17T17:53:04.267Z, Signature: 4ydexGGkC77PovHhbfzAMA1H0nD
nqIQxG9q+Yq3uw5s=, )
2009-12-17 09:53:04,464 [main] DEBUG com.amazonaws.request - Received successful
 response: 200, AWS Request ID: 06c12a39-eb35-11de-ae07-adb69edbb1e4
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-1261072381023, 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: 06e071cb-eb35-11de-81f7-01604e1b25ff
```

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
```

Access Control Policies

AWS access control policies allow you to specify fine-grained access controls on your AWS resources. You can allow or deny access to your AWS resources based on:

- what *resource* is being accessed.
- who is accessing the resource (i.e., the principal).
- what action is being taken on the resource.
- a variety of other conditions including date restrictions, IP address restrictions, etc.

Access control policies are a collection of statements. Each statement takes the form: "A has permission to do B to C where D applies".

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
Your AWS entity that the principal wants to access, such as an Amazon SQS queue, or an object stored in Amazon S3.

D is the set of conditions
The optional constraints that specify when to allow or deny access for the principal to access your resource. Many expressive conditions are available, some specific to each service. For example, you can use date conditions to allow access to your resources only after or before a specific time.

Amazon S3 Example

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

```
Statement allowPublicReadStatement = new Statement(Effect.Allow)
    .withPrincipals(Principal.AllUsers)
    .withActions(S3Actions.GetObject)
```

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

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

AmazonS3 s3 = new AmazonS3Client(myAwsCredentials);
s3.setBucketPolicy(myBucketName, policy.toJson());
```

Amazon SQS Example

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

```
/*
 * This policy allows an SNS topic to send messages to an SQS queue.
 * You can find your SNS topic's ARN through the SNS getTopicAttributes opera
tion.
 */
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 = new AmazonSQSClient(myAwsCredentials);
sqs.setQueueAttributes(new SetQueueAttributesRequest(myQueueUrl, queueAttrib
utes));
```

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, SQS) and endpoint (e.g., email address, URL, SQS ARN) of the request to subscribe to a topic.

```
/*
 * This SNS condition allows you to restrict subscriptions to an Amazon SNS
topic
 * based on the requested endpoint (email address, SQS queue ARN, etc.) used
when
 * someone tries to subscribe to your SNS 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 = new AmazonSNSClient(myAwsCredentials);
sns.setTopicAttributes(
    new SetTopicAttributesRequest(myTopicArn, "Policy", policy.toJson()));
```

Using Amazon Web Services with the AWS SDK for Java

This section provides information about how to program various Amazon Web Services using the SDK for Java.

Topics

Programming DynamoDB with the AWS SDK for Java

This section provides information specific to programming DynamoDB with the SDK for Java.

Topics

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.Dy
namoDBSessionManager"
               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 AWS Elastic Beanstalk

If you're using the session manager in AWS 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.

You can find in more information on customizing the AWS Elastic Beanstalk environments in the developer guide for that service.

If you deploy to AWS 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 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. (For information on working with DynamoDB tables and provisioned throughput, see the Amazon DynamoDB Developer Guide.) 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.

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**.
- **Region** – Optional string attribute. The AWS region in which to use DynamoDB. For a list of available AWS regions, see Regions and Endpoints in the *AWS 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 *AWS 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 AWS 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.

Programming Amazon EC2 with the AWS SDK for Java

This section provides information specific to programming Amazon EC2 with the SDK for Java.

Topics

Starting an Amazon EC2 Instance

This topic demonstrates how to use the AWS SDK for Java to start an Amazon Elastic Compute Cloud (Amazon EC2) instance.

Topics

Create an Amazon EC2 Client

You will need an Amazon EC2 client in order to create security groups and key pairs, and run Amazon EC2 instances. Before initializing your client, however, you need to create an `AwsCredentials.properties` file to store your AWS access key and your secret key.

The file looks like this:

```
secretKey=YOUR_SECRET_KEY
accessKey=YOUR_ACCESS_KEY
```

Specify your AWS credentials as values for the `secretKey` and `accessKey` entries. To learn more about your AWS credentials, including where to find them, go to AWS Security Credentials.

After you create this file, you are ready to create and initialize your Amazon EC2 client.

To create and initialize an Amazon EC2 client

1. Create and initialize an AWSCredentials instance. Specify the `AwsCredentials.properties` file you created, as follows:

   ```
   AWSCredentials credentials =
     new PropertiesCredentials(
            AwsConsoleApp.class.getResourceAsStream("AwsCredentials.proper
   ties"));
   ```

2. Use the `AWSCredentials` object to create a new AmazonEC2Client instance, as follows:

   ```
   amazonEC2Client =
     new AmazonEC2Client(credentials);
   ```

3. By default, the service endpoint is `ec2.us-east-1.amazonaws.com`. To specify a different endpoint, use the setEndpoint method. For example:

   ```
   amazonEC2Client.setEndpoint("ec2.us-west-2.amazonaws.com");
   ```

   The AWS SDK for Java uses US East (N. Virginia) as the default region if you do not specify a region in your code. However, the AWS Management Console uses US West (Oregon) as its default. Therefore, when using the AWS Management Console in conjunction with your development, be sure to specify the same region in both your code and the console.

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

Before running an Amazon EC2 instance, you will need to create an Amazon EC2 security group, authorize security group ingress, and create a key pair to allow you to log into your instance.

For information about creating a security group, see Create an Amazon EC2 Security Group (p. 24).

For information about authorizing security group ingress, see Authorize Amazon EC2 Security Group Ingress (p. 25).

For information about creating a key pair, see Create a Key Pair (p. 26).

For information about running your Amazon EC2 instance, see Run an Amazon EC2 Instance (p. 27).

Create an Amazon EC2 Security Group

An Amazon EC2 *security group* controls traffic through your Amazon EC2 instances, much like a firewall. If you do not create a security group, Amazon EC2 provides a default security group that allows no inbound traffic. For more information about security groups, go to Security Group Concepts.

If you want to allow inbound traffic, create a security group and assign a *rule* to it that allows the ingress that you want. Then associate the new security group with an Amazon EC2 instance. For more information, see Authorize Security Group Ingress (p. 25).

To create an Amazon EC2 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 createSecurityGroupRequest =
 new CreateSecurityGroupRequest();

createSecurityGroupRequest.withGroupName("JavaSecurityGroup")
 .withDescription("My Java 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);
```

   You can create up to 500 security groups per AWS account.

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

Before starting an Amazon EC2 instance, you next need to authorize security group ingress and create a key pair to allow you to log into your instance.

For information about authorizing security group ingress, see Authorize Amazon EC2 Security Group Ingress (p. 25).

For information about creating a key pair, see Create a Key Pair (p. 26).

For information about running your Amazon EC2 instance, see Run an Amazon EC2 Instance (p. 27).

Authorize Security Group Ingress

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 withIpRanges 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(authorizeSecurityGroupIngress
Request);
```

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. You can add up to 100 rules per security group. For more information about security groups, go to Security Group Concepts.

Before starting an Amazon EC2 instance, you need to create a key pair to allow you to log into your instance. For information about creating a key pair, see Create a Key Pair (p. 26).

Create a Key Pair

Public AMI instances have no default password. To log into your Amazon EC2 instance, you must generate an Amazon EC2 key pair. The key pair consists of a public key and a private key, and is not the same as your AWS access credentials. For more information about Amazon EC2 key pairs, go to Getting an SSH Key Pair.

To create a key pair and obtain 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);
```

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

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

   Key pair names must be unique. If you attempt to create a key pair with the same key name as an existing key pair, `createKeyPair` returns an exception.
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();
```

   Calling `createKeyPair` is the only way to obtain the private key programmatically.

Before logging onto an Amazon EC2 instance, you must create the instance and ensure that it is running. For information on how to run an Amazon EC2 instance, see Run an Amazon EC2 Instance (p. 27).

For information on how to use your key pair to connect to your Amazon EC2 instance, see Connect to Your Amazon EC2 Instance (p. 28).

Run an Amazon EC2 Instance

Before running an Amazon EC2 instance, ensure that you have created a security group and a key pair for your instance. For information about creating a key pair, see Create a Key Pair (p. 26). For information about creating a security group, see Create an Amazon EC2 Security Group (p. 24).

To run an Amazon EC2 instance

1. Create and initialize a RunInstancesRequest instance. Specify the Amazon Machine Image (AMI) (withImageId), the instance type (withInstanceType), the minimum (withMinCount) and maximum (withMaxCount) number of instances to run, key pair name (withKeyName), and the name of one or more security groups (withSecurityGroups), as follows:

```
RunInstancesRequest runInstancesRequest =
```

```
 new RunInstancesRequest();

runInstancesRequest.withImageId("ami-4b814f22")
                   .withInstanceType("m1.small")
                   .withMinCount(1)
                   .withMaxCount(1)
                   .withKeyName("YourKeyName")
                   .withSecurityGroups("YourSecurityGroupName");
```

You must specify a public or privately-provided AMI. A large selection of Amazon-provided public AMIs is available for you to use. For a list of public AMIs provided by Amazon, go to Amazon Machine Images. Ensure that the specified image ID exists in the region in which your client was created.

The instance type must match the AMI you want to run. For 64-bit architecture, you cannot specify an instance type of `m1.small`. For more information on instance types, go to Instance Families and Types.

You must specify a minimum number and a maximum number of instances to launch. If the specified number of instances is greater than the number of instances you are authorized to launch, no instances are launched. The specified number of maximum instances must be no greater than the maximum number allowed for your account; by default, the maximum number of instances is 20. If there are fewer instances available than the maximum number specified, the largest possible number of images are launched.

Ensure that the specified key name and security group exists for the region in which your client was created.

2. Pass the request object into the runInstances method. The method returns a RunInstancesResult object, as follows:

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

After you have created your Amazon EC2 instance, you can log onto the AWS Management Console to check the status of the instance.

Once your Amazon EC2 instance is running, you can remotely connect to it using your key pair. For information about connecting to your instance, see Connect to Your Amazon EC2 Instance (p. 28).

Connect to Your Amazon EC2 Instance

Before connecting to your Amazon EC2 instance, you must ensure that the instance's SSH/RDP port is open to traffic. You must also install an SSH/RDP client on the computer you are accessing your instance from. You will need your Amazon EC2 instance ID and the private key from the key pair you created. For information on how to obtain the private key, see Create a Key Pair (p. 26).

If you did not authorize ingress for the security group that your instance belongs to, you will not be able to connect to your instance. By default, Amazon EC2 instances do not permit inbound traffic. For more information on authorizing security group ingress, see Authorize Security Group Ingress (p. 25).

For information on how to connect to your Amazon EC2 instance, go to Connecting to Instances in the *Amazon Elastic Compute Cloud User Guide*.

Related Resources

The following table lists related resources that you'll find useful when using Amazon EC2 with the AWS SDK for Java.

Resource	Description
Java Developer Center	Provides sample code, documentation, tools, and additional resources to help you build applications on Amazon Web Services.
AWS SDK for Java Documentation	Provides documentation for the AWS SDK for Java.
Amazon Elastic Compute Cloud (Amazon EC2) Documentation	Provides documentation for the Amazon EC2 service.

Using IAM Roles for EC2 Instances with the AWS SDK for Java

When writing software that accesses Amazon Web Services (AWS), you must always consider how you will manage authentication credentials: *all requests to AWS* must be cryptographically signed using credentials issued by AWS.

For software that runs on Amazon Elastic Compute Cloud (Amazon EC2) instances which are, by definition, internet-accessible servers, you must manage AWS credentials in a way that *keeps them secure* but also *makes them accessible* to your application so it can make AWS requests.

Using *IAM roles for EC2 instances* is a secure, effective way to provide AWS credentials to applications that run on EC2 instances. This section will describe what IAM roles for EC2 instances are, how they're used, and then show you how it works with a simple Java program.

> **Note**
> For complete information about using IAM Roles for EC2 instances, see Using Identity and Access Management in the *AWS Identity and Access Management User Guide*.

Topics

How Applications Obtain AWS Credentials Using IAM Roles for EC2 Instances

Using IAM roles, you can develop software and deploy it to an EC2 instance without needing to directly manage the credentials that your software uses to access AWS. Instead, you create an IAM role and configure it with the permissions that your software requires. Your application can then use this role to access AWS through an *instance profile*, a logical container for the IAM role you defined that is associated with your EC2 instance.

> **Note**
> Instance profiles that can be assigned to EC2 instances are automatically created whenever a new IAM role is created on the AWS Management Console, so if you use the IAM console to create your roles, you won't need to create instance profiles directly.

Using IAM Roles with the SDK for Java

If your application creates an AWS client using its default constructor (by providing the constructor with no arguments), then the SDK for Java will search for credentials, in order, in the following places:

1. In the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`.
2. In the Java system properties `aws.accessKeyId` and `aws.secretKey`.
3. From the Instance Metadata Service (IMDS). The IMDS is what provides credentials using the IAM role contained in the EC2 instance profile.

By default, if the AWS client constructor can't find credentials in the *environment variables* or in the *Java system properties*, it will attempt to obtain temporary AWS credentials from the IMDS. These credentials have the same permissions as the IAM role associated with the EC2 instance in its instance profile.

> **Note**
> The credentials provided by the IMDS are temporary and eventually expire, but the SDK for Java client *periodically refreshes them* so that your application does not lose access to AWS. This credential refesh is automatic and is transparent to your application; no actions need to be taken by your application to refresh the credentials obtained through the IMDS.

You can tell the SDK for Java to use AWS credentials obtained from the IMDS *first*, before it attempts to look for credentials in the environment or java system properties, by passing an InstanceProfileCredentialsProvider object to the AWS client's constructor. For example, to create an S3 client using IMDS-supplied credentials:

```
AmazonS3 s3Client = new AmazonS3Client(new InstanceProfileCredentialsProvider());
```

If the client constructor can't find credentials in the IMDS or through any of the other methods in the credentials provider chain, an AmazonClientException will be thrown.

> **Note**
> AWS CloudFormation does not support calling its API with an IAM role. You must call the AWS CloudFormation API as a regular IAM user.

Walkthrough: Using IAM Roles to Retrieve an Amazon S3 Object from an EC2 Instance

In this walkthrough, we'll show an example of a program that uses IAM roles for EC2 instances to manage access.

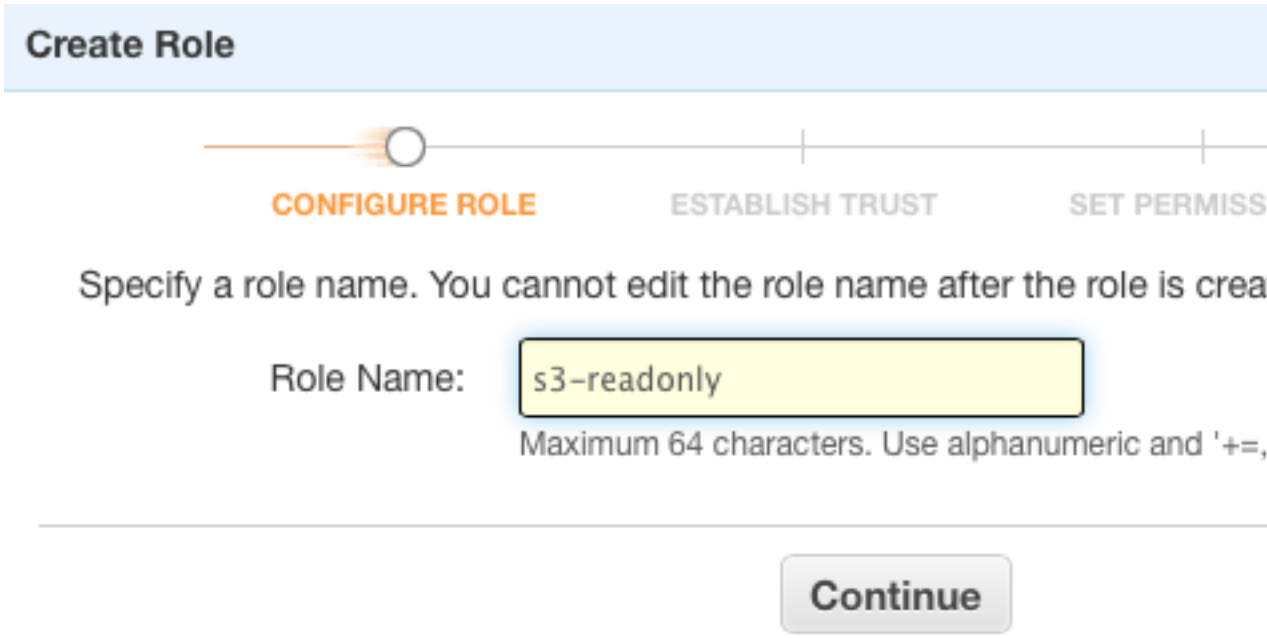
Topics

Create the IAM Role

We'll begin by creating an IAM role with appropriate permissions for our application. You can create an IAM role in a number of different ways, including with the AWS CLI. For more information about each of the ways to create roles, see Creating a IAM Role in *IAM User Guide*. We'll use the AWS Management Console in this walkthrough, since you don't need to install anything to use it.

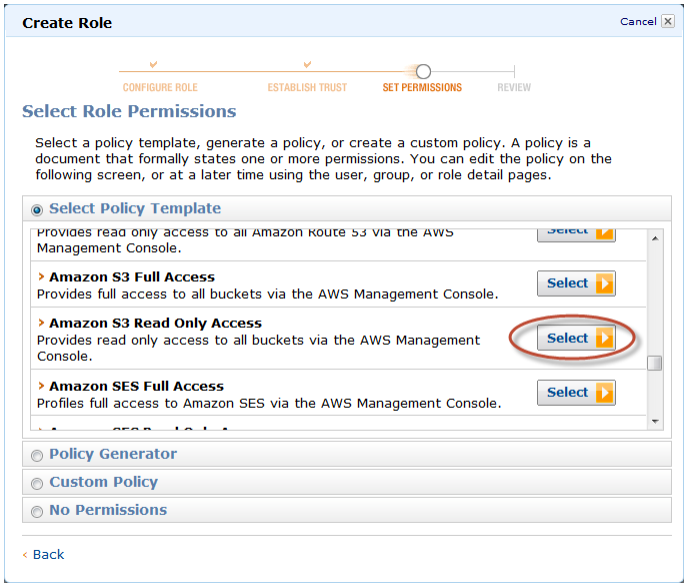
To create the IAM role to give S3 read-only access to your EC2 instance

1. Log in to the AWS Management Console and go to IAM.
2. Select **Roles** in the left sidebar and click **Create New Role**.
3. Provide a memorable name for your role (you'll need to enter its name when you create the EC2 instance, later), and click **Continue**.



4.

The IAM console provides ready-made policy templates for specific AWS services. When you create the IAM role, specify the **Amazon S3 Read Only Access** policy template. The following screen shot from the IAM role creation wizard shows this policy template.



Note
Policies can also be represented in JSON format. Here is a policy that provides read ("Get" and "List") access to Amazon S3.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:Get*",
        "s3:List*"
      ],
      "Resource": "*"
    }
  ]
}
```

This is the policy that provides a trust relationship with EC2, allowing your instance to assume the IAM role:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Remember the name of the IAM role that you create. You'll need to provide it when you create your EC2 instance in the next step.

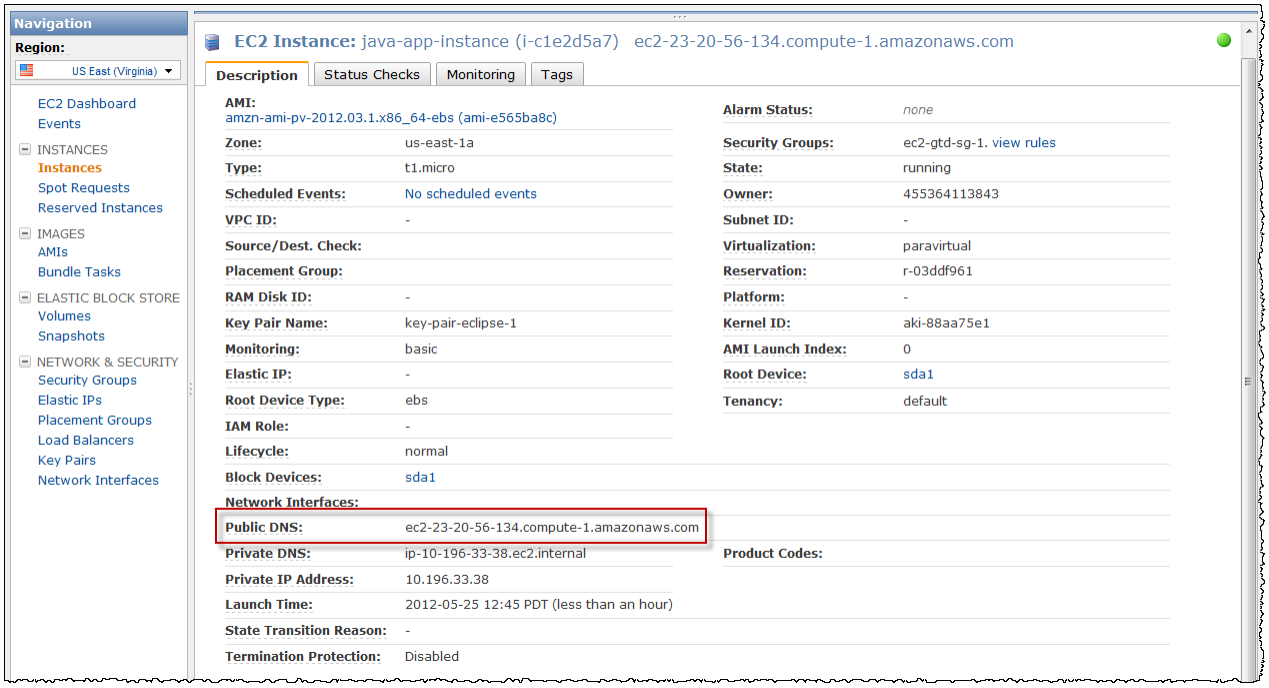
Launch an EC2 Instance with an Instance Profile

To create an EC2 instance, follow the procedure Running an Instance in the Amazon Elastic Compute Cloud User Guide. We recommend that you specify a recent **Amazon Linux AMI** for your EC2 instance. When you create the EC2 instance, specify the IAM role that you created previously in the IAM console. This role can be used by your application through the Instance Meta Data Service (IMDS).

When you create your EC2 instance, you also need to specify a *key pair* and a *security group*. Make sure to use a key pair for which you have the private key (PEM file) stored on your local computer. Specify a security group that enables you to connect to your EC2 instance using SSH (port 22).

Note
Information about key pairs and security groups is provided in Running an Instance in the Amazon Elastic Compute Cloud User Guide.

After you create the EC2 instance, go to the **EC2 Instances** area of the AWS Management Console and view your instance. Once the instance is **Running**, record the public DNS name for the instance. You will use this DNS name to connect to the instance with SSH.



Create your Application

We'll now build the application that will run on the Amazon EC2 instance. First, create a directory that you can use to hold your tutorial files. Call it something like `GetS3ObjectApp`.

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

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

Here is the program that we'll be transferring to your EC2 instance. Open a new file, call it `GetS3Objbect.java`, and type (or paste) 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 = new AmazonS3Client();

    try
```

```
    {
      System.out.println("Downloading an object");
      S3Object s3object = s3Client.getObject(
          new GetObjectRequest(bucketName, key));
      displayTextInputStream(s3object.getObjectContent());
    }
    catch(AmazonServiceException ase)
    {
      System.out.println( "AmazonServiceException" );
    }
    catch(AmazonClientException ace)
    {
      System.out.println( "AmazonClientException" );
    }
  }

  private static void displayTextInputStream(InputStream input) throws IOExcep
tion
  {
    // 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();
  }
}
```

You'll also need an Ant build script to build and run your application. Open a new file in the same directory as `GetS3Object.java` and call it `build.xml`. Add the following lines to the file:

```
<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. Since no credentials are stored in the program, unless you have your AWS credentials specified in your environment, it's likely that you'll get an 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 EC2 instance using secure copy (scp). You will also need to copy the SDK for Java libraries. The sequence of commands will look something like the following:

```
scp -p -i key-pair-eclipse-1.pem GetS3Object.class \
  ec2-user@ec2-ex-am-pl-e01.compute-1.amazonaws.com:GetS3Object.class

scp -p -i key-pair-eclipse-1.pem build.xml \
  ec2-user@ec2-ex-am-pl-e01.compute-1.amazonaws.com:build.xml

scp -r -p -i key-pair-eclipse-1.pem lib \
  ec2-user@ec2-ex-am-pl-e01.compute-1.amazonaws.com:lib

scp -r -p -i key-pair-eclipse-1.pem third-party \
  ec2-user@ec2-ex-am-pl-e01.compute-1.amazonaws.com:third-party
```

Note
If you launched an AMI *other* than the Amazon Linux AMI recommended earlier, you may need to use "root" instead of "ec2-user" when connecting to the instance using SCP or SSH.

In the preceding commands, `GetS3Object.class` is your compiled program, `build.xml` is the ant file used to build and run your program, and the `lib` and `third-party` directories are the corresponding library folders from the 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 from the SDK for Java.

The `-p` switch indicates that **scp** should preserve the permissions of the source files when it copies them to the destination. If you are copying the files from Windows, you may need to fix the permissions on your instance by using the following command:

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

Run the Program

To run the program, use SSH to connect to your EC2 instance.

```
ssh -i key-pair-eclipse-1.pem ec2-user@ec2-23-20-56-134.compute-1.amazonaws.com
```

If ant is not installed on the AMI that you selected, you can install it using the yum installer.

```
sudo yum install ant
```

Run the program using ant.

```
ant getS3Object
```

The program should output the contents of your Amazon S3 object to your command window.

Tutorial: Amazon EC2 Spot Instances

Overview

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

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

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

It's important to note two points:

- 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 provides a quick overview of how to use the Java programming language 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 need to be signed up for Amazon Web Services (AWS). If you have not yet signed up for AWS, go to the Amazon Web Services website, and click **Create an AWS Account** in the upper right corner of the page. In addition, you also need to install the AWS Java SDK.

If you are using the Eclipse development environment, we recommend that you install the AWS Toolkit for Eclipse. Note that the AWS Toolkit for Eclipse includes the latest version of the AWS SDK for Java.

Step 1: Setting Up Your Credentials

To begin using this code sample, you need to populate your credentials in the `AwsCredentials.properties` file. Specifically, you need to populate your secret key and access key.

Copy and paste your access key ID and secret access key into the `AwsCredentials.properties` file.

To get your access key ID and secret access key

Access keys consist of an access key ID and secret access key, which are used to sign programmatic requests that you make to AWS. If you don't have access keys, you can create them by using the AWS Management Console. We recommend that you use IAM access keys instead of AWS root account access keys. IAM lets you securely control access to AWS services and resources in your AWS account.

> **Note**
> To create access keys, you must have permissions to perform the required IAM actions. For more information, see Granting IAM User Permission to Manage Password Policy and Credentials in *Using IAM*.

1. Go to the IAM console.
2. From the navigation menu, click **Users**.
3. Select your IAM user name.
4. Click **User Actions**, and then click **Manage Access Keys**.
5. Click **Create Access Key**.

   Your keys will look something like this:

   - Access key ID example: AKIAIOSFODNN7EXAMPLE
   - Secret access key example: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY

6. Click **Download Credentials**, and store the keys in a secure location.

   Your secret key will no longer be available through the AWS Management Console; you will have the only copy. Keep it confidential in order to protect your account, and never email it. Do not share it outside your organization, even if an inquiry appears to come from AWS or Amazon.com. No one who legitimately represents Amazon will ever ask you for your secret key.

Related topics

- What Is IAM? in *Using IAM*

- AWS Security Credentials in *AWS General Reference*

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.

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

    // Create the AmazonEC2Client object so we can call various APIs.
    AmazonEC2 ec2 = new AmazonEC2Client(credentials);
 15
    // Create a new security group.
    try {
        CreateSecurityGroupRequest securityGroupRequest = new CreateSecurity
GroupRequest("GettingStartedGroup", "Getting Started Security Group");
        ec2.createSecurityGroup(securityGroupRequest);
 20 } catch (AmazonServiceException ase) {
        // Likely this means that the group is already created, so ignore.
        System.out.println(ase.getMessage());
    }

 25 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 {
 30     InetAddress addr = InetAddress.getLocalHost();
```

```
        // Get IP Address
        ipAddr = addr.getHostAddress()+"/10";
    } catch (UnknownHostException e) {
35  }

    // Create a range that you would like to populate.
    ArrayList<String> ipRanges = new ArrayList<String>();
    ipRanges.add(ipAddr);
40
    // 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();
45  ipPermission.setIpProtocol("tcp");
    ipPermission.setFromPort(new Integer(22));
    ipPermission.setToPort(new Integer(22));
    ipPermission.setIpRanges(ipRanges);
    ipPermissions.add(ipPermission);
50
    try {
        // Authorize the ports to the used.
        AuthorizeSecurityGroupIngressRequest ingressRequest =
            new AuthorizeSecurityGroupIngressRequest("GettingStartedGroup",ip
Permissions);
55      ec2.authorizeSecurityGroupIngress(ingressRequest);
    } catch (AmazonServiceException ase) {
        // Ignore because this likely means the zone has
        // already been authorized.
        System.out.println(ase.getMessage());
60  }
```

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. Go to the toolkit documentation 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.

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.

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 AmazonEC2Client object so we can call various APIs.
AmazonEC2 ec2 = new AmazonEC2Client(credentials);

// 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 the Java Developers: Advanced Spot Features Tutorials or the RequestSpotInstances API in the SDK for Java.

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(re
questRequest);
   List<SpotInstanceRequest> requestResponses = requestResult.getSpotInstance
Requests();

// 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.getSpotIn
stanceRequestId());
      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 DescribeSpot
InstanceRequestsRequest();
       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.describeS
potInstanceRequests(describeRequest);
           List<SpotInstanceRequest> describeResponses = describeResult.get
SpotInstanceRequests();
```

```
20          // 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
25              // almost immediately to closed or cancelled so we compare
                // against open instead of active.
            if (describeResponse.getState().equals("open")) {
                anyOpen = true;
                break;
30          }
        }
    } catch (AmazonServiceException e) {
          // If we have an exception, ensure we don't break out of
          // the loop. This prevents the scenario where there was
35        // blip on the wire.
          anyOpen = true;
        }

        try {
40          // Sleep for 60 seconds.
            Thread.sleep(60*1000);
        } catch (Exception e) {
            // Do nothing because it woke up early.
        }
45  } 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—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.

```
  1 try {
        // Cancel requests.
       CancelSpotInstanceRequestsRequest cancelRequest = new CancelSpotInstan
ceRequestsRequest(spotInstanceRequestIds);
        ec2.cancelSpotInstanceRequests(cancelRequest);
  5 } 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());
 10     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 DescribeSpot
InstanceRequestsRequest();
    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.describeS
potInstanceRequests(describeRequest);
        List<SpotInstanceRequest> describeResponses = describeResult.get
SpotInstanceRequests();

        // 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.

```
  1 try {
        // Terminate instances.
        TerminateInstancesRequest terminateRequest = new TerminateInstances
Request(instanceIds);
        ec2.terminateInstances(terminateRequest);
  5 } 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());
 10     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 is available for download at GitHub.

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

Next Steps

We recommend that you take the Java Developers: Tutorial: Advanced Amazon EC2 Spot Request Management (p. 45).

Tutorial: Advanced Amazon EC2 Spot Request Management

Overview

Spot Instances allow you to bid on unused Amazon Elastic Compute Cloud (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. Customers whose bids meet or exceed the Spot Price gain access to the available Spot Instances. Like On-Demand Instances and Reserved Instances, Spot Instances provide you an additional 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 can provide access to large amounts of additional compute capacity when your need for the capacity is not urgent.

This tutorial provides a quick overview of some advanced Spot Request features, such as detailed options to create Spot requests, alternative methods for launching Spot Instances, and methods to manage your instances. This tutorial is not meant to be a complete list of all advanced topics associated with Spot Instances. Instead, it gives you a quick reference of code samples for some of the commonly used methods for managing Spot Requests and Spot Instances.

Prerequisites

To use this tutorial you need to be signed up for Amazon Web Services (AWS). If you have not yet signed up for AWS, go to the Amazon Web Services website, and click **Create an AWS Account** in the upper right corner of the page. In addition, you also need to install the AWS SDK for Java.

If you are using the Eclipse development environment, we recommend that you install the AWS Toolkit for Eclipse. The Toolkit for Eclipse includes the latest version of the AWS SDK for Java.

Step 1: Setting Up Your Credentials

To begin using this code sample, you need to populate your credentials in the `AwsCredentials.properties` file. Specifically, you need to populate your secretKey and accessKey.

Copy and paste your access key ID and secret access key into the `AwsCredentials.properties` file.

To get your access key ID and secret access key

Access keys consist of an access key ID and secret access key, which are used to sign programmatic requests that you make to AWS. If you don't have access keys, you can create them by using the AWS Management Console. We recommend that you use IAM access keys instead of AWS root account access keys. IAM lets you securely control access to AWS services and resources in your AWS account.

> **Note**
> To create access keys, you must have permissions to perform the required IAM actions. For more information, see Granting IAM User Permission to Manage Password Policy and Credentials in *Using IAM*.

1. Go to the IAM console.
2. From the navigation menu, click **Users**.
3. Select your IAM user name.
4. Click **User Actions**, and then click **Manage Access Keys**.
5. Click **Create Access Key**.

   Your keys will look something like this:

   - Access key ID example: AKIAIOSFODNN7EXAMPLE
   - Secret access key example: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY

6. Click **Download Credentials**, and store the keys in a secure location.

   Your secret key will no longer be available through the AWS Management Console; you will have the only copy. Keep it confidential in order to protect your account, and never email it. Do not share it outside your organization, even if an inquiry appears to come from AWS or Amazon.com. No one who legitimately represents Amazon will ever ask you for your secret key.

Related topics

- What Is IAM? in *Using IAM*
- AWS Security Credentials in *AWS General Reference*

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.

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

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

    // Create the AmazonEC2Client object so we can call various APIs.
    AmazonEC2 ec2 = new AmazonEC2Client(credentials);
 15
    // Create a new security group.
    try {
        CreateSecurityGroupRequest securityGroupRequest =
            new CreateSecurityGroupRequest("GettingStartedGroup", "Getting
Started Security Group");
 20     ec2.createSecurityGroup(securityGroupRequest);
    } catch (AmazonServiceException ase) {
        // Likely this means that the group is already created, so ignore.
        System.out.println(ase.getMessage());
    }
 25
    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.
30 try {
       InetAddress addr = InetAddress.getLocalHost();

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

   // Create a range that you would like to populate.
   ArrayList<String> ipRanges = new ArrayList<String>();
40 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> ();
45 IpPermission ipPermission = new IpPermission();
   ipPermission.setIpProtocol("tcp");
   ipPermission.setFromPort(new Integer(22));
   ipPermission.setToPort(new Integer(22));
   ipPermission.setIpRanges(ipRanges);
50 ipPermissions.add(ipPermission);

   try {
       // Authorize the ports to the used.
       AuthorizeSecurityGroupIngressRequest ingressRequest =
55         new AuthorizeSecurityGroupIngressRequest("GettingStartedGroup",ip
Permissions);
       ec2.authorizeSecurityGroupIngress(ingressRequest);
   } catch (AmazonServiceException ase) {
       // Ignore because this likely means the zone has already
       // been authorized.
60     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.

You can also create the security group using the AWS Toolkit for Eclipse. Go to the toolkit documentation for more information.

Detailed Spot Instance Request Creation Options

As we explained in Tutorial: Amazon EC2 Spot Instances (p. 36), 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.)

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

    // Create the AmazonEC2Client object so we can call various APIs.
    AmazonEC2 ec2 = new AmazonEC2Client(credentials);
 15
    // Initializes a Spot Instance Request
    RequestSpotInstancesRequest requestRequest = new RequestSpotInstances
Request();

    // Request 1 x t1.micro instance with a bid price of $0.03.
 20 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
 25 // 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");
 30
    // Add the security group to the request.
    ArrayList<String> securityGroups = new ArrayList<String>();
    securityGroups.add("GettingStartedGroup");
    launchSpecification.setSecurityGroups(securityGroups);
 35
    // Add the launch specification.
    requestRequest.setLaunchSpecification(launchSpecification);

    // Call the RequestSpotInstance API.
 40 RequestSpotInstancesResult requestResult = ec2.requestSpotInstances(re
questRequest);
```

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.

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

    // Create the AmazonEC2Client object so we can call various APIs.
 15 AmazonEC2 ec2 = new AmazonEC2Client(credentials);

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

 20 // 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.
 25 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
 30 // Amazon Linux AMI id or another of your choosing.
    LaunchSpecification launchSpecification = new LaunchSpecification();
    launchSpecification.setImageId("ami-8c1fece5");
    launchSpecification.setInstanceType("t1.micro");

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

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

    // Call the RequestSpotInstance API.
    RequestSpotInstancesResult requestResult = ec2.requestSpotInstances(re
questRequest);
 45
```

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.

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

    // Create the AmazonEC2Client object so we can call various APIs.
    AmazonEC2 ec2 = new AmazonEC2Client(credentials);

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

    // 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(re
questRequest);
```

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.

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

    // Create the AmazonEC2Client object so we can call various APIs.
    AmazonEC2 ec2 = new AmazonEC2Client(credentials);
 15
    // Initializes a Spot Instance Request
    RequestSpotInstancesRequest requestRequest = new RequestSpotInstances
Request();

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

    // Set the launch group.
    requestRequest.setLaunchGroup("ADVANCED-DEMO-LAUNCH-GROUP");
 25
    // 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.
 30 LaunchSpecification launchSpecification = new LaunchSpecification();
    launchSpecification.setImageId("ami-8c1fece5");
    launchSpecification.setInstanceType("t1.micro");

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

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

    // Call the RequestSpotInstance API.
    RequestSpotInstancesResult requestResult = ec2.requestSpotInstances(re
```

```
questRequest);
```

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.

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

    // Create the AmazonEC2Client object so we can call various APIs.
    AmazonEC2 ec2 = new AmazonEC2Client(credentials);
 15
    // Initializes a Spot Instance Request
    RequestSpotInstancesRequest requestRequest = new RequestSpotInstances
Request();

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

    // Set the availability zone group.
    requestRequest.setAvailabilityZoneGroup("ADVANCED-DEMO-AZ-GROUP");
 25
    // 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.
 30 LaunchSpecification launchSpecification = new LaunchSpecification();
    launchSpecification.setImageId("ami-8c1fece5");
    launchSpecification.setInstanceType("t1.micro");

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

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

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

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.

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

    // Create the AmazonEC2Client object so we can call various APIs.
    AmazonEC2 ec2 = new AmazonEC2Client(credentials);

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

    // 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(re
questRequest);
```

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.

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

    // Create the AmazonEC2Client object so we can call various APIs.
    AmazonEC2 ec2 = new AmazonEC2Client(credentials);
 15
    // Initializes a Spot Instance Request
    RequestSpotInstancesRequest requestRequest = new RequestSpotInstances
Request();

    // Request 1 x t1.micro instance with a bid price of $0.03.
 20 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.
 25 // 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");
 30
    // Add the security group to the request.
    ArrayList<String> securityGroups = new ArrayList<String>();
    securityGroups.add("GettingStartedGroup");
    launchSpecification.setSecurityGroups(securityGroups);
 35
    // 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");
 40 launchSpecification.setPlacement(placement);

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

 45 // Call the RequestSpotInstance API.
    RequestSpotInstancesResult requestResult = ec2.requestSpotInstances(re
questRequest);
```

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.

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

    // Create the AmazonEC2Client object so we can call various APIs.
    AmazonEC2 ec2 = new AmazonEC2Client(credentials);
 15
    // Initializes a Spot Instance Request
    RequestSpotInstancesRequest requestRequest = new RequestSpotInstances
Request();

    // Request 1 x t1.micro instance with a bid price of $0.03.
 20 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.
 25 // 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");
 30
    // Add the security group to the request.
    ArrayList<String> securityGroups = new ArrayList<String>();
    securityGroups.add("GettingStartedGroup");
    launchSpecification.setSecurityGroups(securityGroups);
 35
```

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

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

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

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

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

 55 // Call the RequestSpotInstance API.
    RequestSpotInstancesResult requestResult = ec2.requestSpotInstances(re
questRequest);
```

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 add tags to your resources, you need to tag them *after* they have been requested. Specifically, you must add a tag after a Spot request has been submitted or after the `RunInstances` call has been performed. The following code example illustrates adding tags.

```
  1
    /*
     * Copyright 2010-2011 Amazon.com, Inc. or its affiliates. All Rights Re
served.
     *
  5  * Licensed under the Apache License, Version 2.0 (the "License").
     * You may not use this file except in compliance with the License.
     * A copy of the License is located at
     *
     *  http://aws.amazon.com/apache2.0
 10  *
```

```
    * or in the "license" file accompanying this file. This file is distributed
    * on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
     * express or implied. See the License for the specific language governing
     * permissions and limitations under the License.
 15  */
   package com.amazonaws.codesamples.advanced;

   import java.io.IOException;
   import java.util.ArrayList;
 20 import java.util.List;

   import com.amazonaws.AmazonServiceException;
   import com.amazonaws.auth.AWSCredentials;
   import com.amazonaws.auth.PropertiesCredentials;
 25 import com.amazonaws.codesamples.getting_started.GettingStartedApp;
   import com.amazonaws.services.ec2.AmazonEC2;
   import com.amazonaws.services.ec2.AmazonEC2Client;
   import com.amazonaws.services.ec2.model.CancelSpotInstanceRequestsRequest;
   import com.amazonaws.services.ec2.model.CreateTagsRequest;
 30 import com.amazonaws.services.ec2.model.DescribeSpotInstanceRequestsRequest;
   import com.amazonaws.services.ec2.model.DescribeSpotInstanceRequestsResult;
    import com.amazonaws.services.ec2.model.LaunchSpecification;
    import com.amazonaws.services.ec2.model.RequestSpotInstancesRequest;
    import com.amazonaws.services.ec2.model.RequestSpotInstancesResult;
 35 import com.amazonaws.services.ec2.model.SpotInstanceRequest;
    import com.amazonaws.services.ec2.model.Tag;
    import com.amazonaws.services.ec2.model.TerminateInstancesRequest;

    /**
 40  * Welcome to your new AWS Java SDK based project!
     *
    * This class is meant as a starting point for your console-based application
 that
     * makes one or more calls to the AWS services supported by the Java SDK,
such as EC2,
     * SimpleDB, and S3.
 45  *
     * In order to use the services in this sample, you need:
     *
     *  - A valid Amazon Web Services account. You can register for AWS at:
     *       https://aws-portal.amazon.com/gp/aws/developer/registration/in
dex.html
 50  *
     *  - Your account's Access Key ID and Secret Access Key:
     *       http://aws.amazon.com/security-credentials
     *
     *  - A subscription to Amazon EC2. You can sign up for EC2 at:
 55  *       http://aws.amazon.com/ec2/
     *
     */

   public class InlineTaggingCodeSampleApp {
 60
      /**
       * @param args
       */
      public static void main(String[] args) {
 65     //============================================================//
```

```
        //================ Submitting a Request ========================//
        //==============================================================//

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

 80     // Create the AmazonEC2Client object so we can
        // call various APIs.
        AmazonEC2 ec2 = new AmazonEC2Client(credentials);

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

        // Request 1 x t1.micro instance with a bid price of $0.03.
        requestRequest.setSpotPrice("0.03");
        requestRequest.setInstanceCount(Integer.valueOf(1));
 90
        // 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.
 95     LaunchSpecification launchSpecification = new LaunchSpecification();
        launchSpecification.setImageId("ami-8c1fece5");
        launchSpecification.setInstanceType("t1.micro");

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

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

        //==============================================================//
        //======== Getting the Request ID from the Request ============//
        //==============================================================//
110
        // Call the RequestSpotInstance API.
       RequestSpotInstancesResult requestResult = ec2.requestSpotInstances(re
questRequest);
        List<SpotInstanceRequest> requestResponses = requestResult.getSpotIn
stanceRequests();

115     // Set up 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
120     // determine when they hit the active state.
        for (SpotInstanceRequest requestResponse : requestResponses) {
          System.out.println("Created Spot Request: "+requestResponse.getSpot
InstanceRequestId());
         spotInstanceRequestIds.add(requestResponse.getSpotInstanceRequestId());
        }
125
        //=============================================================//
        //============= Tag the Spot Requests =======================//
        //=============================================================//

130     // Create the list of tags we want to create
        ArrayList<Tag> requestTags = new ArrayList<Tag>();
        requestTags.add(new Tag("keyname1","value1"));

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

        // Try to tag the Spot request submitted.
140     try {
          ec2.createTags(createTagsRequest_requests);
        } catch (AmazonServiceException e) {
          // Write out any exceptions that may have occurred.
          System.out.println("Error terminating instances");
145       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());
        }
150
        //=============================================================//
        //======= Determining the State of the Spot Request =========//
        //=============================================================//

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

        // Initialize variables.
160     ArrayList<String> instanceIds = new ArrayList<String>();

        do {
          // Create the describeRequest with tall of the request
          // id to monitor (e.g. that we started).
165       DescribeSpotInstanceRequestsRequest describeRequest = new DescribeS
potInstanceRequestsRequest();
          describeRequest.setSpotInstanceRequestIds(spotInstanceRequestIds);

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

          try {
            // Retrieve all of the requests we want to monitor.
```

```
            DescribeSpotInstanceRequestsResult describeResult = ec2.describeS
potInstanceRequests(describeRequest);
175         List<SpotInstanceRequest> describeResponses = describeResult.get
SpotInstanceRequests();

            // Look through each request and determine if they are all
            // in the active state.
            for (SpotInstanceRequest describeResponse : describeResponses) {
180           // 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
              // canceled so we compare against open instead of active.
              if (describeResponse.getState().equals("open")) {
185             anyOpen = true;
                break;
              }

              // Add the instance id to the list we will
190           // eventually terminate.
              instanceIds.add(describeResponse.getInstanceId());
            }
          } catch (AmazonServiceException e) {
            // If we have an exception, ensure we don't break out
195         // of the loop. This prevents the scenario where there
            // was blip on the wire.
            anyOpen = true;
          }

200       try {
            // Sleep for 60 seconds.
            Thread.sleep(60*1000);
          } catch (Exception e) {
            // Do nothing because it woke up early.
205       }
        } while (anyOpen);

        //============================================================//
        //============= Tag the Spot Instances ====================//
210     //============================================================//

        // Create the list of tags we want to create
        ArrayList<Tag> instanceTags = new ArrayList<Tag>();
        instanceTags.add(new Tag("keyname1","value1"));
215
        // Create a tag request for instances.
       CreateTagsRequest createTagsRequest_instances = new CreateTagsRequest();
        createTagsRequest_instances.setResources(instanceIds);
        createTagsRequest_instances.setTags(instanceTags);
220
        // Try to tag the Spot instance started.
        try {
          ec2.createTags(createTagsRequest_instances);
        } catch (AmazonServiceException e) {
225       // 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());
```

```
230       System.out.println("Request ID: " + e.getRequestId());
        }

        //====================================================================//
        //================== Canceling the Request ===================//
235     //====================================================================//

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

250     //====================================================================//
        //================ Terminating any Instances ==================//
        //====================================================================//
        try {
          // Terminate instances.
255       TerminateInstancesRequest terminateRequest = new TerminateInstances
Request(instanceIds);
          ec2.terminateInstances(terminateRequest);
        } catch (AmazonServiceException e) {
          // Write out any exceptions that may have occurred.
          System.out.println("Error terminating instances");
260       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());
        }
265   } // main
    }
```

Tags are a simple first step toward making it easier to manage your own cluster of instances. To read more about tagging Amazon EC2 resources, go to Using Tags in the *Amazon Elastic Compute Cloud User Guide*.

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 is available for download at GitHub.

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

Programming Amazon SWF with the AWS SDK for Java

This section provides information specific to programming Amazon SWF with the SDK for Java.

Topics

Registering an Amazon SWF Domain Using the AWS SDK for Java

Every workflow and activity in Amazon SWF needs a *domain* to run in.

To register an Amazon SWF domain

1. Create a new RegisterDomainRequest object, providing it with at least the domain name and workflow execution retention period (these parameters are both required).
2. Call the AmazonSimpleWorkflowClient.registerDomain method with the **RegisterDomainRequest** object.
3. Catch the DomainAlreadyExistsException if the domain you're requesting already exists (in which case, no action is usually required).

The following code demonstrates this procedure:

```
public void register_swf_domain(AmazonSimpleWorkflowClient swf, String name)
{
    RegisterDomainRequest request = new RegisterDomainRequest().withName(name);

     request.setWorkflowExecutionRetentionPeriodInDays("10");
     try
     {
         swf.registerDomain(request);
     }
     catch (DomainAlreadyExistsException e)
     {
         System.out.println("Domain already exists!");
     }
}
```

Listing Amazon SWF Domains Using the AWS SDK for Java

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());
    }
}
```

Additional Resources

This section lists sources of additional information about using Amazon Web Services and the AWS SDK for Java.

Topics

Home Page for the AWS SDK for Java

For more information about the AWS SDK for Java, go to the home page for the SDK for Java at http://aws.amazon.com/sdkforjava.

SDK Reference Documentation

The AWS SDK for Java API Reference includes the facility to browse and search across all code included with the SDK. It provides thorough documentation, usage examples, and even the ability to browse method source.

AWS Java Developer Blog

Viewing the official AWS Java Developer Blog is a great way to keep up-to-date with new features of the SDK for Java and to learn tips and best practices for development directly from the AWS SDK development team.

AWS Forums

Visit the AWS forums to ask questions or provide feedback about AWS. There is a forum specifically for AWS development in Java as well as forums for individual services such as Amazon S3. AWS engineers monitor the forums and respond to questions, feedback, and issues. You can subscribe to RSS feeds for any of the forums.

To visit the AWS forums, visit aws.amazon.com/forums

AWS Toolkit for Eclipse

If you use the Eclipse IDE, you might be interested in the AWS Toolkit for Eclipse, which provides integration of the SDK for Java with Eclipse, an explorer window to view and work with your AWS resources, and more.

For information about setting up and using the Toolkit for Eclipse, see the AWS Toolkit for Eclipse Getting Started Guide.

SDK for Java Source Code and Samples

The complete source code for the AWS SDK for Java is distributed under the Apache License and is hosted on GitHub at the following URL:

- http://github.com/aws/aws-sdk-java

Included with the SDK code is the code for all of the samples that are packaged with the SDK. For more information about the samples, including instructions for running the samples using the command-line or the Eclipse IDE, see SDK for Java Code Samples (p. 66).

AWS SDK for Java Code Samples

The `samples` directory in the Java SDK download includes a number of code samples that demonstrate how to use the SDK for Java to work with a number of different AWS services.

> **Note**
> If you need instructions for downloading and installing the AWS SDK for Java, see Getting Started (p. 4). Since the complete source code of the SDK for Java is available on GitHub, you can browse the sample code there, as well.

Topics

List of Code Samples

The code samples that are currently available include:

- **AmazonDynamoDB** – demonstrates how to make basic requests to the DynamoDB service.

- **AmazonEC2SpotInstances-Advanced** – demonstrates persistent vs. one-time spot requests, launch groups, and availability groups.
- **AmazonEC2SpotInstances-GettingStarted** – demonstrates how to set up requests for Amazon EC2 Spot Instances, how to determine when they have completed, and how to clean up afterwards.
- **AmazonKinesis** – demonstrates basic use of the Amazon Kinesis client.
- **AmazonKinesisApplication** – makes requests to Amazon Kinesis using the Amazon Kinesis client in the SDK for Java.
- **AmazonS3** – demonstrates how to make basic requests to Amazon Simple Storage Service.
- **AmazonS3TransferProgress** – demonstrates how to track transfer progress for uploads to Amazon S3.
- **AmazonSimpleEmailService** – demonstrates how to send email using the Amazon Simple Email Service JavaMail provider.
- **AmazonSimpleQueueService** – demonstrates how to make basic requests to Amazon Simple Queue Service.
- **AwsCloudFormation** – demonstrates how to make basic requests to AWS CloudFormation.
- **AwsConsoleApp** – demonstrates how to make basic requests to multiple services.
- **AwsFlowFramework** – contains samples for the AWS Flow Framework for Java. These samples may require additional setup. For more information about the framework and the samples that are included in this directory, see AWS Flow Framework for Java Developer Guide

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 your AWS Credentials for Use with the SDK for Java (p. 6). 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).

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
```

<step>
</step>

The sample prints information to standard output. for example:

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

    Welcome to the AWS Java SDK!

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

  You have 0 Amazon EC2 instance(s) running.

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

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

Building and Running the Samples using the Eclipse IDE

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

> **Note**
> 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.

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.

Document History

The following table describes the important changes since the last release of the *AWS SDK for Java Developer Guide*.

Last documentation update: May 14, 2014

Change	Description	Release Date
Updated topics	The introduction (p. 1) and getting started (p. 4) material has been heavily revised to support the new guide structure and now includes guidance about how to Set up your AWS Credentials for Use with the SDK for Java (p. 6). The discussion of SDK for Java Code Samples (p. 66) has been moved into its own topic in the Additional Resources (p. 65) section, and information about how to view the SDK revision history (p. 2) has been moved into the introduction.	May 14, 2014
Updated topics	The overall structure of the SDK for Java documentation has been simplified, and the Getting Started (p. 4) and Additional Resources (p. 65) topics have been updated.	May 9, 2014
New topics	New topics have been added: • Providing Credentials (p. 8) – discusses the various ways that you can specify credentials for use with the SDK for Java. • Using IAM Roles for EC2 Instances (p. 29) – provides information about how to securely specify credentials for applications running on EC2 instances.	May 9, 2014
New topic	This topic tracks recent changes to the *AWS SDK for Java Developer Guide*. It is intended as a companion to the release notes history.	September 9, 2013