# AWS Flow Framework for Java Developer Guide

## Table of Contents

- **What is the AWS Flow Framework for Java?** ................................................................. 1
  - What's in this Guide? ........................................................................................................ 1
  - About Amazon Web Services ......................................................................................... 2
- **Getting Started .................................................................................................................. 3**
  - Setting up the Framework .............................................................................................. 3
    - Installing for Maven .................................................................................................... 3
    - Installing for Eclipse ................................................................................................. 4
  - HelloWorld Application .................................................................................................. 12
    - HelloWorld Activities Implementation ...................................................................... 12
    - HelloWorld Workflow Worker ................................................................................ 13
    - HelloWorld Workflow Starter .................................................................................. 14
  - HelloWorldWorkflow Application ................................................................................. 14
    - HelloWorldWorkflow Activities Worker ................................................................. 16
    - HelloWorldWorkflow Workflow Worker .................................................................. 17
    - HelloWorldWorkflow Workflow and Activities Implementation .......................... 20
    - HelloWorldWorkflow Workflow Starter ................................................................. 22
  - HelloWorldWorkflowAsync Application ...................................................................... 25
    - HelloWorldWorkflowAsync Activities Implementation ......................................... 26
    - HelloWorldWorkflowAsync Workflow Implementation .......................................... 26
    - HelloWorldWorkflowAsync Workflow and Activities Host and Starter ................ 28
  - HelloWorldWorkflowDistributed Application ............................................................ 29
  - HelloWorldWorkflowParallel Application .................................................................. 30
    - HelloWorldWorkflowParallel Activities Worker ..................................................... 31
    - HelloWorldWorkflowParallel Workflow Worker ....................................................... 32
    - HelloWorldWorkflowParallel Workflow and Activities Host and Starter ............. 33
- **How AWS Flow Framework for Java Works .................................................................... 34**
  - Application Structure ................................................................................................. 34
    - Role of the Activity Worker ....................................................................................... 36
    - Role of the Workflow Worker ................................................................................... 36
    - Role of the Workflow Starter .................................................................................... 36
    - How Amazon SWF Interacts with Your Application ................................................. 36
    - For More Information ............................................................................................... 37
  - Reliable Execution ........................................................................................................ 37
    - Providing Reliable Communication ........................................................................... 37
    - Ensuring that Results are Not Lost ............................................................................ 37
    - Handling Failed Distributed Components .................................................................. 38
  - Distributed Execution .................................................................................................. 38
    - Replaying Workflows ............................................................................................... 38
    - Replay and Asynchronous Workflow Methods ......................................................... 39
    - Replay and Workflow Implementation ...................................................................... 40
  - Task Lists and Task Execution ..................................................................................... 40
  - Scalable Applications .................................................................................................. 41
  - Data Exchange Between Activities and Workflows .................................................... 42
    - The Promise<T> Type ............................................................................................... 42
  - Data Converters and Marshaling .................................................................................. 43
  - Data Exchange Between Applications and Workflow Executions ............................ 43
  - Timeout Types ............................................................................................................. 44
    - Timeouts in Workflow and Decision Tasks ............................................................... 44
    - Timeouts in Activity Tasks ......................................................................................... 45
- **Best Practices .................................................................................................................. 47**
  - Making Changes to Decider Code ............................................................................... 47
    - The Replay Process and Code Changes ................................................................... 47
    - Example Scenario ...................................................................................................... 47
    - Solutions .................................................................................................................... 52
What is the AWS Flow Framework for Java?

The Amazon Simple Workflow Service (Amazon SWF) provides a powerful and flexible way for developers to implement distributed asynchronous workflow applications. The AWS Flow Framework is a programming framework that simplifies the process of implementing a distributed asynchronous application while providing all the benefits of Amazon SWF. It is ideal for implementing applications that address a broad range of scenarios including business processes, media encoding, long-running tasks, and background processing.

By using the AWS Flow Framework, you can focus on implementing your workflow logic, while leaving the details of communication and coordination with Amazon SWF to the framework. Behind the scenes, the framework uses Amazon SWF to manage your workflow's execution and make it scalable, reliable, and auditable. AWS Flow Framework-based workflows are highly concurrent, and workflows can be distributed across multiple components—each of which can run as separate processes on separate computers and can be scaled independently. The workflow will continue to run if any of its components are running, making it highly fault tolerant.

Topics

• What's in this Guide? (p. 1)
• About Amazon Web Services (p. 2)

What's in this Guide?

This is the AWS Flow Framework for Java Developer Guide, which aims to provide you with information about how to install, set up, and use the Flow Framework to develop Amazon SWF applications.

Here is a guide to the contents:

Getting Started with the AWS Flow Framework for Java (p. 3)

If you are just starting out with the AWS Flow Framework for Java, you should first read through the Getting Started with the AWS Flow Framework for Java (p. 3) section. It will guide you through downloading and installing the AWS Flow Framework for Java, how to set up your development environment, and lead you through a simple example of creating a workflow using the Flow Framework.

How AWS Flow Framework for Java Works (p. 34)

Introduces basic Amazon SWF and Flow Framework concepts, describing the basic structure of a Flow Framework application and how data is exchanged between parts of a distributed workflow.

AWS Flow Framework for Java Programming Guide (p. 56)

This chapter provides basic programming guidance for developing workflow applications with the AWS Flow Framework for Java, including how to register activity and workflow types, implement workflow clients, create child workflows, handle errors, and more.

Under the Hood (p. 119)

This chapter provides a more in-depth look at the way the AWS Flow Framework for Java works, providing you with additional information about the order of execution of asynchronous workflows and a logical step-through of a standard workflow execution.
Troubleshooting and Debugging Tips (p. 123)

This chapter provides information about common errors that you can use to troubleshoot your workflows, or that you can use to learn to avoid common errors.

AWS Flow Framework for Java Reference (p. 127)

This chapter is a reference to the Annotations, Exceptions and Packages that the AWS Flow Framework for Java adds to the SDK for Java.

Document History (p. 136)

This chapter provides details about major changes to the documentation. New sections and topics as well as significantly revised topics are listed here.

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, see Test-Driving AWS in the Free Usage Tier. To obtain an AWS account, open the AWS home page and then click Sign Up.
Getting Started with the AWS Flow Framework for Java

This section introduces the AWS Flow Framework by walking you through a series of simple example applications that introduce the basic programming model and API. The example applications are based on the standard Hello World application that is used to introduce C and related programming languages. Here is a typical Java implementation of Hello World:

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!"ثن;
    }
}
```

The following is a brief description of the example applications. They include complete source code so you can implement and run the applications yourself. Before starting, you should first configure your development environment and create an AWS Flow Framework for Java project, like in Setting up the AWS Flow Framework for Java (p. 3).

- **HelloWorld Application (p. 12)** introduces workflow applications by implementing Hello World as a standard Java application, but structuring it like a workflow application.
- **HelloWorldWorkflow Application (p. 14)** uses the AWS Flow Framework for Java to convert HelloWorld into an Amazon SWF workflow.
- **HelloWorldWorkflowAsync Application (p. 25)** modifies HelloWorldWorkflow to use an asynchronous workflow method.
- **HelloWorldWorkflowDistributed Application (p. 29)** modifies HelloWorldWorkflowAsync so that the workflow and activity workers can run on separate systems.
- **HelloWorldWorkflowParallel Application (p. 30)** modifies HelloWorldWorkflow to run two activities in parallel.

Setting up the AWS Flow Framework for Java

The AWS Flow Framework for Java is included with the AWS SDK for Java. If you have not already set up the AWS SDK for Java, visit Getting Started in the AWS SDK for Java Developer Guide for information about installing and configuring the SDK itself.

This topic provides information about additional steps required to use the AWS Flow Framework for Java. Steps are provided for Eclipse and Maven.

**Topics**

- Installing for Maven (p. 3)
- Installing for Eclipse (p. 4)

**Installing for Maven**

Amazon provides Amazon SWF build tools in the Maven Central Repository to aid setup of AWS Flow Framework for Java in your Maven projects.
To set up the flow framework for Maven, add the following dependency to your project's pom.xml file:

```xml
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>

<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-swf-build-tools</artifactId>
  <version>1.0</version>
</dependency>
```

The Amazon SWF build tools are open source—to view or download the code or to build the tools yourself, visit the repository at https://github.com/aws/aws-swf-build-tools.

# Installing for Eclipse

If you use the Eclipse IDE, install the AWS Flow Framework for Java using the AWS Toolkit for Eclipse.

**Topics**
- Installing the AWS Toolkit for Eclipse (p. 4)
- Creating an AWS Flow Framework for Java Project (p. 5)

## Installing the AWS Toolkit for Eclipse

Installing the Toolkit for Eclipse is the simplest way to get started with the AWS Flow Framework for Java. To install the Toolkit for Eclipse, see Setting Up the AWS Toolkit for Eclipse in the AWS Toolkit for Eclipse Getting Started Guide.

**Important**

Be sure to include both the AWS SDK for Java and AWS Simple Workflow Tools when selecting which packages to install in Eclipse's Available Software dialog box:
Creating an AWS Flow Framework for Java Project

Creating a properly configured AWS Flow Framework for Java project in Eclipse involves a number of steps:

1. Create an AWS Java project.
2. Enable annotation processing for your project.
3. Enable and configure AspectJ.

Each of these steps will now be described in detail.

**To create an AWS Java project**

1. Launch Eclipse.
2. To select the Java perspective, choose **Window, Open Perspective, Java**.
3. Choose **File, New, AWS Java Project**.
4. Use the AWS Java project wizard to create a new project.

   **Note**
   The first time you create an AWS Java project with Eclipse, the SDK for Java will be automatically downloaded and installed when the project wizard starts.

After creating your AWS Java project, enable *annotation processing* for the project. The AWS Flow Framework for Java includes an annotation processor that generates several key classes based on annotated source code.

**To enable annotation processing**

1. In **Project Explorer**, right-click your project and select **Properties**.
2. In the **Properties** dialog box, navigate to **Java Compiler > Annotation Processing**.
3. Check **Enable project specific settings** (which should also **Enable annotation processing**, but if it doesn't make sure that this option is also checked). Then choose **OK**.
Note
You will need to rebuild your project after enabling annotation processing.
Enabling and Configuring AspectJ

Next, you should enable and configure AspectJ. Certain AWS Flow Framework for Java annotations such as @Asynchronous require AspectJ. You don’t need to use AspectJ directly, but you must enable it with either load-time weaving or compile-time weaving.

**Note**
The recommended approach is to use load-time weaving.

**Topics**
- Prerequisites (p. 8)
- Configuring AspectJ Load-Time Weaving (p. 8)
- AspectJ Compile-Time Weaving (p. 9)
- Working around issues with AspectJ and Eclipse (p. 11)

**Prerequisites**

Before configuring AspectJ, you need the AspectJ version that matches your Java version:

- If you are using Java 8, download the latest AspectJ 1.8.X release.
- If you are using Java 7, download the latest AspectJ 1.7.X release.
- If you are using Java 6, download the latest AspectJ 1.6.X release.

You can download either of these versions of AspectJ from the Eclipse download page.

After you have finished downloading AspectJ, execute the downloaded .jar file to install AspectJ. The AspectJ installation will ask you where you would like to install the binaries, and on the final screen, will provide recommended steps for completing the installation. *Remember the location of the aspectjweaver.jar file; you'll need it to configure AspectJ in Eclipse.*

**Configuring AspectJ Load-Time Weaving**

To configure AspectJ load-time weaving for your AWS Flow Framework for Java project, first designate the AspectJ JAR file as a Java agent, and then configure it by adding an aop.xml file to your project.

**To add AspectJ as a Java agent**

1. To open the Preferences dialog box, choose Window, Preferences.
2. Navigate to Java > Installed JREs.
3. Select the appropriate JRE and choose Edit.
4. In the Default VM arguments box, enter the path to the installed AspectJ binary. This will be a path such as /home/user/aspectj1.7/lib/aspectjweaver.jar, depending on your operating system and on the version of AspectJ you downloaded.

   On Linux, macOS, or Unix use:

   `-javaagent:/your_path/aspectj/lib/aspectjweaver.jar`

   On Windows, use a standard Windows-style path instead:

   `-javaagent:C:\your_path\aspectj\lib\aspectjweaver.jar`
To configure AspectJ for AWS Flow Framework for Java, add an `aop.xml` file to the project.

**To add an `aop.xml` file**

1. In your project's `src` directory, add a directory named `META-INF`.
2. Add a file named `aop.xml` to `META-INF` with the following contents.

```xml
<aspectj>
  <aspects>
    <aspect name="com.amazonaws.services.simpleworkflow.flow.aspectj.AsynchronousAspect"/>
    <aspect name="com.amazonaws.services.simpleworkflow.flow.aspectj.ExponentialRetryAspect"/>
  </aspects>
  <weaver options="-verbose">
    <include within="MySimpleWorkflow.*"/>
  </weaver>
</aspectj>
```

The value of `<include within="/"/>` depends on how you name your project's packages. The above example assumes that the project's packages followed the pattern `MySimpleWorkflow.*`. Use a value appropriate for your own project's packages.

**AspectJ Compile-Time Weaving**

To enable and configure AspectJ compile-time weaving, you must first install the AspectJ developer tools for Eclipse, which are available from http://www.eclipse.org/aspectj/downloads.php.

**To install the AspectJ Developer Tools in Eclipse**

1. On the Help menu, choose Install New Software.
2. In the Available Software dialog box, enter `http://download.eclipse.org/tools/ajdt/version/dev/update`, where `version` represents your Eclipse version number. For example, if you are using Eclipse 4.6, you would enter: `http://download.eclipse.org/tools/ajdt/46/dev/update`
Important
Be sure that the AspectJ version matches your Eclipse version, or installation of AspectJ will fail.

3. Choose **Add** to add the location. Once the location is added, the AspectJ developer tools will be listed.

4. Choose **Select All** to select all of the AspectJ developer tools, then choose **Next** to install them.

   **Note**
   You will need to restart Eclipse to complete the installation.

You must then configure your project.

To configure your project for AspectJ compile-time weaving

1. In **Project Explorer**, right-click your project and select **Configure > Convert to AspectJ Project**.
The AspectJ Runtime Library will be added to your project.

2. Right-click your project again and then choose Properties.
3. Choose AspectJ Build and then choose the Aspect Path tab.
4. Choose Add External JARs and add the AWS SDK for Java JAR file to your project's Aspect Path.

**Note**
The AWS Toolkit for Eclipse installs the AWS SDK for Java JAR file in your workspace, in the .metadata/.plugins/com.amazonaws.eclipse.core/aws-java-sdk/AWS Version/lib directory, where you replace AWS Version with the installed AWS SDK version number. Otherwise, you can use the JAR file that is included with the regular AWS SDK installation, which is in the lib directory.

**Working around issues with AspectJ and Eclipse**

The AspectJ Eclipse plug-in has an issue that can prevent generated code from being compiled. The fastest way to force generated code to be recognized after you recompile it is to change the order of the source directory that contains the generated code on the Order and Export tab of the Java Build Path settings page (for example, you can set the default to apt/java).
HelloWorld Application

To introduce the way Amazon SWF applications are structured, we'll create a Java application that behaves like a workflow, but that runs locally in a single process. No connection to Amazon Web Services will be needed.

**Note**
The HelloWorldWorkflow (p. 14) example builds upon this one, connecting to Amazon SWF to handle management of the workflow.

A workflow application consists of three basic components:

- An **activities worker** supports a set of **activities**, each of which is a method that executes independently to perform a particular task.
- A **workflow worker** orchestrates the activities' execution and manages data flow. It is a programmatic realization of a workflow topology, which is basically a flow chart that defines when the various activities execute, whether they execute sequentially or concurrently, and so on.
- A **workflow starter** starts a workflow instance, called an execution, and can interact with it during execution.

HelloWorld is implemented as three classes and two related interfaces, which are described in the following sections. Before starting, you should set up your development environment and create a new AWS Java project as described in Setting up the AWS Flow Framework for Java (p. 3). The packages used for the following walkthroughs are all named helloWorld.XYZ. To use those names, set the within attribute in aop.xml as follows:

```xml
...<weaver options="-verbose">
   <include within="helloWorld.."/>
</weaver>
```

To implement HelloWorld, create a new Java package in your AWS SDK project named helloWorld.HelloWorld and add the following files:

- An interface file named GreeterActivities.java
- A class file named GreeterActivitiesImpl.java, which implements the activities worker.
- An interface file named GreeterWorkflow.java.
- A class file named GreeterWorkflowImpl.java, which implements the workflow worker.
- A class file named GreeterMain.java, which implements the workflow starter.

The details are discussed in the following sections and include the complete code for each component, which you can add to the appropriate file.

**HelloWorld Activities Implementation**

HelloWorld breaks the overall task of printing a "Hello World!" greeting to the console into three tasks, each of which is performed by an **activity method**. The activity methods are defined in the GreeterActivities interface, as follows.

```java
public interface GreeterActivities {
    public String getName();
    public String getGreeting(String name);
    public void say(String what);
}
```
HelloWorld has one activity implementation, GreeterActivitiesImpl, which provides the GreeterActivities methods as shown:

```java
class GreeterActivitiesImpl implements GreeterActivities {
    @Override
    public String getName() {
        return "World";
    }

    @Override
    public String getGreeting(String name) {
        return "Hello " + name + "!";
    }

    @Override
    public void say(String what) {
        System.out.println(what);
    }
}
```

Activities are independent of each other and can often be used by different workflows. For example, any workflow can use the say activity to print a string to the console. Workflows can also have multiple activity implementations, each performing a different set of tasks.

### HelloWorld Workflow Worker

To print "Hello World!" to the console, the activity tasks must execute in sequence in the correct order with the correct data. The HelloWorld workflow worker orchestrates the activities’ execution based on a simple linear workflow topology, which is shown in the following figure.

The three activities execute in sequence, and the data flows from one activity to the next.

The HelloWorld workflow worker has a single method, the workflow’s entry point, which is defined in the GreeterWorkflow interface, as follows:

```java
public interface GreeterWorkflow {
    public void greet();
}
```

The GreeterWorkflowImpl class implements this interface, as follows:

```java
class GreeterWorkflowImpl implements GreeterWorkflow{
    private GreeterActivities operations = new GreeterActivitiesImpl();

    public void greet() {
        String name = operations.getName();
        String greeting = operations.getGreeting(name);
        operations.say(greeting);
    }
}
```

The greet method implements HelloWorld topology by creating an instance of GreeterActivitiesImpl, calling each activity method in the correct order, and passing the appropriate data to each method.
HelloWorld Workflow Starter

A workflow starter is an application that starts a workflow execution, and might also communicate with the workflow while it is executing. The `GreeterMain` class implements the HelloWorld workflow starter, as follows:

```java
public class GreeterMain {
    public static void main(String[] args) {
        GreeterWorkflow greeter = new GreeterWorkflowImpl();
        greeter.greet();
    }
}
```

`GreeterMain` creates an instance of `GreeterWorkflowImpl` and calls `greet` to run the workflow worker. Run `GreeterMain` as a Java application and you should see "Hello World!" in the console output.

HelloWorldWorkflow Application

Although the basic HelloWorld (p. 12) example is structured like a workflow, it differs from an Amazon SWF workflow in several key respects:

<table>
<thead>
<tr>
<th>Conventional and Amazon SWF Workflow Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HelloWorld</strong></td>
</tr>
<tr>
<td>Runs locally as a single process.</td>
</tr>
<tr>
<td>Activities are synchronous methods, which block until they complete.</td>
</tr>
<tr>
<td>The workflow worker interacts with an activities worker by calling the appropriate method.</td>
</tr>
<tr>
<td>The workflow starter interacts with workflow worker by calling the appropriate method.</td>
</tr>
</tbody>
</table>

You could implement a distributed asynchronous workflow application from scratch, for example, by having your workflow worker interact with an activities worker directly through web services calls. However, you must then implement all the complicated code required to manage the asynchronous execution of multiple activities, handle the data flow, and so on. The AWS Flow Framework for Java and Amazon SWF take care of all those details, which allows you to focus on implementing the business logic.

`HelloWorldWorkflow` is a modified version of HelloWorld that runs as an Amazon SWF workflow. The following figure summarizes how the two applications work.
HelloWorld runs as a single process and the starter, workflow worker, and activities worker interact by using conventional method calls. With HelloWorldWorkflow, the starter, workflow worker, and activities worker are distributed components that interact through Amazon SWF by using HTTP requests. Amazon SWF manages the interaction by maintaining lists of workflow and activities tasks, which it dispatches to the respective components. This section describes how the framework works for HelloWorldWorkflow.

HelloWorldWorkflow is implemented by using the AWS Flow Framework for Java API, which handles the sometimes complicated details of interacting with Amazon SWF in the background and simplifies the development process considerably. You can use the same project that you did for HelloWorld, which is already configured for AWS Flow Framework for Java applications. However, to run the application, you must set up an Amazon SWF account, as follows:

- Sign up for an AWS account, if you don't already have one, at Amazon Web Services.
- Assign your account's Access ID and secret ID to the AWS_ACCESS_KEY_ID and AWS_SECRET_KEY environment variables, respectively. It's a good practice to not expose the literal key values in your code. Storing them in environment variables is a convenient way to handle the issue.
- Sign up for Amazon SWF account at Amazon Simple Workflow Service.
- Log into the AWS Management Console and select the Amazon SWF service.
- Choose Manage Domains in the upper right corner and register a new Amazon SWF domain. A domain is a logical container for your application resources, such as workflow and activity types, and workflow executions. You can use any convenient domain name, but the walkthroughs use "helloWorldWalkthrough".

For more details about how to manage Amazon SWF workflows, see Getting Set Up.

To implement the HelloWorldWorkflow, create a copy of the helloWorld.HelloWorld package in your project directory and name it helloWorld.HelloWorldWorkflow. The following sections describe how to modify the original HelloWorld code to use the AWS Flow Framework for Java and run as an Amazon SWF workflow application.
HelloWorldWorkflow Activities Worker

HelloWorld implemented its activities worker as a single class. An AWS Flow Framework for Java activities worker has three basic components:

- The *activity methods*—which perform the actual tasks—are defined in an interface and implemented in a related class.
- An *ActivityWorker* class manages the interaction between the activity methods and Amazon SWF.
- An *activities host* application registers and starts the activities worker, and handles cleanup.

This section discusses the activity methods; the other two classes are discussed later.

HelloWorldWorkflow defines the activities interface in `GreeterActivities`, as follows:

```java
import com.amazonaws.services.simpleworkflow.flow.annotations.Activities;
import com.amazonaws.services.simpleworkflow.flow.annotations.ActivityRegistrationOptions;
@ActivityRegistrationOptions(defaultTaskScheduleToStartTimeoutSeconds = 300,
                             defaultTaskStartToCloseTimeoutSeconds = 10)
@Activities(version="1.0")
public interface GreeterActivities {
    public String getName();
    public String getGreeting(String name);
    public void say(String what);
}
```

This interface wasn't strictly necessary for HelloWorld, but it is for an AWS Flow Framework for Java application. Notice that the interface definition itself hasn't changed. However, you must apply two AWS Flow Framework for Java annotations, `@ActivityRegistrationOptions` (p. 128) and `@Activities` (p. 127), to the interface definition. The annotations provide configuration information and direct the AWS Flow Framework for Java annotation processor to use the interface definition to generate an *activities client* class, which is discussed later.

`@ActivityRegistrationOptions` has several named values that are used to configure the activities' behavior. HelloWorldWorkflow specifies two timeouts:

- `defaultTaskScheduleToStartTimeoutSeconds` specifies how long the tasks can be queued in the activities task list, and is set to 300 seconds (5 minutes).
- `defaultTaskStartToCloseTimeoutSeconds` specifies the maximum time the activity can take to perform the task and is set to 10 seconds.

These timeouts ensure that the activity completes its task in a reasonable amount of time. If either timeout is exceeded, the framework generates an error and the workflow worker must decide how to handle the issue. For a discussion of how to handle such errors, see Error Handling (p. 101).

`@Activities` has several values, but typically it just specifies the activities' version number, which allows you to keep track of different generations of activity implementations. If you change an activity interface after you have registered it with Amazon SWF, including changing the `@ActivityRegistrationOptions` values, you must use a new version number.

HelloWorldWorkflow implements the activity methods in `GreeterActivitiesImpl`, as follows:

```java
public class GreeterActivitiesImpl implements GreeterActivities {
    @Override
    public String getName() {
```

API Version 2012-01-25
16
Notice that the code is identical to the HelloWorld implementation. At its core, an AWS Flow Framework activity is just a method that executes some code and perhaps returns a result. The difference between a standard application and an Amazon SWF workflow application lies in how the workflow executes the activities, where the activities execute, and how the results are returned to the workflow worker.

**HelloWorldWorkflow Workflow Worker**

An Amazon SWF workflow worker has three basic components.

- A *workflow implementation*, which is a class that performs the workflow-related tasks.
- An *activities client* class, which is basically a proxy for the activities class and is used by a workflow implementation to execute activity methods asynchronously.
- A *WorkflowWorker* class, which manages the interaction between the workflow and Amazon SWF.

This section discusses the workflow implementation and activities client; the WorkflowWorker class is discussed later.

HelloWorldWorkflow defines the workflow interface in GreeterWorkflow, as follows:

```java
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 3600)
public interface GreeterWorkflow {
    @Execute(version = "1.0")
    public void greet();
}
```

This interface also isn't strictly necessary for HelloWorld but is essential for an AWS Flow Framework for Java application. You must apply two AWS Flow Framework for Java annotations, [@Workflow](p. 131) and [@WorkflowRegistrationOptions](p. 131), to the workflow interface definition. The annotations provide configuration information and also direct the AWS Flow Framework for Java annotation processor to generate a workflow client class based on the interface, as discussed later.

@Workflow has one optional parameter, *dataConverter*, which is often used with its default value, NullDataConverter, which indicates that JsonDataConverter should be used.

@WorkflowRegistrationOptions also has a number of optional parameters that can be used to configure the workflow worker. Here, we set defaultExecutionStartToCloseTimeoutSeconds—which specifies how long the workflow can run—to 3600 seconds (1 hour).

The GreeterWorkflow interface definition differs from HelloWorld in one important way, the @Execute (p. 129) annotation. Workflow interfaces specify the methods that can be called by
Applications such as the workflow starter and are limited to a handful of methods, each with a particular role. The framework doesn't specify a name or parameter list for workflow interface methods; you use a name and parameter list that is suitable for your workflow and apply an AWS Flow Framework for Java annotation to identify the method's role.

@Execute has two purposes:

- It identifies `greet` as the workflow's entry point—the method that the workflow starter calls to start the workflow. In general, an entry point can take one or more parameters, which allows the starter to initialize the workflow, but this example doesn't require initialization.
- It specifies the workflow's version number, which allows you to keep track of different generations of workflow implementations. To change a workflow interface after you have registered it with Amazon SWF, including changing the timeout values, you must use a new version number.

For information about the other methods that can be included in a workflow interface, see Workflow and Activity Contracts (p. 57).

HelloWorldWorkflow implements the workflow in GreeterWorkflowImpl, as follows:

```java
import com.amazonaws.services.simpleworkflow.flow.core.Promise;

public class GreeterWorkflowImpl implements GreeterWorkflow {
    private GreeterActivitiesClient operations = new GreeterActivitiesClientImpl();

    public void greet() {
        Promise<String> name = operations.getName();
        Promise<String> greeting = operations.getGreeting(name);
        operations.say(greeting);
    }
}
```

The code is similar to HelloWorld, but with two important differences.

- `GreeterWorkflowImpl` creates an instance of `GreeterActivitiesClientImpl`, the activities client, instead of `GreeterActivitiesImpl`, and executes activities by calling methods on the client object.
- The name and greeting activities return `Promise<String>` objects instead of `String` objects.

HelloWorld is a standard Java application that runs locally as a single process, so `GreeterWorkflowImpl` can implement the workflow topology by simply creating an instance of `GreeterActivitiesImpl`, calling the methods in order, and passing the return values from one activity to the next. With an Amazon SWF workflow, an activity's task is still performed by an activity method from `GreeterActivitiesImpl`. However, the method doesn't necessarily run in the same process as the workflow—it might not even run on the same system—and the workflow needs to execute the activity asynchronously. These requirements raise the following issues:

- How to execute an activity method that might be running in a different process, perhaps on a different system.
- How to execute an activity method asynchronously.
- How to manage activities' input and return values. For example, if the Activity A return value is an input to Activity B, you must ensure that Activity B doesn't execute until Activity A is complete.

You can implement a variety of workflow topologies through the application's control flow by using familiar Java flow control combined with the activities client and the `Promise<T>`.
Activities Client

`GreeterActivitiesClientImpl` is basically a proxy for `GreeterActivitiesImpl` that allows a workflow implementation to execute the `GreeterActivitiesImpl` methods asynchronously.

The `GreeterActivitiesClient` and `GreeterActivitiesClientImpl` classes are generated automatically for you using the information provided in the annotations applied to your `GreeterActivities` class. You don't need to implement these yourself.

**Note**
Eclipse generates these classes when you save your project. You can view the generated code in the `.apt_generated` subdirectory of your project directory.
To avoid compilation errors in your `GreeterWorkflowImpl` class, it is a good practice to move the `.apt_generated` directory to the top of the `Order and Export` tab of the `Java Build Path` dialog box.

A workflow worker executes an activity by calling the corresponding client method. The method is asynchronous and immediately returns a `Promise<T>` object, where `T` is the activity's return type. The returned `Promise<T>` object is basically a placeholder for the value that the activity method will eventually return.

- When the activities client method returns, the `Promise<T>` object is initially in an *unready state*, which indicates that the object doesn't yet represent a valid return value.
- When the corresponding activity method completes its task and returns, the framework assigns the return value to the `Promise<T>` object and puts it in the *ready state*.

Promise<T> Type

The primary purpose of `Promise<T>` objects is to manage data flow between asynchronous components and control when they execute. It relieves your application of the need to explicitly manage synchronization or depend on mechanisms such as timers to ensure that asynchronous components don't execute prematurely. When you call an activities client method, it immediately returns but the framework defers executing the corresponding activity method until any input `Promise<T>` objects are ready and represent valid data.

From `GreeterWorkflowImpl` perspective, all three activities client methods return immediately. From the `GreeterActivitiesImpl` perspective, the framework doesn't call `getGreeting` until `name` completes, and doesn't call `say` until `getGreeting` completes.

By using `Promise<T>` to pass data from one activity to the next, `HelloWorldWorkflow` not only ensures that activity methods don't attempt to use invalid data, it also controls when the activities execute and implicitly defines the workflow topology. Passing each activity's `Promise<T>` return value to the next activity requires the activities to execute in sequence, defining the linear topology discussed earlier. With AWS Flow Framework for Java, you don't need to use any special modeling code to define even complex topologies, just standard Java flow control and `Promise<T>`. For an example of how to implement a simple parallel topology, see `HelloWorldWorkflowParallel Activities Worker` (p. 31).

**Note**
When an activity method such as `say` doesn't return a value, the corresponding client method returns a `Promise<Void>` object. The object doesn't represent data, but it is initially unready and becomes ready when the activity completes. You can therefore pass a `Promise<Void>` object to other activities client methods to ensure that they defer execution until the original activity completes.

`Promise<T>` allows a workflow implementation to use activities client methods and their return values much like synchronous methods. However, you must be careful about accessing a `Promise<T>` object's
value. Unlike the Java Future\langle T\rangle type, the framework handles synchronization for Promise\langle T\rangle, not the application. If you call Promise\langle T\rangle\text{.}get and the object isn't ready, get throws an exception. Notice that HelloWorldWorkflow never accesses a Promise\langle T\rangle object directly; it simply passes the objects from one activity to the next. When an object becomes ready, the framework extracts the value and passes it to the activity method as a standard type.

Promise\langle T\rangle objects should be accessed only by asynchronous code, where the framework guarantees that the object is ready and represents a valid value. HelloWorldWorkflow deals with this issue by passing Promise\langle T\rangle objects only to activities client methods. You can access a Promise\langle T\rangle object's value in your workflow implementation by passing the object to an asynchronous workflow method, which behaves much like an activity. For an example, see HelloWorldWorkflowAsync Application (p. 25).

HelloWorldWorkflow Workflow and Activities Implementation

The workflow and activities implementations have associated worker classes, ActivityWorker and WorkflowWorker. They handle communication between Amazon SWF and the activities and workflow implementations by polling the appropriate Amazon SWF task list for tasks, executing the appropriate method for each task, and managing the data flow. For details, see AWS Flow Framework Basic Concepts: Application Structure (p. 34)

To associate the activity and workflow implementations with the corresponding worker objects, you implement one or more worker applications which:

- Register workflows or activities with Amazon SWF.
- Create worker objects and associate them with the workflow or activity worker implementations.
- Direct the worker objects to start communicating with Amazon SWF.

If you want to run the workflow and activities as separate processes, you must implement separate workflow and activities worker hosts. For an example, see HelloWorldWorkflowDistributed Application (p. 29). For simplicity, HelloWorldWorkflow implements a single worker host that runs activities and workflow workers in the same process, as follows:

```java
import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient;
import com.amazonaws.services.simpleworkflow.flow.ActivityWorker;
import com.amazonaws.services.simpleworkflow.flow.WorkflowWorker;

public class GreeterWorker {
    public static void main(String[] args) throws Exception {
        ClientConfiguration config = new ClientConfiguration().withSocketTimeout(70*1000);

        String swfAccessId = System.getenv("AWS_ACCESS_KEY_ID");
        String swfSecretKey = System.getenv("AWS_SECRET_KEY");
        AWSCredentials awsCredentials = new BasicAWSCredentials(swfAccessId, swfSecretKey);

        AmazonSimpleWorkflow service = new AmazonSimpleWorkflowClient(awsCredentials, config);
        service.setEndpoint("https://swf.us-east-1.amazonaws.com");

        String domain = "helloWorldWalkthrough";
        String taskListToPoll = "HelloWorldList";
        ActivityWorker aw = new ActivityWorker(service, domain, taskListToPoll);
```
GreeterWorker has no HelloWorld counterpart, so you must add a Java class named GreeterWorker to the project and copy the example code to that file.

The first step is to create and configure an AmazonSimpleWorkflowClient object, which invokes the underlying Amazon SWF service methods. To do so, GreeterWorker:

1. Creates a ClientConfiguration object and specifies a socket timeout of 70 seconds. This value specifies long to wait for data to be transferred over an established open connection before closing the socket.
2. Creates a BasicAWSCredentials object to identify the Amazon AWS account and passes the account keys to the constructor. For convenience, and to avoid exposing them as plain text in the code, the keys are stored as environment variables.
3. Creates an AmazonSimpleWorkflowClient object to represent the workflow, and passes the BasicAWSCredentials and ClientConfiguration objects to the constructor.
4. Sets the client object's service endpoint URL. Amazon SWF is currently available in all AWS regions.

For convenience, GreeterWorker defines two string constants.

- domain is the workflow's Amazon SWF domain name, which you created when you set up your Amazon SWF account. HelloWorldWorkflow assumes that you are running the workflow in the "helloWorldWalkthrough" domain.
- taskListToPoll is the name of the task lists that Amazon SWF uses to manage communication between the workflow and activities workers. You can set the name to any convenient string. HelloWorldWorkflow uses "HelloWorldList" for both workflow and activity task lists. Behind the scenes, the names end up in different namespaces, so the two task lists are distinct.

GreeterWorker uses the string constants and the AmazonSimpleWorkflowClient object to create worker objects, which manage the interaction between the activities and worker implementations and Amazon SWF. In particular, the worker objects handle the task of polling the appropriate task list for tasks.

GreeterWorker creates an ActivityWorker object and configures it to handle GreeterActivitiesImpl by adding a new class instance. GreeterWorker then calls the ActivityWorker object's start method, which directs the object to start polling the specified activities task list.

GreeterWorker creates a WorkflowWorker object and configures it to handle GreeterWorkflowImpl by adding the class file name, GreeterWorkflowImpl.class. It then calls the WorkflowWorker object's start method, which directs the object to start polling the specified workflow task list.

You can run GreeterWorker successfully at this point. It registers the workflow and activities with Amazon SWF and starts the worker objects polling their respective task lists. To verify this, run GreeterWorker and go to the Amazon SWF console and select helloWorldWalkthrough from the list of domains. If you choose Workflow Types in the Navigation pane, you should see GreeterWorkflow.greet:

```
aw.addActivitiesImplementation(new GreeterActivitiesImpl());
aw.start();

WorkflowWorker wfw = new WorkflowWorker(service, domain, taskListToPoll);
wwf.addWorkflowImplementationType(GreeterWorkflowImpl.class);
wwf.start();
```
If you choose Activity Types, the GreeterActivities methods are displayed:

However, if you choose Workflow Executions, you will not see any active executions. Although the workflow and activities workers are polling for tasks, we have not yet started a workflow execution.

HelloWorldWorkflow Starter

The final piece of the puzzle is to implement a workflow starter, which is an application that starts the workflow execution. The execution state is stored by Amazon SWF, so that you can view its history and execution status. HelloWorldWorkflow implements a workflow starter by modifying the GreeterMain class, as follows:

```java
import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient;
```
public class GreeterMain {

    public static void main(String[] args) throws Exception {
        ClientConfiguration config = new ClientConfiguration().withSocketTimeout(70*1000);

        String swfAccessId = System.getenv("AWS_ACCESS_KEY_ID");
        String swfSecretKey = System.getenv("AWS_SECRET_KEY");
        AWSCredentials awsCredentials = new BasicAWSCredentials(swfAccessId, swfSecretKey);

        AmazonSimpleWorkflow service = new AmazonSimpleWorkflowClient(awsCredentials, config);
        service.setEndpoint("https://swf.us-east-1.amazonaws.com");

        String domain = "helloWorldWalkthrough";

        GreeterWorkflowClientExternalFactory factory = new GreeterWorkflowClientExternalFactoryImpl(service, domain);
        GreeterWorkflowClientExternal greeter = factory.getClient("someID");
        greeter.greet();
    }
}

GreeterMain creates an AmazonSimpleWorkflowClient object by using the same code as GreeterWorker. It then creates a GreeterWorkflowClientExternal object, which acts as a proxy for the workflow in much the same way that the activities client created in GreeterWorkflowClientImpl acts as a proxy for the activity methods. Rather than create a workflow client object by using new, you must:

1. Create an external client factory object and pass the AmazonSimpleWorkflowClient object and Amazon SWF domain name to the constructor. The client factory object is created by the framework's annotation processor, which creates the object name by simply appending "ClientExternalFactoryImpl" to the workflow interface name.

2. Create an external client object by calling the factory object's getClient method, which creates the object name by appending "ClientExternal" to the workflow interface name. You can optionally pass getClient a string which Amazon SWF will use to identify this instance of the workflow. Otherwise, Amazon SWF represents a workflow instance by using a generated GUID.

The client returned from the factory will only create workflows that are named with the string passed into the getClient method, (the client returned from the factory already has state in Amazon SWF). To run a workflow with a different id, you need to go back to the factory and create a new client with the different id specified.

The workflow client exposes a greet method that GreeterMain calls to begin the workflow, as greet() was the method specified with the @Execute annotation.

    Note
    The annotation processor also creates an internal client factory object that is used to create child workflows. For details, see Child Workflow Executions (p. 85).

Shut down GreeterWorker for the moment if it is still running, and run GreeterMain. You should now see someID on the Amazon SWF console's list of active workflow executions:
If you choose someID and choose the Events tab, the events are displayed:

If you started GreeterWorker earlier, and it is still running, you will see a longer event list for reasons discussed shortly. Stop GreeterWorker and try running GreaterMain again.

The Events tab shows only two events:

- WorkflowExecutionStarted indicates that the workflow has started executing.
- DecisionTaskScheduled indicates that Amazon SWF has queued the first decision task.

The reason that the workflow is blocked at the first decision task is that the workflow is distributed across two applications, GreeterMain and GreeterWorker. GreeterMain started the workflow execution, but GreeterWorker isn't running, so the workers aren't polling the lists and executing tasks. You can run either application independently, but you need both for workflow execution to proceed beyond the first decision task. If you now run GreeterWorker, the workflow and activity workers will start polling and the various tasks will be completed rapidly. If you now check the Events tab, the first batch of events is displayed.
You can choose individual events to get more information. By the time you've finished looking, the workflow should have printed "Hello World!" to your console.

After the workflow completes, it no longer appears on the list of active executions. However, if you want to review it, choose the Closed execution status button and then choose List Executions. This displays all the completed workflow instances in the specified domain (helloWorldWalkthrough) that have not exceeded their retention time, which you specified when you created the domain.

Notice that each workflow instance has a unique Run ID value. You can use the same Execution ID for different workflow instances, but only for one active execution at a time.

HelloWorldWorkflowAsync Application

Sometimes, it's preferable to have a workflow perform certain tasks locally instead of using an activity. However, workflow tasks often involve processing the values represented by Promise<T> objects. If you pass a Promise<T> object to a synchronous workflow method, the method executes immediately but it can't access the Promise<T> object's value until the object is ready. You could poll Promise<T>.isReady until it returns true, but that's inefficient and the method might block for a long time. A better approach is to use an asynchronous method.
An asynchronous method is implemented much like a standard method—often as a member of the workflow implementation class—and runs in the workflow implementation's context. You designate it as an asynchronous method by applying an @Asynchronous annotation, which directs the framework to treat it much like an activity.

- When a workflow implementation calls an asynchronous method, it returns immediately. Asynchronous methods typically return a Promise<T> object, which becomes ready when the method completes.
- If you pass an asynchronous method one or more Promise<T> objects, it defers execution until all the input objects are ready. An asynchronous method can therefore access its input Promise<T> values without risking an exception.

**Note**
Because of the way that the AWS Flow Framework for Java executes the workflow, asynchronous methods typically execute multiple times, so you should use them only for quick low-overhead tasks. You should use activities to perform lengthy tasks such as large computations. For details, see AWS Flow Framework Basic Concepts: Distributed Execution (p. 38).

This topic is a walkthrough of HelloWorldWorkflowAsync, a modified version of HelloWorldWorkflow that replaces one of the activities with an asynchronous method. To implement the application, create a copy of the helloWorld.HelloWorldWorkflow package in your project directory and name it helloWorld.HelloWorldWorkflowAsync. The following sections describe how to modify the original HelloWorldWorkflow code to use an asynchronous method.

**HelloWorldWorkflowAsync Activities Implementation**

HelloWorldWorkflowAsync implements its activities worker interface in GreeterActivities, as follows:

```java
import com.amazonaws.services.simpleworkflow.flow.annotations.Activities;
import com.amazonaws.services.simpleworkflow.flow.annotations.ActivityRegistrationOptions;

@Activities(version="2.0")
@ActivityRegistrationOptions(defaultTaskScheduleToStartTimeoutSeconds = 300,
   defaultTaskStartToCloseTimeoutSeconds = 10)
public interface GreeterActivities {
  public String getName();
  public void say(String what);
}
```

This interface is similar to the one used by HelloWorldWorkflow, with the following exceptions:
- It omits the getGreeting activity; that task is now handled by an asynchronous method.
- The version number is set to 2.0. After you have registered an activities interface with Amazon SWF, you can't modify it unless you change the version number.

The remaining activity method implementations are identical to HelloWorldWorkflow. Just delete getGreeting from GreeterActivitiesImpl.

**HelloWorldWorkflowAsync Workflow Implementation**

HelloWorldWorkflowAsync defines the workflow interface as follows:

```java
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
```
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 3600)
public interface GreeterWorkflow {
    @Execute(version = "2.0")
    public void greet();
}

The interface is identical to HelloWorldWorkflow apart from a new version number. As with activities, if you want to change a registered workflow, you must change its version.

HelloWorldWorkflowAsync implements the workflow as follows:

import com.amazonaws.services.simpleworkflow.flow.annotations.Asynchronous;
import com.amazonaws.services.simpleworkflow.flow.core.Promise;

public class GreeterWorkflowImpl implements GreeterWorkflow {
    private GreeterActivitiesClient operations = new GreeterActivitiesClientImpl();

    @Override
    public void greet() {
        Promise<String> name = operations.getName();
        Promise<String> greeting = getGreeting(name);
        operations.say(greeting);
    }

    @Asynchronous
    private Promise<String> getGreeting(Promise<String> name) {
        String returnString = "Hello " + name.get() + "!";
        return Promise.asPromise(returnString);
    }
}

HelloWorldWorkflowAsync replaces the getGreeting activity with a getGreeting asynchronous method but the greet method works in much the same way:

1. Execute the getName activity, which immediately returns a Promise<String> object, name, that represents the name.
2. Call the getGreeting asynchronous method and pass it the name object. getGreeting immediately returns a Promise<String> object, greeting, that represents the greeting.
3. Execute the say activity and pass it the greeting object.
4. When getName completes, name becomes ready and getGreeting uses its value to construct the greeting.
5. When getGreeting completes, greeting becomes ready and say prints the string to the console.

The difference is that, instead of calling the activities client to execute a getGreeting activity, greet calls the asynchronous getGreeting method. The net result is the same, but the getGreeting method works somewhat differently than the getGreeting activity.

- The workflow worker uses standard function call semantics to execute getGreeting. However, the asynchronous execution of the activity is mediated by Amazon SWF.
- getGreeting runs in the workflow implementation's process.
- getGreeting returns a Promise<String> object rather than a String object. To get the String value held by the Promise, you call its get() method. However, since the activity is being run
asynchronously, its return value might not be ready immediately; `get()` will raise an exception until the return value of the asynchronous method is available.

For more information about how Promise works, see AWS Flow Framework Basic Concepts: Data Exchange Between Activities and Workflows (p. 42).

getGreeting creates a return value by passing the greeting string to the static `Promise.asPromise` method. This method creates a `Promise<T>` object of the appropriate type, sets the value, and puts it in the ready state.

**HelloWorldWorkflowAsync Workflow and Activities Host and Starter**

HelloWorldWorkflowAsync implements `GreeterWorker` as the host class for the workflow and activity implementations. It is identical to the HelloWorldWorkflow implementation except for the `taskListToPoll` name, which is set to "HelloWorldAsyncList".

```java
import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient;
import com.amazonaws.services.simpleworkflow.flow.ActivityWorker;
import com.amazonaws.services.simpleworkflow.flow.WorkflowWorker;

public class GreeterWorker {
    public static void main(String[] args) throws Exception {
        ClientConfiguration config = new ClientConfiguration().withSocketTimeout(70*1000);
        String swfAccessId = System.getenv("AWS_ACCESS_KEY_ID");
        String swfSecretKey = System.getenv("AWS_SECRET_KEY");
        AWSCredentials awsCredentials = new BasicAWSCredentials(swfAccessId, swfSecretKey);
        AmazonSimpleWorkflow service = new AmazonSimpleWorkflowClient(awsCredentials, config);
        service.setEndpoint("https://swf.us-east-1.amazonaws.com");

        String domain = "helloWorldWalkthrough";
        String taskListToPoll = "HelloWorldAsyncList";

        ActivityWorker aw = new ActivityWorker(service, domain, taskListToPoll);
        aw.addActivitiesImplementation(new GreeterActivitiesImpl());
        aw.start();

        WorkflowWorker wfw = new WorkflowWorker(service, domain, taskListToPoll);
        wfw.addWorkflowImplementationType(GreeterWorkflowImpl.class);
        wfw.start();
    }
}
```

HelloWorldWorkflowAsync implements the workflow starter in `GreeterMain`; it is identical to the HelloWorldWorkflow implementation.

To execute the workflow, run `GreeterWorker` and `GreeterMain`, just as with HelloWorldWorkflow.
HelloWorldWorkflowDistributed Application

With HelloWorldWorkflow and HelloWorldWorkflowAsync, Amazon SWF mediates the interaction between the workflow and activities implementations, but they run locally as a single process. GreeterMain is in a separate process, but it still runs on the same system.

A key feature of Amazon SWF is that it supports distributed applications. For example, you could run the workflow worker on an Amazon EC2 instance, the workflow starter on a data center computer, and the activities on a client desktop computer. You can even run different activities on different systems.

The HelloWorldWorkflowDistributed application extends HelloWorldWorkflowAsync to distribute the application across two systems and three processes.

- The workflow and workflow starter run as separate processes on one system.
- The activities run on a separate system.

To implement the application, create a copy of the helloWorld.HelloWorldWorkflowAsync package in your project directory and name it helloWorld.HelloWorldWorkflowDistributed. The following sections describe how to modify the original HelloWorldWorkflowAsync code to distribute the application across two systems and three processes.

You don't need to change the workflow or activities implementations to run them on separate systems, not even the version numbers. You also don't need to modify GreeterMain. All you need to change is the activities and workflow host.

With HelloWorldWorkflowAsync, a single application serves as the workflow and activity host. To run the workflow and activity implementations on separate systems, you must implement separate applications. Delete GreeterWorker from the project and add two new class files, GreeterWorkflowWorker and GreeterActivitiesWorker.

HelloWorldWorkflowDistributed implements its activities host in GreeterActivitiesWorker, as follows:

```java
class GreeterActivitiesWorker {
    public static void main(String[] args) throws Exception {
        ClientConfiguration config = new ClientConfiguration().withSocketTimeout(70*1000);
        String swfAccessId = System.getenv("AWS_ACCESS_KEY_ID");
        String swfSecretKey = System.getenv("AWS_SECRET_KEY");
        AWSCredentials awsCredentials = new BasicAWSCredentials(swfAccessId, swfSecretKey);
        AmazonSimpleWorkflow service = new AmazonSimpleWorkflowClient(awsCredentials, config);
        service.setEndpoint("https://swf.us-east-1.amazonaws.com");
        String domain = "helloWorldExamples";
        String taskListToPoll = "HelloWorldAsyncList";
        ActivityWorker aw = new ActivityWorker(service, domain, taskListToPoll);
        aw.addActivitiesImplementation(new GreeterActivitiesImpl());
        aw.start();
    }
}
```

API Version 2012-01-25
HelloWorldWorkflowDistributed implements its workflow host in GreeterWorkflowWorker, as follows:

```java
import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient;
import com.amazonaws.services.simpleworkflow.flow.WorkflowWorker;

public class GreeterWorkflowWorker {
    public static void main(String[] args) throws Exception {
        ClientConfiguration config = new ClientConfiguration().withSocketTimeout(70*1000);

        String swfAccessId = System.getenv("AWS_ACCESS_KEY_ID");
        String swfSecretKey = System.getenv("AWS_SECRET_KEY");
        AWSCredentials awsCredentials = new BasicAWSCredentials(swfAccessId, swfSecretKey);

        AmazonSimpleWorkflow service = new AmazonSimpleWorkflowClient(awsCredentials, config);
        service.setEndpoint("https://swf.us-east-1.amazonaws.com");

        String domain = "helloWorldExamples";
        String taskListToPoll = "HelloWorldAsyncList";

        WorkflowWorker wfw = new WorkflowWorker(service, domain, taskListToPoll);
        wfw.addWorkflowImplementationType(GreeterWorkflowImpl.class);
        wfw.start();
    }
}
```

Note that GreeterActivitiesWorker is just GreeterWorker without the WorkflowWorker code and GreeterWorkflowWorker is just GreeterWorker without the ActivityWorker code.

To run the workflow:
1. Create a runnable JAR file with GreeterActivitiesWorker as the entry point.
2. Copy the JAR file from Step 1 to another system, which can be running any operating system that supports Java.
3. Ensure that AWS credentials with access to the same Amazon SWF domain are made available on the other system.
4. Run the JAR file.
5. On your development system, use Eclipse to run GreeterWorkflowWorker and GreeterMain.

Other than the fact that the activities are running on a different system than the workflow worker and workflow starter, the workflow works in exactly the same way as HelloWorldAsync. However, because println call that prints "Hello World!" to the console is in the say activity, the output will appear on the system that is running the activities worker.
With HelloWorldWorkflowParallel, getName and getGreeting run in parallel and each return part of the greeting. say then merges the two strings into a greeting, and prints it to the console.

To implement the application, create a copy of the helloWorld.HelloWorldWorkflow package in your project directory and name it helloWorld.HelloWorldWorkflowParallel. The following sections describe how to modify the original HelloWorldWorkflow code to run getName and getGreeting in parallel.

### HelloWorldWorkflowParallel Activities Worker

The HelloWorldWorkflowParallel activities interface is implemented in GreeterActivities, as shown in the following example.

```java
import com.amazonaws.services.simpleworkflow.flow.annotations.Activities;
import com.amazonaws.services.simpleworkflow.flow.annotations.ActivityRegistrationOptions;

@Activities(version="5.0")
@ActivityRegistrationOptions(defaultTaskScheduleToStartTimeoutSeconds = 300,
                               defaultTaskStartToCloseTimeoutSeconds = 10)
public interface GreeterActivities {
    public String getName();
    public String getGreeting();
    public void say(String greeting, String name);
}
```

The interface is similar to HelloWorldWorkflow, with the following exceptions:

- `getGreeting` doesn't take any input; it simply returns a greeting string.
- `say` takes two input strings, the greeting and the name.
- The interface has a new version number, which is required any time that you change a registered interface.

HelloWorldWorkflowParallel implements the activities in GreeterActivitiesImpl, as follows:

```java
public class GreeterActivitiesImpl implements GreeterActivities {
    @Override
    public String getName() {
        return "World!";
    }

    @Override
    public String getGreeting() {
        return "Hello ";
    }

    @Override
    public void say(String greeting, String name) {
        System.out.println(greeting + name);
    }
}
```
getName and getGreeting now simply return half of the greeting string. say concatenates the two pieces to produce the complete phrase, and prints it to the console.

HelloWorldWorkflowParallel Workflow Worker

The HelloWorldWorkflowParallel workflow interface is implemented in GreeterWorkflow, as follows:

```java
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 3600)
public interface GreeterWorkflow {
    @Execute(version = "5.0")
    public void greet();
}
```

The class is identical to the HelloWorldWorkflow version, except that the version number has been changed to match the activities worker.

The workflow is implemented in GreeterWorkflowImpl, as follows:

```java
import com.amazonaws.services.simpleworkflow.flow.core.Promise;

public class GreeterWorkflowImpl implements GreeterWorkflow {
    private GreeterActivitiesClient operations = new GreeterActivitiesClientImpl();

    public void greet() {
        Promise<String> name = operations.getName();
        Promise<String> greeting = operations.getGreeting();
        operations.say(greeting, name);
    }
}
```

At a glance, this implementation looks very similar to HelloWorldWorkflow; the three activities client methods execute in sequence. However, the activities don't.

- HelloWorldWorkflow passed name to getGreeting. Because name was a Promise<T> object, getGreeting deferred executing the activity until getName completed, so the two activities executed in sequence.
- HelloWorldWorkflowParallel doesn't pass any input getName or getGreeting. Neither method defers execution and the associated activity methods execute immediately, in parallel.

The say activity takes both greeting and name as input parameters. Because they are Promise<T> objects, say defers execution until both activities complete, and then constructs and prints the greeting.

Notice that HelloWorldWorkflowParallel doesn't use any special modeling code to define the workflow topology. It does it implicitly by using standard Java flow control and taking advantage of the properties of Promise<T> objects. AWS Flow Framework for Java applications can implement even complex topologies simply by using Promise<T> objects in conjunction with conventional Java control flow constructs.
HelloWorldWorkflowParallel Workflow and Activities Host and Starter

HelloWorldWorkflowParallel implements GreeterWorker as the host class for the workflow and activity implementations. It is identical to the HelloWorldWorkflow implementation except for the taskListToPoll name, which is set to "HelloWorldParallelList".

HelloWorldWorkflowParallel implements the workflow starter in GreeterMain, and it is identical to the HelloWorldWorkflow implementation.

To execute the workflow, run GreeterWorker and GreeterMain, just as with HelloWorldWorkflow.
How AWS Flow Framework for Java Works

The AWS Flow Framework for Java works with Amazon SWF to make it easy to create scalable and fault-tolerant applications to perform asynchronous tasks that may be long running, remote, or both. The "Hello World!" examples in What is the AWS Flow Framework for Java? (p. 1) introduced the basics of how to use the AWS Flow Framework to implement basic workflow applications. This section provides conceptual information about how AWS Flow Framework applications work. The first section summarizes the basic structure of an AWS Flow Framework application, and the remaining sections provide further detail about how AWS Flow Framework applications work.

Topics

- AWS Flow Framework Basic Concepts: Application Structure (p. 34)
- AWS Flow Framework Basic Concepts: Reliable Execution (p. 37)
- AWS Flow Framework Basic Concepts: Distributed Execution (p. 38)
- AWS Flow Framework Basic Concepts: Task Lists and Task Execution (p. 40)
- AWS Flow Framework Basic Concepts: Scalable Applications (p. 41)
- AWS Flow Framework Basic Concepts: Data Exchange Between Activities and Workflows (p. 42)
- AWS Flow Framework Basic Concepts: Data Exchange Between Applications and Workflow Executions (p. 43)
- Amazon SWF Timeout Types (p. 44)

AWS Flow Framework Basic Concepts: Application Structure

Conceptually, an AWS Flow Framework application consists of three basic components: workflow starters, workflow workers, and activity workers. One or more host applications are responsible for registering the workers (workflow and activity) with Amazon SWF, starting the workers, and handling cleanup. The workers handle the mechanics of executing the workflow and may be implemented on several hosts.

This diagram represents a basic AWS Flow Framework application:
Implementing these components in three separate applications is convenient conceptually, but you can create applications to implement this functionality in a variety of ways. For example, you can use a single host application for the activity and workflow workers, or use separate activity and workflow hosts. You can also have multiple activity workers, each handling a different set of activities on separate hosts, and so on.

The three AWS Flow Framework components interact indirectly by sending HTTP requests to Amazon SWF, which manages the requests. Amazon SWF does the following:

- Maintains one or more decision task lists, which determine the next step to be performed by a workflow worker.
- Maintains one or more activities task lists, which determine which tasks will be performed by an activity worker.
- Maintains a detailed step-by-step history of the workflow's execution.

With the AWS Flow Framework, your application code doesn't need to deal directly with many of the details shown in the figure, such as sending HTTP requests to Amazon SWF. You simply call AWS Flow Framework methods and the framework handles the details behind the scenes.
Role of the Activity Worker

The activity worker performs the various tasks that the workflow must accomplish. It consists of:

- The activities implementation, which includes a set of activity methods that perform particular tasks for the workflow.
- An ActivityWorker object, which uses HTTP long poll requests to poll Amazon SWF for activity tasks to be performed. When a task is needed, Amazon SWF responds to the request by sending the information required to perform the task. The ActivityWorker object then calls the appropriate activity method, and returns the results to Amazon SWF.

Role of the Workflow Worker

The workflow worker orchestrates the execution of the various activities, manages data flow, and handles failed activities. It consists of:

- The workflow implementation, which includes the activity orchestration logic, handles failed activities, and so on.
- An activities client, which serves as a proxy for the activity worker and enables the workflow worker to schedule activities to be executed asynchronously.
- A WorkflowWorker object, which uses HTTP long poll requests to poll Amazon SWF for decision tasks. If there are tasks on the workflow task list, Amazon SWF responds to the request by returning the information that is required to perform the task. The framework then executes the workflow to perform the task and returns the results to Amazon SWF.

Role of the Workflow Starter

The workflow starter starts a workflow instance, also referred to as a workflow execution, and can interact with an instance during execution in order to pass additional data to the workflow worker or obtain the current workflow state.

The workflow starter uses a workflow client to start the workflow execution, interacts with the workflow as needed during execution, and handles cleanup. The workflow starter could be a locally-run application, a web application, the AWS CLI or even the AWS Management Console.

How Amazon SWF Interacts with Your Application

Amazon SWF mediates the interaction between the workflow components and maintains a detailed workflow history. Amazon SWF doesn't initiate communication with the components; it waits for HTTP requests from the components and manages the requests as required. For example:

- If the request is from a worker, polling for available tasks, Amazon SWF responds directly to the worker if a task is available. For more information about how polling works, see Polling for Tasks in the Amazon Simple Workflow Service Developer Guide.
- If the request is a notification from an activity worker that a task is complete, Amazon SWF records the information in the execution history and adds a task to the decision task list to inform the workflow worker that the task is complete, allowing it to proceed to the next step.
- If the request is from the workflow worker to execute an activity, Amazon SWF records the information in the execution history and adds a task to the activities task list to direct an activity worker to execute the appropriate activity method.
This approach allows workers to run on any system with an Internet connection, including Amazon EC2 instances, corporate data centers, client computers, and so on. They don't even have to be running the same operating system. Because the HTTP requests originate with the workers, there is no need for externally visible ports; workers can run behind a firewall.

For More Information

For a more thorough discussion of how Amazon SWF works, see Amazon Simple Workflow Service Developer Guide.

AWS Flow Framework Basic Concepts: Reliable Execution

Asynchronous distributed applications must deal with reliability issues that are not encountered by conventional applications, including:

- How to provide reliable communication between asynchronous distributed components, such as long-running components on remote systems.
- How to ensure that results are not lost if a component fails or is disconnected, especially for long-running applications.
- How to handle failed distributed components.

Applications can rely on the AWS Flow Framework and Amazon SWF to manage these issues. We'll explore how Amazon SWF provides mechanisms to ensure that your workflows operate reliably and in a predictable way, even when they are long-running and depend on asynchronous tasks carried out computationally and with human interaction.

Providing Reliable Communication

AWS Flow Framework provides reliable communication between a workflow worker and its activities workers by using Amazon SWF to dispatch tasks to distributed activities workers and return the results to the workflow worker. Amazon SWF uses the following methods to ensure reliable communication between a worker and its activities:

- Amazon SWF durably stores scheduled activity and workflow tasks and guarantees that they will be performed at most once.
- Amazon SWF guarantees that an activity task will either complete successfully and return a valid result or it will notify the workflow worker that the task failed.
- Amazon SWF durably stores each completed activity's result or, for failed activities, it stores relevant error information.

The AWS Flow Framework then uses the activity results from Amazon SWF to determine how to proceed with the workflow's execution.

Ensuring that Results are Not Lost

Maintaining Workflow History

An activity that performs a data-mining operation on a petabyte of data might take hours to complete, and an activity that directs a human worker to perform a complex task might take days, or even weeks to complete!
To accommodate scenarios such as these, AWS Flow Framework workflows and activities can take arbitrarily long to complete: *up to a limit of one year for a workflow execution*. Reliably executing long running processes requires a mechanism to durably store the workflow's execution history on an ongoing basis.

The AWS Flow Framework handles this by depending on Amazon SWF, which maintains a running history of each workflow instance. The workflow's history provides a complete and authoritative record of the workflow's progress, including all the workflow and activity tasks that have been scheduled and completed, and the information returned by completed or failed activities.

AWS Flow Framework applications usually don't need to interact with the workflow history directly, although they can access it if necessary. For most purposes, applications can simply let the framework interact with the workflow history behind the scenes. For a full discussion of workflow history, see Workflow History in the Amazon Simple Workflow Service Developer Guide.

Stateless Execution

The execution history allows workflow workers to be *stateless*. If you have multiple instances of a workflow or activity worker, any worker can perform any task. The worker receives all the state information that it needs to perform the task from Amazon SWF.

This approach makes workflows more reliable. For example, if an activity worker fails, you don't have to restart the workflow. Just restart the worker and it will start polling the task list and processing whatever tasks are on the list, regardless of when the failure occurred. You can make your overall workflow fault-tolerant by using two or more workflow and activity workers, perhaps on separate systems. Then, if one of the workers fails, the others will continue to handle scheduled tasks without any interruption in workflow progress.

Handling Failed Distributed Components

Activities often fail for ephemeral reasons, such as a brief disconnection, so a common strategy for handling failed activities is to retry the activity. Instead of handling the retry process by implementing complex message passing strategies, applications can depend on the AWS Flow Framework. It provides several mechanisms for retrying failed activities, and provides a built-in exception-handling mechanism that works with asynchronous, distributed execution of tasks in a workflow.

AWS Flow Framework Basic Concepts: Distributed Execution

A *workflow instance* is essentially a virtual thread of execution that can span activities and orchestration logic running on multiple remote computers. Amazon SWF and the AWS Flow Framework function as an operating system that manages workflow instances on a virtual CPU by:

- Maintaining each instance's execution state.
- Switching between instances.
- Resuming execution of an instance at the point that it was switched out.

Replaying Workflows

Because activities can be long-running, it's undesirable to have the workflow simply block until it completes. Instead, the AWS Flow Framework manages workflow execution by using a *replay* mechanism, which relies on the workflow history maintained by Amazon SWF to execute the workflow in episodes.
Each episode replays the workflow logic in a way that executes each activity only once, and ensures that activities and asynchronous methods don’t execute until their Promise (p. 42) objects are ready.

The workflow starter initiates the first replay episode when it starts the workflow execution. The framework calls the workflow’s entry point method and:

1. Executes all workflow tasks that don’t depend on activity completion, including calling all activity client methods.
2. Gives Amazon SWF a list of activities tasks to be scheduled for execution. For the first episode, this list consists of only those activities that don’t depend on a Promise and can be executed immediately.
3. Notifies Amazon SWF that the episode is complete.

Amazon SWF stores the activity tasks in the workflow history and schedules them for execution by placing them on the activity task list. The activity workers poll the task list and execute the tasks.

When an activity worker completes a task, it returns the result to Amazon SWF, which records it in the workflow execution history and schedules a new workflow task for the workflow worker by placing it on the workflow task list. The workflow worker polls the task list and when it receives the task, it runs the next replay episode, as follows:

1. The framework runs the workflow’s entry point method again and:
   - Executes all workflow tasks that don’t depend on activity completion, including calling all activity client methods. However, the framework checks the execution history and doesn’t schedule duplicate activity tasks.
   - Checks the history to see which activity tasks have completed and executes any asynchronous workflow methods that depend on those activities.
2. When all workflow tasks that can be executed have completed, the framework reports back to Amazon SWF:
   - It gives Amazon SWF a list of any activities whose input Promise<T> objects have become ready since the last episode and can be scheduled for execution.
   - If the episode generated no additional activity tasks but there are still uncompleted activities, the framework notifies Amazon SWF that the episode is complete. It then waits for another activity to complete, initiating the next replay episode.
   - If the episode generated no additional activity tasks and all activities have completed, the framework notifies Amazon SWF that the workflow execution is complete.

For examples of replay behavior, see AWS Flow Framework for Java Replay Behavior (p. 116).

Replay and Asynchronous Workflow Methods

Asynchronous workflow methods are often used much like activities, because the method defers execution until all input Promise<T> objects are ready. However, the replay mechanism handles asynchronous methods differently than activities.

- Replay doesn’t guarantee that an asynchronous method will execute only once. It defers execution on an asynchronous method until its input Promise objects are ready, but it then executes that method for all subsequent episodes.
- When an asynchronous method completes, it doesn’t start a new episode.

An example of replaying an asynchronous workflow is provided in AWS Flow Framework for Java Replay Behavior (p. 116).
Replay and Workflow Implementation

For the most part, you don't need to be concerned with the details of the replay mechanism. It is basically something that happens behind the scenes. However, replay has two important implications for your workflow implementation.

- Do not use workflow methods to perform long-running tasks, because replay will repeat that task multiple times. Even asynchronous workflow methods typically run more than once. Instead, use activities for long running tasks; replay executes activities only once.
- Your workflow logic must be completely deterministic; every episode must take the same control flow path. For example, the control flow path should not depend on the current time. For a detailed description of replay and the determinism requirement, see Nondeterminism (p. 122).

AWS Flow Framework Basic Concepts: Task Lists and Task Execution

Amazon SWF manages workflow and activity tasks by posting them to named lists. Amazon SWF maintains at least two task lists, one for workflow workers and one for activity workers.

**Note**

You can specify as many task lists as you need, with different workers assigned to each list. There is no limit to the number of task lists. You typically specify a worker's task list in the worker host application when you create the worker object.

The following excerpt from the `HelloWorldWorkflow` host application creates a new activity worker and assigns it to the `HelloWorldList` activities task list.

```java
public class GreeterWorker  {
    public static void main(String[] args) throws Exception {
        ...
        String domain = "helloWorldExamples";
        String taskListToPoll = "HelloWorldList";

        ActivityWorker aw = new ActivityWorker(service, domain, taskListToPoll);
        aw.addActivitiesImplementation(new GreeterActivitiesImpl());
        aw.start();
        ...
    }
}
```

By default, Amazon SWF schedules the worker's tasks on the `HelloWorldList` list. Then the worker polls that list for tasks. You can assign any name to a task list. You can even use the same name for both workflow and activity lists. Internally, Amazon SWF puts workflow and activity task list names in different namespaces, so the two lists will be distinct.

If you don't specify a task list, the AWS Flow Framework specifies a default list when the worker registers the type with Amazon SWF. For more information, see Workflow and Activity Type Registration (p. 59).

Sometimes it's useful to have a specific worker or group of workers perform certain tasks. For example, an image processing workflow might use one activity to download an image and another activity to process the image. It's more efficient to perform both tasks on the same system, and avoid the overhead of transferring large files over the network.

To support such scenarios, you can explicitly specify a task list when you call an activity client method by using an overload that includes a `schedulingOptions` parameter. You specify the task list by passing the method an appropriately configured `ActivitySchedulingOptions` object.
For example, suppose that the say activity of the HelloWorldWorkflow application is hosted by an activity worker different from getName and getGreeting. The following example shows how to ensure that say uses the same task list as getName and getGreeting, even if they were originally assigned to different lists.

```java
public class GreeterWorkflowImpl implements GreeterWorkflow {
    private GreeterActivitiesClient operations1 = new GreeterActivitiesClientImpl1(); // getName and getGreeting
    private GreeterActivitiesClient operations2 = new GreeterActivitiesClientImpl2(); // say

    @Override
    public void greet() {
        Promise<String> name = operations1.getName();
        Promise<String> greeting = operations1.getGreeting(name);
        runSay(greeting);
    }

    @Asynchronous
    private void runSay(Promise<String> greeting){
        String taskList = operations1.getSchedulingOptions().getTaskList();
        ActivitySchedulingOptions schedulingOptions = new ActivitySchedulingOptions();
        schedulingOptions.setTaskList(taskList);
        operations2.say(greeting, schedulingOptions);
    }
}
```

The asynchronous runSay method gets the getGreeting task list from its client object. Then it creates and configures an ActivitySchedulingOptions object that ensures that say polls the same task list as getGreeting.

**Note**

When you pass a schedulingOptions parameter to an activity client method, it overrides the original task list only for that activity execution. If you call the activities client method again without specifying a task list, Amazon SWF assigns the task to the original list, and the activity worker will poll that list.

### AWS Flow Framework Basic Concepts: Scalable Applications

Amazon SWF has two key features that make it easy to scale a workflow application to handle the current load:

- A complete workflow execution history, which allows you to implement a stateless application.
- Task scheduling that is loosely coupled to task execution, which makes it easy to scale your application to meet current demands.

Amazon SWF schedules tasks by posting them to dynamically allocated task lists, not by communicating directly with workflow and activity workers. Instead, the workers use HTTP requests to poll their respective lists for tasks. This approach loosely couples task scheduling to task execution and allows workers to run on any suitable system, including Amazon EC2 instances, corporate data centers, client computers, and so on. Since the HTTP requests originate with the workers, there is no need for externally visible ports, which enables workers to even run behind a firewall.

The long-polling mechanism that workers use to poll for tasks ensures that workers don't get overloaded. Even if there is a spike in scheduled tasks, workers pull tasks at their own pace. However, because workers are stateless, you can dynamically scale an application to meet increased load by starting additional worker instances. Even if they are running on different systems, each instance polls
the same task list and the first available worker instance executes each task, regardless of where the worker is located or when it started. When the load declines, you can reduce the number of workers accordingly.

AWS Flow Framework Basic Concepts: Data Exchange Between Activities and Workflows

When you call an asynchronous activity client method, it immediately returns a Promise (also known as a Future) object, which represents the activity method's return value. Initially, the Promise is in an unready state and the return value is undefined. After the activity method completes its task and returns, the framework marshals the return value across the network to the workflow worker, which assigns a value to the Promise and puts the object in a ready state.

Even if an activity method has no return value, you can still use the Promise for managing workflow execution. If you pass a returned Promise to an activity client method or an asynchronous workflow method, it defers execution until the object is ready.

If you pass one or more Promises to an activity client method, the framework queues the task but defers scheduling it until all the objects are ready. It then extracts the data from each Promise and marshals it across the internet to the activity worker, which passes it to the activity method as a standard type.

Note
If you need to transfer large amounts of data between workflow and activity workers, the preferred approach is to store the data in a convenient location and just pass the retrieval information. For example, you can store the data in an Amazon S3 bucket and pass the associated URL.

The Promise<T> Type

The Promise<T> type is similar in some ways to the Java Future<T> type. Both types represent values returned by asynchronous methods and are initially undefined. You access an object's value by calling its get method. Beyond that, the two types behave quite differently.

- Future<T> is a synchronization construct that allows an application to wait on an asynchronous method's completion. If you call get and the object isn't ready, it blocks until the object is ready.
- With Promise<T>, synchronization is handled by the framework. If you call get and the object isn't ready, get throws an exception.

The primary purpose of Promise<T> is to manage data flow from one activity to another. It ensures that an activity doesn't execute until the input data is valid. In many cases, workflow workers don't need to access Promise<T> objects directly; they simply pass the objects from one activity to another and let the framework and the activity workers handle the details. To access a Promise<T> object's value in a workflow worker, you must be certain that the object is ready before calling its get method.

- The preferred approach is to pass the Promise<T> object to an asynchronous workflow method and process the values there. An asynchronous method defers execution until all of its input Promise<T> objects are ready, which guarantees that you can safely access their values.
- Promise<T> exposes an isReady method that returns true if the object is ready. Using isReady to poll a Promise<T> object isn't recommended, but isReady is useful in certain circumstances. For an example, see AWS Flow Framework Recipes.

The AWS Flow Framework for Java also includes a Settable<T> type, which is derived from Promise<T> and has similar behavior. The difference is that the framework usually sets the value of a
Promise<T> object and the workflow worker is responsible for setting the value of a Settable<T> For an example, see AWS Flow Framework Recipes

There are some circumstance where a workflow worker needs to create a Promise<T> object and set its value. For example, an asynchronous method that returns a Promise<T> object needs to create a return value.

- To create an object that represents a typed value, call the static Promise.asPromise method, which creates a Promise<T> object of the appropriate type, sets its value, and puts it in the ready state.
- To create a Promise<Void> object, call the static Promise.Void method.

Note
Promise<T> can represent any valid type. However, if the data must be marshaled across the internet, the type must be compatible with the data converter. See the next section for details.

Data Converters and Marshaling

The AWS Flow Framework marshals data across the internet by using a data converter. By default, the framework uses a data converter that is based on the Jackson JSON processor. However, this converter has some limitations. For example, it can't marshal maps that don't use strings as keys. If the default converter isn't sufficient for your application, you can implement a custom data converter. For details, see DataConverters (p. 89).

AWS Flow Framework Basic Concepts: Data Exchange Between Applications and Workflow Executions

A workflow entry point method can have one or more parameters, which allows the workflow starter to pass initial data to the workflow. It can also useful to provide additional data to the workflow during execution. For example, if a customer changes their shipping address, you could notify the order-processing workflow so that it can make appropriate changes.

Amazon SWF allows workflows to implement a signal method, which allows applications such as the workflow starter to pass data to the workflow at any time. A signal method can have any convenient name and parameters. You designate it as a signal method by including it in your workflow interface definition, and applying a @Signal annotation to the method declaration.

The following example shows an order processing workflow interface that declares a signal method, changeOrder, which allows the workflow starter to change the original order after the workflow has started.

```java
@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 300)
public interface WaitForSignalWorkflow {
    @Execute(version = "1.0")
    public void placeOrder(int amount);
    @Signal
    public void changeOrder(int amount);
}
```

The framework's annotation processor creates a workflow client method with the same name as the signal method and the workflow starter calls the client method to pass data to the workflow. For an example, see AWS Flow Framework Recipes
Amazon SWF Timeout Types

To ensure that workflow executions run correctly, Amazon SWF enables you to set different types of timeouts. Some timeouts specify how long the workflow can run in its entirety. Other timeouts specify how long activity tasks can take before being assigned to a worker and how long they can take to complete from the time they are scheduled. All timeouts in the Amazon SWF API are specified in seconds. Amazon SWF also supports the string `NONE` as a timeout value, which indicates no timeout.

For timeouts related to decision tasks and activity tasks, Amazon SWF adds an event to the workflow execution history. The attributes of the event provide information about what type of timeout occurred and which decision task or activity task was affected. Amazon SWF also schedules a decision task. When the decider receives the new decision task, it will see the timeout event in the history and take an appropriate action by calling the `RespondDecisionTaskCompleted` action.

A task is considered open from the time that it is scheduled until it is closed. Therefore a task is reported as open while a worker is processing it. A task is closed when a worker reports it as completed, canceled, or failed. A task may also be closed by Amazon SWF as the result of a timeout.

Timeouts in Workflow and Decision Tasks

The following diagram shows how workflow and decision timeouts are related to the lifetime of a workflow:

![Diagram showing workflow and decision timeouts](image)

There are two timeout types that are relevant to workflow and decision tasks:

- **Workflow Start to Close (timeoutType: START_TO_CLOSE)** – This timeout specifies the maximum time that a workflow execution can take to complete. It is set as a default during workflow registration, but it can be overridden with a different value when the workflow is started. If this timeout is exceeded, Amazon SWF closes the workflow execution and adds an event of type `WorkflowExecutionTimedOut` to the workflow execution history. In addition to the timeoutType, the event attributes specify the childPolicy that is in effect for this workflow execution. The child policy specifies how child workflow executions are handled if the parent workflow execution times out or otherwise terminates. For example, if the childPolicy is set to TERMINATE, then child workflow executions will be terminated. Once a workflow execution has timed out, you can't take any action on it other than visibility calls.

- **Decision Task Start to Close (timeoutType: START_TO_CLOSE)** – This timeout specifies the maximum time that the corresponding decider can take to complete a decision task. It is set during workflow type registration. If this timeout is exceeded, the task is marked as timed out in the workflow execution history, and Amazon SWF adds an event of type `DecisionTaskTimedOut` to the workflow execution history.
Timeouts in Activity Tasks

The following diagram shows how timeouts are related to the lifetime of an activity task:

There are four timeout types that are relevant to activity tasks:

- **Activity Task Start to Close** (timeoutType: START_TO_CLOSE) – This timeout specifies the maximum time that an activity worker can take to process a task after the worker has received the task. Attempts to close a timed out activity task using `RespondActivityTaskCanceled`, `RespondActivityTaskCompleted`, and `RespondActivityTaskFailed` will fail.

- **Activity Task Heartbeat** (timeoutType: HEARTBEAT) – This timeout specifies the maximum time that a task can run before providing its progress through the `RecordActivityTaskHeartbeat` action.

- **Activity Task Schedule to Start** (timeoutType: SCHEDULE_TO_START) – This timeout specifies how long Amazon SWF waits before timing out the activity task if no workers are available to perform the task. Once timed out, the expired task will not be assigned to another worker.

- **Activity Task Schedule to Close** (timeoutType: SCHEDULE_TO_CLOSE) – This timeout specifies how long the task can take from the time it is scheduled to the time it is complete. As a best practice, this value should not be greater than the sum of the task schedule-to-start timeout and the task start-to-close timeout.

**Note**
Each of the timeout types has a default value, which is generally set to NONE (infinite). The maximum time for any activity execution is limited to one year, however.

You set default values for these during activity type registration, but you can override them with new values when you schedule the activity task. When one of these timeouts occurs, Amazon SWF will add...
an event of type ActivityTaskTimedOut to the workflow history. The timeoutType value attribute of this event will specify which of these timeouts occurred. For each of the timeouts, the value of timeoutType is shown in parentheses. The event attributes will also include the IDs for the events that correspond to when the activity task was scheduled (scheduledEventId) and when it was started (startedEventId). In addition to adding the event, Amazon SWF also schedules a new decision task to alert the decider that the timeout occurred.
Best Practices

Use these best practices to make the most of the AWS Flow Framework for Java.

Topics

- Making Changes to Decider Code: Versioning and Feature Flags (p. 47)

Making Changes to Decider Code: Versioning and Feature Flags

This section shows how to avoid backwards-incompatible changes to a decider using two methods:

- Versioning (p. 52) provides a basic solution.
- Versioning with Feature Flags (p. 54) builds on the Versioning solution: No new version of the workflow is introduced, and there is no need to push new code to update the version.

Before you try these solutions, familiarize yourself with the Example Scenario (p. 47) section which explains the causes and effects of backwards-incompatible decider changes.

The Replay Process and Code Changes

When an AWS Flow Framework for Java decider worker executes a decision task, it first must rebuild the current state of the execution before it can add steps to it. The decider does this using a process called replay.

The replay process re-executes the decider code from the beginning, while simultaneously going through the history of events that have already occurred. Going through the event history allows the framework to react to signals or task completion and unblock Promise objects in the code.

When the framework executes the decider code, it assigns an ID to each scheduled task (an activity, Lambda function, timer, child workflow, or outgoing signal) by incrementing a counter. The framework communicates this ID to Amazon SWF, and adds the ID to history events, such as ActivityTaskCompleted.

For the replay process to succeed, it is important for the decider code to be deterministic, and to schedule the same tasks in the same order for every decision in every workflow execution. If you don't adhere to this requirement, the framework might, for example, fail to match the ID in an ActivityTaskCompleted event to an existing Promise object.

Example Scenario

There is a class of code changes considered to be backwards-incompatible. These changes include updates that modify the number, type, or order of the scheduled tasks. Consider the following example:

You write decider code to schedule two timer tasks. You start an execution and run a decision. As a result, two timer tasks are scheduled, with IDs 1 and 2.

If you update the decider code to schedule only one timer before the next decision to be executed, during the next decision task the framework will fail to replay the second TimerFired event, because ID 2 doesn't match any timer tasks that the code has produced.
Scenario Outline

The following outline shows the steps of this scenario. The final goal of the scenario is to migrate to a system that schedules only one timer but doesn’t cause failures in executions started before the migration.

1. The Initial Decider Version
   1. Write the decider.
   2. Start the decider.
   3. The decider schedules two timers.
   4. The decider starts five executions.
   5. Stop the decider.

2. A Backwards-Incompatible Decider Change
   1. Modify the decider.
   2. Start the decider.
   3. The decider schedules one timer.
   4. The decider starts five executions.

The following sections include examples of Java code that show how to implement this scenario. The code examples in the Solutions (p. 52) section show various ways to fix backwards-incompatible changes.

**Note**
You can use the latest version of the AWS SDK for Java to run this code.

Common Code

The following Java code doesn't change between the examples in this scenario.

SampleBase.java

```java
package sample;
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
import com.amazonaws.services.simpleworkflow.flow.JsonDataConverter;
import com.amazonaws.services.simpleworkflow.model.DescribeWorkflowExecutionRequest;
import com.amazonaws.services.simpleworkflow.model.DomainAlreadyExistsException;
import com.amazonaws.services.simpleworkflow.model.Run;
import com.amazonaws.services.simpleworkflow.model.StartWorkflowExecutionRequest;
import com.amazonaws.services.simpleworkflow.model.TaskList;
import com.amazonaws.services.simpleworkflow.model.WorkflowExecution;
import com.amazonaws.services.simpleworkflow.model.WorkflowExecutionDetail;
import com.amazonaws.services.simpleworkflow.model.WorkflowType;
public class SampleBase {
    protected String domain = "DeciderChangeSample";
    protected String taskList = "DeciderChangeSample-" + UUID.randomUUID().toString();
    protected AmazonSimpleWorkflow service = AmazonSimpleWorkflowClientBuilder.defaultClient();
}
```
Example Scenario

```java
{ try {
    AmazonSimpleWorkflowClientBuilder.defaultClient().registerDomain(new RegisterDomainRequest().withName(domain).withDescription("desc").withWorkflowExecutionRetentionPeriodInDays("7"));
} catch (DomainAlreadyExistsException e) {
}
}

protected List<WorkflowExecution> workflowExecutions = new ArrayList<>();

protected void startFiveExecutions(String workflow, String version, Object input) {
    for (int i = 0; i < 5; i++) {
        String id = UUID.randomUUID().toString();
        Run startWorkflowExecution = service.startWorkflowExecution(
            new StartWorkflowExecutionRequest().withDomain(domain).withTaskList(new TaskList().withName(taskList)).withInput(new JsonDataConverter().toData(new Object[ ] { input })).withWorkflowId(id).withWorkflowType(new WorkflowType().withName(workflow).withVersion(version)));
        workflowExecutions.add(new WorkflowExecution().withWorkflowId(id).withRunId(startWorkflowExecution.getRunId()));
        sleep(1000);
    }
}

protected void printExecutionResults() {
    waitForExecutionsToClose();
    System.out.println("Results:");
    for (WorkflowExecution wid : workflowExecutions) {
        WorkflowExecutionDetail details = service.describeWorkflowExecution(new DescribeWorkflowExecutionRequest().withDomain(domain).withExecution(wid));
        System.out.println(wid.getWorkflowId() + " " + details.getExecutionInfo().getCloseStatus());
    }
}

protected void waitForExecutionsToClose() {
    loop: while (true) {
        for (WorkflowExecution wid : workflowExecutions) {
            WorkflowExecutionDetail details = service.describeWorkflowExecution(new DescribeWorkflowExecutionRequest().withDomain(domain).withExecution(wid));
            if ("OPEN".equals(details.getExecutionInfo().getExecutionStatus())) {
                sleep(1000);
                continue loop;
            }
        }
        return;
    }
}

protected void sleep(int millis) {
    try {
        Thread.sleep(millis);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}
}

Input.java

package sample;

public class Input {
```
private Boolean skipSecondTimer;

public Input() {
    this.skipSecondTimer = null;
}

public Input(Boolean skipSecondTimer) {
    this.skipSecondTimer = skipSecondTimer;
}

public Boolean getSkipSecondTimer() {
    return skipSecondTimer != null && skipSecondTimer;
}

public Input setSkipSecondTimer(Boolean skipSecondTimer) {
    this.skipSecondTimer = skipSecondTimer;
    return this;
}

Writing Initial Decider Code

The following is the initial Java code of the decider. It's registered as version 1 and it schedules two five-second timer tasks.

InitialDecider.java

```java
package sample.v1;

import com.amazonaws.services.simpleworkflow.flow.DecisionContext;
import com.amazonaws.services.simpleworkflow.flow.DecisionContextProviderImpl;
import com.amazonaws.services.simpleworkflow.flow.WorkflowClock;
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;
import sample.Input;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
defaultTaskStartToCloseTimeoutSeconds = 5)
public interface Foo {
    @Execute(version = "1")
    public void sample(Input input);
}

public static class Impl implements Foo {
    private DecisionContext decisionContext = new DecisionContextProviderImpl().getDecisionContext();
    private WorkflowClock clock = decisionContext.getWorkflowClock();

    @Override
    public void sample(Input input) {
        System.out.println("Decision (V1) WorkflowId: ");
        decisionContext.getWorkflowContext().getWorkflowExecution().getWorkflowId()
        clock.createTimer(5);
        clock.createTimer(5);
    }
}
```
Simulating a Backwards-Incompatible Change

The following modified Java code of the decider is a good example of a backwards-incompatible change. The code is still registered as version 1, but schedules only one timer.

**ModifiedDecider.java**

```java
package sample.v1.modified;

import com.amazonaws.services.simpleworkflow.flow.DecisionContext;
import com.amazonaws.services.simpleworkflow.flow.DecisionContextProviderImpl;
import com.amazonaws.services.simpleworkflow.flow.WorkflowClock;
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;
import sample.Input;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
defaultTaskStartToCloseTimeoutSeconds = 5)
public interface Foo {
    @Execute(version = "1")
    public void sample(Input input);

    public static class Impl implements Foo {
        private DecisionContext decisionContext = new DecisionContextProviderImpl().getDecisionContext();
        private WorkflowClock clock = decisionContext.getWorkflowClock();

        @Override
        public void sample(Input input) {
            System.out.println("Decision (V1 modified) WorkflowId: " +
            decisionContext.getWorkflowContext().getWorkflowExecution().getWorkflowId());
            clock.createTimer(5);}
    }
}
```

The following Java code allows you to simulate the problem of making backwards-incompatible changes by running the modified decider.

**RunModifiedDecider.java**

```java
package sample;

import com.amazonaws.services.simpleworkflow.flow.WorkflowWorker;

public class BadChange extends SampleBase {
    public static void main(String[] args) throws Exception {
        new BadChange().run();
    }

    public void run() throws Exception {
        // Start the first version of the decider
        WorkflowWorker before = new WorkflowWorker(service, domain, taskList);
        before.addWorkflowImplementationType(sample.v1.Foo.Impl.class);
        before.start();
    }
}
```
// Start a few executions
startFiveExecutions("Foo.sample", "1", new Input());

// Stop the first decider worker and wait a few seconds
// for its pending pollers to match and return
before.suspendPolling();
sleep(2000);

// At this point, three executions are still open, with more decisions to make

// Start the modified version of the decider
WorkflowWorker after = new WorkflowWorker(service, domain, taskList);
after.addWorkflowImplementationType(sample.v1.modified.Foo.Impl.class);
after.start();

// Start a few more executions
startFiveExecutions("Foo.sample", "1", new Input());

printExecutionResults();
}

When you run the program, the three executions that fail are those that started under the initial version of the decider and continued after the migration.

Solutions

You can use the following solutions to avoid backwards-incompatible changes. For more information, see Making Changes to Decider Code (p. 47) and Example Scenario (p. 47).

Using Versioning

In this solution, you copy the decider to a new class, modify the decider, and then register the decider under a new workflow version.

VersionedDecider.java

```java
package sample.v2;

import com.amazonaws.services.simpleworkflow.flow.DecisionContext;
import com.amazonaws.services.simpleworkflow.flow.DecisionContextProviderImpl;
import com.amazonaws.services.simpleworkflow.flow.WorkflowClock;
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;
import sample.Input;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
    defaultTaskStartToCloseTimeoutSeconds = 5)
public interface Foo {
    @Execute(version = "2")
    public void sample(Input input);

    public static class Impl implements Foo {
        private DecisionContext decisionContext = new
            DecisionContextProviderImpl().getDecisionContext();
    }
}
```
private WorkflowClock clock = decisionContext.getWorkflowClock();

@Override
public void sample(Input input) {
    System.out.println("Decision (V2) WorkflowId: " +
    decisionContext.getWorkflowContext().getWorkflowExecution().getWorkflowId());
    clock.createTimer(5);
}

In the updated Java code, the second decider worker runs both versions of the workflow, allowing in-flight executions to continue to execute independently of the changes in version 2.

```
package sample;
import com.amazonaws.services.simpleworkflow.flow.WorkflowWorker;
public class VersionedChange extends SampleBase {
    public static void main(String[] args) throws Exception {
        new VersionedChange().run();
    }

    public void run() throws Exception {
        // Start the first version of the decider, with workflow version 1
        WorkflowWorker before = new WorkflowWorker(service, domain, taskList);
        before.addWorkflowImplementationType(sample.v1.Foo.Impl.class);
        before.start();

        // Start a few executions with version 1
        startFiveExecutions("Foo.sample", "1", new Input());

        // Stop the first decider worker and wait a few seconds
        // for its pending pollers to match and return
        before.suspendPolling();
        sleep(2000);

        // At this point, three executions are still open, with more decisions to make

        // Start a worker with both the previous version of the decider (workflow version 1)
        // and the modified code (workflow version 2)
        WorkflowWorker after = new WorkflowWorker(service, domain, taskList);
        after.addWorkflowImplementationType(sample.v1.Foo.Impl.class);
        after.addWorkflowImplementationType(sample.v2.Foo.Impl.class);
        after.start();

        // Start a few more executions with version 2
        startFiveExecutions("Foo.sample", "2", new Input());

        printExecutionResults();
    }
}
```

When you run the program, all executions complete successfully.
Using Feature Flags

Another solution to backwards-compatibility issues is to branch code to support two implementations in the same class is to branch based on input data instead of workflow versions.

When you take this approach, you add fields to (or modify existing fields of) your input objects every time you introduce sensitive changes. For executions that start before the migration, the input object won’t have the field (or will have a different value). Thus, you don’t have to increase the version number.

**Note**

If you add new fields, ensure that the JSON deserialization process is backwards-compatible. Objects serialized before the introduction of the field should still successfully deserialize after the migration. Because JSON sets a null value whenever a field is missing, always use boxed types (Boolean instead of boolean) and handle the cases where the value is null.

### FeatureFlagDecider.java

```java
package sample.v1.featureflag;
import com.amazonaws.services.simpleworkflow.flow.DecisionContext;
import com.amazonaws.services.simpleworkflow.flow.DecisionContextProviderImpl;
import com.amazonaws.services.simpleworkflow.flow.WorkflowClock;
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;
import sample.Input;
@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
defaultTaskStartToCloseTimeoutSeconds = 5)
public interface Foo {
    @Execute(version = "1")
    public void sample(Input input);
    public static class Impl implements Foo {
        private DecisionContext decisionContext = new
DecisionContextProviderImpl().getDecisionContext();
        private WorkflowClock clock = decisionContext.getWorkflowClock();

        @Override
        public void sample(Input input) {
            System.out.println("Decision (V1 feature flag) WorkflowId: " +
decisionContext.getWorkflowContext().getWorkflowExecution().getWorkflowId());
            clock.createTimer(5);
            if (!input.getSkipSecondTimer()) {
                clock.createTimer(5);
            }
        }
    }
}
```

In the updated Java code, the code for both versions of the workflow is still registered for version 1. However, after the migration, new executions start with the `skipSecondTimer` field of the input data set to true.

### RunFeatureFlagDecider.java

```java
package sample;
```
import com.amazonaws.services.simpleworkflow.flow.WorkflowWorker;

public class FeatureFlagChange extends SampleBase {

    public static void main(String[] args) throws Exception {
        new FeatureFlagChange().run();
    }

    public void run() throws Exception {
        // Start the first version of the decider
        WorkflowWorker before = new WorkflowWorker(service, domain, taskList);
        before.addWorkflowImplementationType(sample.v1.Foo.Impl.class);
        before.start();

        // Start a few executions
        startFiveExecutions("Foo.sample", "1", new Input());

        // Stop the first decider worker and wait a few seconds
        // for its pending pollers to match and return
        before.suspendPolling();
        sleep(2000);

        // At this point, three executions are still open, with more decisions to make

        // Start a new version of the decider that introduces a change
        // while preserving backwards compatibility based on input fields
        WorkflowWorker after = new WorkflowWorker(service, domain, taskList);
        after.addWorkflowImplementationType(sample.v1.featureflag.Foo.Impl.class);
        after.start();

        // Start a few more executions and enable the new feature through the input data
        startFiveExecutions("Foo.sample", "1", new Input().setSkipSecondTimer(true));

        printExecutionResults();
    }
}

When you run the program, all executions complete successfully.
AWS Flow Framework for Java Programming Guide

This section provides details about how to use the features of the AWS Flow Framework for Java to implement workflow applications.

Topics
- Implementing Workflow Applications with the AWS Flow Framework (p. 56)
- Workflow and Activity Contracts (p. 57)
- Workflow and Activity Type Registration (p. 59)
- Activity and Workflow Clients (p. 61)
- Workflow Implementation (p. 71)
- Activity Implementation (p. 74)
- Implementing AWS Lambda Tasks (p. 76)
- Running Programs Written with the AWS Flow Framework for Java (p. 80)
- Execution Context (p. 83)
- Child Workflow Executions (p. 85)
- Continuous Workflows (p. 87)
- Setting Task Priority (p. 88)
- DataConverters (p. 89)
- Passing Data to Asynchronous Methods (p. 90)
- Testability and Dependency Injection (p. 92)
- Error Handling (p. 101)
- Retry Failed Activities (p. 107)
- Daemon Tasks (p. 115)
- AWS Flow Framework for Java Replay Behavior (p. 116)

Implementing Workflow Applications with the AWS Flow Framework

The typical steps involved in developing a workflow with the AWS Flow Framework are:

1. **Define activity and workflow contracts.** Analyze your application's requirements, then determine the required activities and the workflow topology. The activities handle the required processing tasks, while the workflow topology defines the workflow's basic structure and business logic.

   For example, a media processing application might need to download a file, process it, and then upload the processed file to an Amazon Simple Storage Service (S3) bucket. This can be broken down into four activity tasks:

   1. download the file from a server
   2. process the file (for instance, by transcoding it to a different media format)
   3. upload the file to the S3 bucket
   4. perform cleanup by deleting the local files
This workflow would have an entry point method and would implement a simple linear topology that runs the activities in sequence, much like the HelloWorldWorkflow Application (p. 14).

2. **Implement activity and workflow interfaces.** The workflow and activity contracts are defined by Java interfaces, making their calling conventions predictable by SWF, and providing you flexibility when implementing your workflow logic and activity tasks. The various parts of your program can act as consumers of each others' data, yet don't need to be aware of much of the implementation details of any of the other parts.

   For example, you can define a FileProcessingWorkflow interface and provide different workflow implementations for video encoding, compression, thumbnails, and so on. Each of those workflows can have different control flows and can call different activity methods; your workflow starter doesn't need to know. By using interfaces, it is also simple to test your workflows by using mock implementations that can be replaced later with working code.

3. **Generate activity and workflow clients.** The AWS Flow Framework eliminates the need for you to implement the details of managing asynchronous execution, sending HTTP requests, marshaling data, and so forth. Instead, the workflow starter executes a workflow instance by calling a method on the workflow client, and the workflow implementation executes activities by calling methods on the activities client. The framework handles the details of these interactions in the background.

   If you are using Eclipse and you have configured your project, like in Setting up the AWS Flow Framework for Java (p. 3), the AWS Flow Framework annotation processor uses the interface definitions to automatically generate workflow and activities clients that expose the same set of methods as the corresponding interface.

4. **Implement activity and workflow host applications.** Your workflow and activity implementations must be embedded in host applications that poll Amazon SWF for tasks, marshal any data, and call the appropriate implementation methods. AWS Flow Framework for Java includes WorkflowWorker and ActivityWorker classes that make implementing host applications straightforward and easy to do.

5. **Test your workflow.** AWS Flow Framework for Java provides JUnit integration that you can use to test your workflows inline and locally.

6. **Deploy the workers.** You can deploy your workers as appropriate—for example, you can deploy them to Amazon EC2 instances or to computers in your data center. Once deployed and started, the workers start polling Amazon SWF for tasks and handle them as required.

7. **Start executions.** An application starts a workflow instance by using the workflow client to call the workflow’s entry point. You can also start workflows by using the Amazon SWF console. Regardless of how you start a workflow instance, you can use Amazon SWF console to monitor running workflow instance and examine the workflow history for running, completed, and failed instances.

The AWS SDK for Java includes a set of AWS Flow Framework for Java samples that you can browse and run by following the instructions in the readme.html file in the root directory. There are also a set of recipes—simple applications—that show how to handle a variety of specific programming issue, which are available from AWS Flow Framework Recipes.

## Workflow and Activity Contracts

Java interfaces are used to declare the signatures of workflows and activities. The interface forms the contract between the implementation of the workflow (or activity) and the client of that workflow (or activity). For example, a workflow type MyWorkflow is defined using an interface that is annotated with the @Workflow annotation:

```java
@Workflow
```
The contract has no implementation-specific settings. This use of implementation-neutral contracts allows clients to be decoupled from the implementation and hence provides the flexibility to change the implementation details without breaking the client. Conversely, you may also change the client without necessitating changes to the workflow or activity being consumed. For example, the client may be modified to call an activity asynchronously using promises (Promise<T>) without requiring a change to the activity implementation. Similarly, the activity implementation may be changed so that it is completed asynchronously, for example, by a person sending an email—without requiring the clients of the activity to be changed.

In the example above, the workflow interface MyWorkflow contains a method, startMyWF, for starting a new execution. This method is annotated with the @Execute annotation and must have a return type of void or Promise<>. In a given workflow interface, at most one method can be annotated with this annotation. This method is the entry point of the workflow logic, and the framework calls this method to execute the workflow logic when a decision task is received.

The workflow interface also defines the signals that may be sent to the workflow. The signal method gets invoked when a signal with a matching name is received by the workflow execution. For example, the MyWorkflow interface declares a signal method, signal1, annotated with the @Signal annotation.

The @Signal annotation is required on signal methods. The return type of a signal method must be void. A workflow interface may have zero or more signal methods defined in it. You may declare a workflow interface without an @Execute method and some @Signal methods to generate clients that can't start their execution but can send signals to running executions.

Methods annotated with @Execute and @Signal annotations may have any number of parameters of any type other than Promise<T> or its derivatives. This allows you to pass strongly typed inputs to a workflow execution at start and while it is running. The return type of the @Execute method must be void or Promise<>.

Additionally, you may also declare a method in the workflow interface to report the latest state of a workflow execution, for instance, the getState method in the previous example. This state isn't the entire application state of the workflow. The intended use of this feature is to allow you to store up to 32 KB of data to indicate the latest status of the execution. For example, in an order processing workflow, you may store a string that indicates that the order has been received, processed, or canceled. This method is called by the framework every time a decision task is completed to get the latest state. The state is stored in Amazon Simple Workflow Service (Amazon SWF) and can be retrieved using the generated external client. This allows you to check the latest state of a workflow execution. Methods annotated with @GetState must not take any arguments and must not have a void return type.

You can return any type, which fits your needs, from this method. In the above example, an object of MyWorkflowState (see definition below) is returned by the method that is used to store a string state and a numeric percent complete. The method is expected to perform read-only access of the workflow implementation object and is invoked synchronously, which disallows use of any asynchronous operations like calling methods annotated with @Asynchronous. At most one method in a workflow interface can be annotated with @GetState annotation.
public class MyWorkflowState {
    public String status;
    public int percentComplete;
}

Similarly, a set of activities are defined using an interface annotated with @Activities annotation. Each method in the interface corresponds to an activity—for example:

```java
@Activities(version = "1.0")
@ActivityRegistrationOptions(
    defaultTaskScheduleToStartTimeoutSeconds = 300,
    defaultTaskStartToCloseTimeoutSeconds = 3600)
public interface MyActivities {
    // Overrides values from annotation found on the interface
    @ActivityRegistrationOptions(description = "This is a sample activity",
        defaultTaskScheduleToStartTimeoutSeconds = 100,
        defaultTaskStartToCloseTimeoutSeconds = 60)
    int activity1();
    void activity2(int a);
}
```

The interface allows you to group together a set of related activities. You can define any number of activities within an activities interface, and you can define as many activities interfaces as you want. Similar to @Execute and @Signal methods, activity methods can take any number of arguments of any type other than Promise<T> or its derivatives. The return type of an activity must not be Promise<T> or its derivatives.

Workflow and Activity Type Registration

Amazon SWF requires activity and workflow types to be registered before they can be used. The framework automatically registers the workflows and activities in the implementations you add to the worker. The framework looks for types that implement workflows and activities and registers them with Amazon SWF. By default, the framework uses the interface definitions to infer registration options for workflow and activity types. All workflow interfaces are required to have either the @WorkflowRegistrationOptions annotation or the @SkipRegistration annotation. The workflow worker registers all workflow types it is configured with that have the @WorkflowRegistrationOptions annotation. Similarly, each activity method is required to be annotated with either the @ActivityRegistrationOptions annotation or the @SkipRegistration annotation or one of these annotations must be present on the @Activities interface. The activity worker registers all activity types that it is configured with that an @ActivityRegistrationOptions annotation applies to. The registration is performed automatically when you start one of the workers. Workflow and activity types that have the @SkipRegistration annotation are not registered. @ActivityRegistrationOptions, and @SkipRegistration annotations have override semantics and the most specific one is applied to an activity type.

Note that Amazon SWF doesn’t allow you to re-register or modify the type once it has been registered. The framework will try to register all types, but if the type is already registered it will not be re-registered and no error will be reported.

If you need to modify registered settings, you must register a new version of the type. You can also override registered settings when starting a new execution or when calling an activity that uses the generated clients.

The registration requires a type name and some other registration options. The default implementation determines these as follows:
**Workflow Type Name and Version**

The framework determines the name of the workflow type from the workflow interface. The form of the default workflow type name is `{prefix}{name}`. The `{prefix}` is set to the name of the `@Workflow` interface followed by a `.` and the `{name}` is set to the name of the `@Execute` method. The default name of the workflow type in the preceding example is `MyWorkflow.startMyWF`. You can override the default name using the name parameter of the `@Execute` method. The default name of the workflow type in the example is `startMyWF`. The name must not be an empty string. Note that when you override the name using `@Execute`, the framework doesn't automatically prepend a prefix to it. You are free to use your own naming scheme.

The workflow version is specified using the `version` parameter of the `@Execute` annotation. There is no default for `version` and it must be explicitly specified; `version` is a free form string, and you are free to use your own versioning scheme.

**Signal Name**

The name of the signal can be specified using the name parameter of the `@Signal` annotation. If not specified, it is defaulted to the name of the signal method.

**Activity Type Name and Version**

The framework determines the name of the activity type from the activities interface. The form of the default activity type name is `{prefix}{name}`. The `{prefix}` is set to the name of the `@Activities` interface followed by a `.` and the `{name}` is set to the method name. The default `{prefix}` can be overridden in the `@Activities` annotation on the activities interface. You can also specify the activity type name using the `@Activity` annotation on the activity method. Note that when you override the name using `@Activity`, the framework will not automatically prepend a prefix to it. You are free to use your own naming scheme.

The activity version is specified using the `version` parameter of the `@Activities` annotation. This version is used as the default for all activities defined in the interface and can be overridden on a per-activity basis using the `@Activity` annotation.

**Default Task List**

The default task list can be configured using the `@WorkflowRegistrationOptions` and `@ActivityRegistrationOptions` annotations and setting the `defaultTaskList` parameter. By default, it is set to `USE_WORKER_TASK_LIST`. This is a special value that instructs the framework to use the task list that is configured on the worker object that is used to register the activity or workflow type. You can also choose to not register a default task list by setting the default task list to `NO_DEFAULT_TASK_LIST` using these annotations. This can be used in cases where you want to require that the task list be specified at run time. If no default task list has been registered, then you must specify the task list when starting the workflow or calling the activity method using the `StartWorkflowOptions` and `ActivitySchedulingOptions` parameters on the respective method overload of the generated client.

**Other Registration Options**

All workflow and activity type registration options that are allowed by the Amazon SWF API can be specified through the framework.

For a complete list of workflow registration options, see the following:

- `@Workflow` (p. 131)
Activity and Workflow Clients

Workflow and activity clients are generated by the framework based on the `@Workflow` and `@Activities` interfaces. Separate client interfaces are generated that contain methods and settings that make sense only on the client. If you are developing using Eclipse, this is done by the Amazon SWF Eclipse plug-in every time you save the file containing the appropriate interface. The generated code is placed in the generated sources directory in your project in the same package as the interface.

**Note**

Note that the default directory name used by Eclipse is `.apt_generated`. Eclipse doesn't show directories whose names start with a '.' in Package Explorer. Use a different directory name if you want to view the generated files in Project Explorer. In Eclipse, right-click the package in Package Explorer, and then choose _Properties_, _Java Compiler_, _Annotation processing_, and modify the _Generate source directory_ setting.

Workflow Clients

The generated artifacts for the workflow contain three client-side interfaces and the classes that implement them. The generated clients include:

- An _asynchronous client_ intended to be consumed from within a workflow implementation that provides asynchronous methods to start workflow executions and send signals
- An _external client_ that can be used to start executions and send signals and retrieve workflow state from outside the scope of a workflow implementation
- A _self client_ that can be used to create continuous workflows

For example, the generated client interfaces for the example `MyWorkflow` interface are:

```java
//Client for use from within a workflow
public interface MyWorkflowClient extends WorkflowClient { 
    Promise<Void> startMyWF( 
        int a, String b);

    Promise<Void> startMyWF( 
        int a, String b, 
        Promise<?>... waitFor);

    Promise<Void> startMyWF( 
        int a, String b, 
```
StartWorkflowOptions optionsOverride,
Promise<?>... waitFor);

Promise<Void> startMyWF(
    Promise<Integer> a,
    Promise<String> b);

Promise<Void> startMyWF(
    Promise<Integer> a,
    Promise<String> b,
    Promise<?>... waitFor);

Promise<Void> startMyWF(
    Promise<Integer> a,
    Promise<String> b,
    StartWorkflowOptions optionsOverride,
    Promise<?>... waitFor);

void signal1(
    int a, int b, String c);
}

//External client for use outside workflows
public interface MyWorkflowClientExternal extends WorkflowClientExternal
{
    void startMyWF(
        int a, String b);

    void startMyWF(
        int a, String b,
        StartWorkflowOptions optionsOverride);

    void signal1(
        int a, int b, String c);

    MyWorkflowState getState();
}

//self client for creating continuous workflows
public interface MyWorkflowSelfClient extends WorkflowSelfClient
{
    void startMyWF(
        int a, String b);

    void startMyWF(
        int a, String b,
        Promise<?>... waitFor);

    void startMyWF(
        int a, String b,
        StartWorkflowOptions optionsOverride,
        Promise<?>... waitFor);

    void startMyWF(
        Promise<Integer> a,
        Promise<String> b);

    void startMyWF(
        Promise<Integer> a,
        Promise<String> b,
        Promise<?>... waitFor);

    void startMyWF(
        Promise<Integer> a,
        Promise<String> b,
        StartWorkflowOptions optionsOverride,
The interfaces have overloaded methods corresponding to each method in the @Workflow interface that you declared.

The external client mirrors the methods on the @Workflow interface with one additional overload of the @Execute method that takes StartWorkflowOptions. You can use this overload to pass additional options when starting a new workflow execution. These options allow you to override the default task list, timeout settings, and associate tags with the workflow execution.

On the other hand, the asynchronous client has methods that allow asynchronous invocation of the @Execute method. The following method overloads are generated in the client interface for the @Execute method in the workflow interface:

1. An overload that takes the original arguments as is. The return type of this overload will be Promise<Void> if the original method returned void; otherwise, it will be the Promise<> as declared on the original method. For example:

   Original method:
   ```java
   void startMyWF(int a, String b);
   ```

   Generated method:
   ```java
   Promise<Void> startMyWF(int a, String b);
   ```

   This overload should be used when all the arguments of the workflow are available and don't need to be waited for.

2. An overload that takes the original arguments as is and additional variable arguments of type Promise<?>. The return type of this overload will be Promise<Void> if the original method returned void; otherwise, it will be the Promise<> as declared on the original method. For example:

   Original method:
   ```java
   void startMyWF(int a, String b);
   ```

   Generated method:
   ```java
   Promise<void> startMyWF(int a, String b, Promise<?>...waitFor);
   ```

   This overload should be used when all the arguments of the workflow are available and don't need to be waited for, but you want to wait for some other promises to become ready. The variable argument can be used to pass such Promise<?> objects that were not declared as arguments, but you want to wait for before executing the call.

3. An overload that takes the original arguments as is, an additional argument of type StartWorkflowOptions and additional variable arguments of type Promise<?>. The return type of this overload will be Promise<Void> if the original method returned void; otherwise, it will be the Promise<> as declared on the original method. For example:

   Original method:
   ```java
   void startMyWF(int a, String b);
   ```

   Generated method:
Promise<void> startMyWF(
   int a,
   String b,
   StartWorkflowOptions optionOverrides,
   Promise<?>...waitFor);

This overload should be used when all the arguments of the workflow are available and don't need to be waited for, when you want to override default settings used to start the workflow execution, or when you want to wait for some other promises to become ready. The variable argument can be used to pass such Promise<?> objects that were not declared as arguments, but you want to wait for before executing the call.

4. An overload with each argument in the original method replaced with a Promise<> wrapper. The return type of this overload will be Promise<Void> if the original method returned void; otherwise, it will be the Promise<> as declared on the original method. For example:

Original method:

void startMyWF(int a, String b);

Generated method:

Promise<Void> startMyWF(
   Promise<Integer> a,
   Promise<String> b);

This overload should be used when the arguments to be passed to the workflow execution are to be evaluated asynchronously. A call to this method overload will not execute until all arguments passed to it become ready.

If some of the arguments are already ready, then convert them to a Promise that is already in ready state through the Promise.asPromise(value) method. For example:

Promise<Integer> a = getA();
String b = getB();
startMyWF(a, Promise.asPromise(b));

5. An overload with each argument in the original method is replaced with a Promise<> wrapper. The overload also has additional variable arguments of type Promise<?>. The return type of this overload will be Promise<Void> if the original method returned void; otherwise, it will be the Promise<> as declared on the original method. For example:

Original method:

void startMyWF(int a, String b);

Generated method:

Promise<Void> startMyWF(
   Promise<Integer> a,
   Promise<String> b,
   Promise<?>...waitFor);

This overload should be used when the arguments to be passed to the workflow execution are to be evaluated asynchronously and you want to wait for some other promises to become ready as well. A call to this method overload will not execute until all arguments passed to it become ready.
6. An overload with each argument in the original method replaced with a Promise<?> wrapper. The overload also has an additional argument of type StartWorkflowOptions and variable arguments of type Promise<?>. The return type of this overload will be Promise<Void> if the original method returned void; otherwise, it will be the Promise<> as declared on the original method. For example:

Original method:

```java
void startMyWF(int a, String b);
```

Generated method:

```java
Promise<Void> startMyWF(
    Promise<Integer> a,
    Promise<String> b,
    StartWorkflowOptions optionOverrides,
    Promise<?>...waitFor);
```

Use this overload when the arguments to be passed to the workflow execution will be evaluated asynchronously and you want to override default settings used to start the workflow execution. A call to this method overload will not execute until all arguments passed to it become ready.

A method is also generated corresponding to each signal in the workflow interface—for example:

Original method:

```java
void signal1(int a, int b, String c);
```

Generated method:

```java
void signal1(int a, int b, String c);
```

The asynchronous client doesn't contain a method corresponding to the method annotated with @GetState in the original interface. Since retrieval of state requires a web service call, it is not suitable for use within a workflow. Hence, it is provided only through the external client.

The self client is intended to be used from within a workflow to start a new execution on completion of the current execution. The methods on this client are similar to the ones on the asynchronous client, but return void. This client doesn't have methods corresponding to methods annotated with @Signal and @GetState. For more details, see the Continuous Workflows (p. 87).

The generated clients derive from base interfaces: WorkflowClient and WorkflowClientExternal, respectively, which provide methods that you can use to cancel or terminate the workflow execution. For more details about these interfaces, see the AWS SDK for Java documentation.

The generated clients allow you to interact with workflow executions in a strongly typed fashion. Once created, an instance of a generated client is tied to a specific workflow execution and can be used only for that execution. In addition, the framework also provides dynamic clients that are not specific to a workflow type or execution. The generated clients rely on this client under the covers. You may also directly use these clients. See the section on Dynamic Clients (p. 70).

The framework also generates factories for creating the strongly typed clients. The generated client factories for the example MyWorkflow interface are:

```java
//Factory for clients to be used from within a workflow
```
public interface MyWorkflowClientFactory
extends WorkflowClientFactory<MyWorkflowClient>
{
}

//Factory for clients to be used outside the scope of a workflow
public interface MyWorkflowClientExternalFactory
{
    GenericWorkflowClientExternal getGenericClient();
    void setGenericClient(GenericWorkflowClientExternal genericClient);
    DataConverter getDataConverter();
    void setDataConverter(DataConverter dataConverter);
    StartWorkflowOptions getStartWorkflowOptions();
    void setStartWorkflowOptions(StartWorkflowOptions startWorkflowOptions);
    MyWorkflowClientExternal getClient();
    MyWorkflowClientExternal getClient(String workflowId);
    MyWorkflowClientExternal getClient(WorkflowExecution workflowExecution);
    MyWorkflowClientExternal getClient(
        WorkflowExecution workflowExecution,
        GenericWorkflowClientExternal genericClient,
        DataConverter dataConverter,
        StartWorkflowOptions options);
}

The WorkflowClientFactory base interface is:

public interface WorkflowClientFactory<T> {  
    GenericWorkflowClient getGenericClient();
    void setGenericClient(GenericWorkflowClient genericClient);
    DataConverter getDataConverter();
    void setDataConverter(DataConverter dataConverter);
    StartWorkflowOptions getStartWorkflowOptions();
    void setStartWorkflowOptions(StartWorkflowOptions startWorkflowOptions);
    T getClient();
    T getClient(String workflowId);
    T getClient(WorkflowExecution workflowExecution);
    T getClient(WorkflowExecution workflowExecution,
        StartWorkflowOptions options);
    T getClient(WorkflowExecution workflowExecution,
        StartWorkflowOptions options,
        DataConverter dataConverter);
}

You should use these factories to create instances of the client. The factory allows you to configure the
generic client (the generic client should be used for providing custom client implementation) and the
DataConverter used by the client to marshal data, as well as the options used to start the workflow
execution. For more details, see the DataConverters (p. 89) and Child Workflow Executions (p. 85)
sections. The StartWorkflowOptions contains settings that you can use to override the defaults—for
example, timeouts—specified at registration time. For more details about the StartWorkflowOptions
class, see the AWS SDK for Java documentation.

The external client can be used to start workflow executions from outside of the scope of a workflow
while the asynchronous client can be used to start a workflow execution from code within a workflow. In
order to start an execution, you simply use the generated client to call the method that corresponds to
the method annotated with @Execute in the workflow interface.

The framework also generates implementation classes for the client interfaces. These clients create and
send requests to Amazon SWF to perform the appropriate action. The client version of the @Execute
method either starts a new workflow execution or creates a child workflow execution using Amazon SWF
APIs. Similarly, the client version of the @Signal method uses Amazon SWF APIs to send a signal.
Note
The external workflow client must be configured with the Amazon SWF client and domain. You can either use the client factory constructor that takes these as parameters or pass in a generic client implementation that is already configured with the Amazon SWF client and domain. The framework walks the type hierarchy of the workflow interface and also generates client interfaces for parent workflow interfaces and derives from them.

Activity Clients
Similar to the workflow client, a client is generated for each interface annotated with @Activities. The generated artifacts include a client side interface and a client class. The generated interface for the example @Activities interface above (MyActivities) is as follows:

```java
public interface MyActivitiesClient extends ActivitiesClient
{
    Promise<Integer> activity1();
    Promise<Integer> activity1(Promise<?>... waitFor);
    Promise<Integer> activity1(ActivitySchedulingOptions optionsOverride,
                                    Promise<?>... waitFor);
    Promise<Void> activity2(int a);
    Promise<Void> activity2(int a,
                              Promise<?>... waitFor);
    Promise<Void> activity2(int a,
                              ActivitySchedulingOptions optionsOverride,
                              Promise<?>... waitFor);
    Promise<Void> activity2(Promise<Integer> a);
    Promise<Void> activity2(Promise<Integer> a,
                              Promise<?>... waitFor);
    Promise<Void> activity2(Promise<Integer> a,
                              ActivitySchedulingOptions optionsOverride,
                              Promise<?>... waitFor);
}
```

The interface contains a set of overloaded methods corresponding to each activity method in the @Activities interface. These overloads are provided for convenience and allow calling activities asynchronously. For each activity method in the @Activities interface, the following method overloads are generated in the client interface:

1. An overload that takes the original arguments as is. The return type of this overload is Promise\<T\>, where T is the return type of the original method. For example:

   Original method:
   ```java
   void activity2(int foo);
   ```

   Generated method:
   ```java
   Promise<Void> activity2(int foo);
   ```

   This overload should be used when all the arguments of the workflow are available and don’t need to be waited for.

2. An overload that takes the original arguments as is, an argument of type ActivitySchedulingOptions and additional variable arguments of type Promise<?>. The return type of this overload is Promise\<T\>, where T is the return type of the original method. For example:

   Original method:
   ```java
   ```

   Generated method:
   ```java
   ```

   This overload should be used when all the arguments of the workflow are available and don’t need to be waited for.
void activity2(int foo);

Generated method:

Promise<Void> activity2(
    int foo,
    ActivitySchedulingOptions optionsOverride,
    Promise<?>... waitFor);

This overload should be used when all the arguments of the workflow are available and don't need to be waited for, when you want to override the default settings, or when you want to wait for additional Promises to become ready. The variable arguments can be used to pass such additional Promise<?> objects that were not declared as arguments, but you want to wait for before executing the call.

3. An overload with each argument in the original method replaced with a Promise<> wrapper. The return type of this overload is Promise<T>, where T is the return type of the original method. For example:

Original method:

void activity2(int foo);

Generated method:

Promise<Void> activity2(Promise<Integer> foo);

This overload should be used when the arguments to be passed to the activity will be evaluated asynchronously. A call to this method overload will not execute until all arguments passed to it become ready.

4. An overload with each argument in the original method replaced with a Promise<> wrapper. The overload also has an additional argument of type ActivitySchedulingOptions and variable arguments of type Promise<?>. The return type of this overload is Promise<T>, where T is the return type of the original method. For example:

Original method:

void activity2(int foo);

Generated method:

Promise<Void> activity2(
    Promise<Integer> foo,
    ActivitySchedulingOptions optionsOverride,
    Promise<?>... waitFor);

This overload should be used when the arguments to be passed to the activity will be evaluated asynchronously, when you want to override the default settings registered with the type, or when you want to wait for additional Promises to become ready. A call to this method overload will not execute until all arguments passed to it become ready. The generated client class implements this interface. The implementation of each interface method creates and sends a request to Amazon SWF to schedule an activity task of the appropriate type using Amazon SWF APIs.

5. An overload that takes the original arguments as is and additional variable arguments of type Promise<?>. The return type of this overload is Promise<T>, where T is the return type of the original method. For example:
Scheduling Options

The generated activity client allows you to pass in `ActivitySchedulingOptions` as an argument. The `ActivitySchedulingOptions` structure contains settings that determine the configuration of the activity task that the framework schedules in Amazon SWF. These settings override the defaults that are specified as registration options. To specify scheduling options dynamically, create an
ActivitySchedulingOptions object, configure it as desired, and pass it to the activity method. In
the following example, we have specified the task list that should be used for the activity task. This will
override the default registered task list for this invocation of the activity.

```java
public class OrderProcessingWorkflowImpl implements OrderProcessingWorkflow {

    OrderProcessingActivitiesClient activitiesClient = new OrderProcessingActivitiesClientImpl();

    // Workflow entry point
    @Override
    public void processOrder(Order order) {
        Promise<Void> paymentProcessed = activitiesClient.processPayment(order);
        ActivitySchedulingOptions schedulingOptions = new ActivitySchedulingOptions();
        if (order.getLocation() == "Japan") {
            schedulingOptions.setTaskList("TasklistAsia");
        } else {
            schedulingOptions.setTaskList("TasklistNorthAmerica");
        }
        activitiesClient.shipOrder(order,
                                   schedulingOptions,
                                   paymentProcessed);
    }
}
```

**Dynamic Clients**

In addition to the generated clients, the framework also provides general purpose clients
— DynamicWorkflowClient and DynamicActivityClient — that you can use to dynamically start
workflow executions, send signals, schedule activities, etc. For instance, you may want to schedule
an activity whose type isn't known at design time. You can use the DynamicActivityClient for
scheduling such an activity task. Similarly, you can dynamically schedule a child workflow execution by
using the DynamicWorkflowClient. In the following example, the workflow looks up the activity from
a database and uses the dynamic activity client to schedule it:

```java
// Workflow entrypoint
@Override
public void start() {
    MyActivitiesClient client = new MyActivitiesClientImpl();
    Promise<ActivityType> activityType = client.lookUpActivityFromDB();
    Promise<String> input = client.getInput(activityType);
    scheduleDynamicActivity(activityType,
                             input);
}

// Asynchronous
@Asynchronous
void scheduleDynamicActivity(Promise<ActivityType> type,
                             Promise<String> input){
    Promise<?>[] args = new Promise<?>[1];
    args[0] = input;
    DynamicActivitiesClient activityClient = new DynamicActivitiesClientImpl();
    activityClient.scheduleActivity(type.get(),
                                     args,
                                     null,
                                     Void.class);
}
```

For more details, see the AWS SDK for Java documentation.
Signaling and Canceling Workflow Executions

The generated workflow client has methods corresponding to each signal that can be sent to the workflow. You can use them from within a workflow to send signals to other workflow executions. This provides a typed mechanism for sending signals. However, sometimes you may need to dynamically determine the signal name—for example, when the signal name is received in a message. You can use the dynamic workflow client to dynamically send signals to any workflow execution. Similarly, you can use the client to request cancellation of another workflow execution.

In the following example, the workflow looks up the execution to send a signal to from a database and sends the signal dynamically using the dynamic workflow client.

```java
//Workflow entrypoint
public void start()
{
    MyActivitiesClient client = new MyActivitiesClientImpl();
    Promise<WorkflowExecution> execution = client.lookUpExecutionInDB();
    Promise<String> signalName = client.getSignalToSend();
    Promise<String> input = client.getInput(signalName);
    sendDynamicSignal(execution, signalName, input);
}

@Asynchronous
void sendDynamicSignal(
    Promise<WorkflowExecution> execution,
    Promise<String> signalName,
    Promise<String> input)
{
    DynamicWorkflowClient workflowClient
        = new DynamicWorkflowClientImpl(execution.get());
    Object[] args = new Promise<?>[1];
    args[0] = input.get();
    workflowClient.signalWorkflowExecution(signalName.get(), args);
}
```

Workflow Implementation

In order to implement a workflow, you write a class that implements the desired `@Workflow` interface. For instance, the example workflow interface (MyWorkflow) can be implemented like so:

```java
public class MyWFImpl implements MyWorkflow
{
    MyActivitiesClient client = new MyActivitiesClientImpl();
    @Override
    public void startMyWF(int a, String b){
        Promise<Integer> result = client.activity1();
        client.activity2(result);
    }
    @Override
    public void signal1(int a, int b, String c){
        //Process signal
        client.activity2(a + b);
    }
}
```

The `@Execute` method in this class is the entry point of the workflow logic. Since the framework uses replay to reconstruct the object state when a decision task is to be processed, a new object is created for each decision task.
The use of Promise<T> as a parameter is disallowed in the @Execute method within a @Workflow interface. This is done because making an asynchronous call is purely a decision of the caller. The workflow implementation itself doesn't depend on whether the invocation was synchronous or asynchronous. Therefore, the generated client interface has overloads that take Promise<T> parameters so that these methods can be called asynchronously.

The return type of an @Execute method can only be void or Promise<T>. Note that a return type of the corresponding external client is void and not Promise<>. Since the external client isn't intended to be used from the asynchronous code, the external client doesn't return Promise objects. For getting results of workflow executions stated externally, you can design the workflow to update state in an external data store through an activity. Amazon SWF's visibility APIs can also be used to retrieve the result of a workflow for diagnostic purposes. It isn't recommended that you use the visibility APIs to retrieve results of workflow executions as a general practice since these API calls may get throttled by Amazon SWF. The visibility APIs require you to identify the workflow execution using a WorkflowExecution structure. You can get this structure from the generated workflow client by calling the getWorkflowExecution method. This method will return the WorkflowExecution structure corresponding to the workflow execution that the client is bound to. See the Amazon Simple Workflow Service API Reference for more details about the visibility APIs.

When calling activities from your workflow implementation, you should use the generated activities client. Similarly, to send signals, use the generated workflow clients.

**Decision Context**

The framework provides an ambient context anytime workflow code is executed by the framework. This context provides context-specific functionality that you may access in your workflow implementation, such as creating a timer. See the section on Execution Context (p. 83) for more information.

**Exposing Execution State**

Amazon SWF allows you to add custom state in the workflow history. The latest state reported by the workflow execution is returned to you through visibility calls to the Amazon SWF service and in the Amazon SWF console. For example, in an order processing workflow, you may report the order status at different stages like 'order received', 'order shipped', and so on. In the AWS Flow Framework for Java, this is accomplished through a method on your workflow interface that is annotated with the @GetState annotation. When the decider is done processing a decision task, it calls this method to get the latest state from the workflow implementation. Besides visibility calls, the state can also be retrieved using the generated external client (which uses the visibility API calls internally).

The following example demonstrates how to set the execution context.

```java
@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
   defaultTaskStartToCloseTimeoutSeconds = 10)
public interface PeriodicWorkflow {
    @Execute(version = "1.0")
    void periodicWorkflow();

    @GetState
    String getState();
}

@Activities(version = "1.0")
@ActivityRegistrationOptions(defaultTaskScheduleToStartTimeoutSeconds = 300,
   defaultTaskStartToCloseTimeoutSeconds = 3600)
public interface PeriodicActivity {
    void activity1();
}
```
public class PeriodicWorkflowImpl implements PeriodicWorkflow {
    private DecisionContextProvider contextProvider
        = new DecisionContextProviderImpl();
    private WorkflowClock clock
        = contextProvider.getDecisionContext().getWorkflowClock();
    private PeriodicActivityClient activityClient
        = new PeriodicActivityClientImpl();
    private String state;

    @Override
    public void periodicWorkflow() {
        state = "Just Started";
        callPeriodicActivity(0);
    }

    @Asynchronous
    private void callPeriodicActivity(int count,
        Promise<?>... waitFor)
    {
        if(count == 100) {
            state = "Finished Processing";
            return;
        }

        // call activity
        activityClient.activity1();

        // Repeat the activity after 1 hour.
        Promise<Void> timer = clock.createTimer(3600);
        state = "Waiting for timer to fire. Count = "+count;
        callPeriodicActivity(count+1, timer);
    }

    @Override
    public String getState() {
        return state;
    }
}

public class PeriodicActivityImpl implements PeriodicActivity {
    @Override
    public static void activity1()
    {
        ...
    }
}

The generated external client can be used to retrieve the latest state of the workflow execution at any time.

PeriodicWorkflowClientExternal client
    = new PeriodicWorkflowClientExternalFactoryImpl().getClient();
System.out.println(client.getState());

In the above example, the execution state is reported at various stages. When the workflow instance starts, periodicWorkflow reports the initial state as 'Just Started'. Each call to
callPeriodicActivity then updates the workflow state. Once activity1 has been called 100 times, the method returns and the workflow instance completes.

**Workflow Locals**

Sometimes, you may have a need for the use of static variables in your workflow implementation. For example, you may want to store a counter that is to be accessed from various places (possibly different classes) in the implementation of the workflow. However, you can't rely on static variables in your workflows because static variables are shared across threads, which is problematic because a worker may process different decision tasks on different threads at the same time. Alternatively, you may store such state in a field on the workflow implementation, but then you will need to pass the implementation object around. To address this need, the framework provides a `WorkflowExecutionLocal<?>` class. Any state that needs to have static variable like semantics should be kept as an instance local using `WorkflowExecutionLocal<?>`. You can declare and use a static variable of this type. For example, in the following snippet, a `WorkflowExecutionLocal<String>` is used to store a user name.

```java
public class MyWFImpl implements MyWF {
    public static WorkflowExecutionLocal<String> username
        = new WorkflowExecutionLocal<String>();

    @Override
    public void start(String username){
        this.username.set(username);
        Processor p = new Processor();
        p.updateLastLogin();
        p.greetUser();
    }

    public static WorkflowExecutionLocal<String> getUsername() {
        return username;
    }

    public static void setUsername(WorkflowExecutionLocal<String> username) {
        MyWFImpl.username = username;
    }
}

public class Processor {
    void updateLastLogin(){
        UserActivitiesClient c = new UserActivitiesClientImpl();
        c.refreshLastLogin(MyWFImpl.getUsername().get());
    }

    void greetUser(){
        GreetingActivitiesClient c = new GreetingActivitiesClientImpl();
        c.greetUser(MyWFImpl.getUsername().get());
    }
}
```

**Activity Implementation**

Activities are implemented by providing an implementation of the `@Activities` interface. The AWS Flow Framework for Java uses the activity implementation instances configured on the worker to process activity tasks at run time. The worker automatically looks up the activity implementation of the appropriate type.

You can use properties and fields to pass resources to activity instances, such as database connections. Since the activity implementation object may be accessed from multiple threads, shared resources must be thread safe.
Note that the activity implementation doesn't take parameters of type Promise<> or return objects of that type. This is because the implementation of the activity should not depend on how it was invoked (synchronously or asynchronously).

The activities interface shown before can be implemented like this:

```java
public class MyActivitiesImpl implements MyActivities {
    @Override
    @ManualActivityCompletion
    public int activity1() {
        //implementation
    }

    @Override
    public void activity2(int foo) {
        //implementation
    }
}
```

A thread local context is available to the activity implementation that can be used to retrieve the task object, data converter object being used, etc. The current context can be accessed through `ActivityExecutionContextProvider.getActivityExecutionContext()`. For more details, see the AWS SDK for Java documentation for `ActivityExecutionContext` and the section Execution Context (p. 83).

**Manually Completing Activities**

The `@ManualActivityCompletion` annotation in the example above is an optional annotation. It is allowed only on methods that implement an activity and is used to configure the activity to not automatically complete when the activity method returns. This could be useful when you want to complete the activity asynchronously—for example, manually after a human action has been completed.

By default, the framework considers the activity completed when your activity method returns. This means that the activity worker reports activity task completion to Amazon SWF and provides it with the results (if any). However, there are use cases where you don't want the activity task to be marked completed when the activity method returns. This is especially useful when you are modeling human tasks. For example, the activity method may send an email to a person who must complete some work before the activity task is completed. In such cases, you can annotate the activity method with `@ManualActivityCompletion` annotation to tell the activity worker that it should not complete the activity automatically. In order to complete the activity manually, you can either use the `ManualActivityCompletionClient` provided in the framework or use the `RespondActivityTaskCompleted` method on the Amazon SWF Java client provided in the Amazon SWF SDK. For more details, see the AWS SDK for Java documentation.

In order to complete the activity task, you need to provide a task token. The task token is used by Amazon SWF to uniquely identify tasks. You can access this token from the `ActivityExecutionContext` in your activity implementation. You must pass this token to the party that is responsible for completing the task. This token can be retrieved from the `ActivityExecutionContext` by calling `ActivityExecutionContextProvider.getActivityExecutionContext().getTaskToken()`.

The `getName` activity of the Hello World example can be implemented to send an email asking someone to provide a greeting message:

```java
@ManualActivityCompletion
@Override
public String getName() throws InterruptedException {
    ActivityExecutionContext executionContext
```
public class CompleteActivityTask {
    public void completeGetNameActivity(String taskToken) {
        AmazonSimpleWorkflow swfClient = new AmazonSimpleWorkflowClient(...); // use AWS access keys
        ManualActivityCompletionClientFactory manualCompletionClientFactory = new ManualActivityCompletionClientFactoryImpl(swfClient);
        ManualActivityCompletionClient manualCompletionClient = manualCompletionClientFactory.getClient(taskToken);
        String result = "Hello World!"; manualCompletionClient.complete(result);
    }

    public void failGetNameActivity(String taskToken, Throwable failure) {
        AmazonSimpleWorkflow swfClient = new AmazonSimpleWorkflowClient(...); // use AWS access keys
        ManualActivityCompletionClientFactory manualCompletionClientFactory = new ManualActivityCompletionClientFactoryImpl(swfClient);
        ManualActivityCompletionClient manualCompletionClient = manualCompletionClientFactory.getClient(taskToken);
        manualCompletionClient.fail(failure);
    }
}

The following code snippet can be used to provide the greeting and close the task by using the ManualActivityCompletionClient. Alternatively, you can also fail the task:

### Implementing AWS Lambda Tasks

**Topics**

- About AWS Lambda (p. 76)
- Benefits and Limitations of using Lambda Tasks (p. 77)
- Using Lambda tasks in your AWS Flow Framework for Java workflows (p. 77)
- View the HelloLambda sample (p. 79)

**About AWS Lambda**

AWS Lambda is a fully managed compute service that runs your code in response to events generated by custom code or from various AWS services such as Amazon S3, DynamoDB, Amazon Kinesis, Amazon SNS, and Amazon Cognito. For more information about Lambda, see the AWS Lambda Developer Guide.

Amazon Simple Workflow Service provides a Lambda task so that you can run Lambda functions in place of, or alongside traditional Amazon SWF activities.

**Important**

Your AWS account will be charged for Lambda executions (requests) executed by Amazon SWF on your behalf. For details about Lambda pricing, see http://aws.amazon.com/lambda/pricing/.
Benefits and Limitations of using Lambda Tasks

There are a number of benefits of using Lambda tasks in place of a traditional Amazon SWF activity:

- Lambda tasks don’t need to be registered or versioned like Amazon SWF activity types.
- You can use any existing Lambda functions that you've already defined in your workflows.
- Lambda functions are called directly by Amazon SWF; there is no need for you to implement a worker program to execute them as you must do with traditional activities.
- Lambda provides you with metrics and logs for tracking and analyzing your function executions.

There are also a number of limitations regarding Lambda tasks that you should be aware of:

- Lambda tasks can only be run in AWS regions that provide support for Lambda. See Lambda Regions and Endpoints in the Amazon Web Services General Reference for details about the currently-supported regions for Lambda.
- Lambda tasks are currently supported only by the base SWF HTTP API and in the AWS Flow Framework for Java. There is currently no support for Lambda tasks in the AWS Flow Framework for Ruby.

Using Lambda tasks in your AWS Flow Framework for Java workflows

There are three requirements to use Lambda tasks in your AWS Flow Framework for Java workflows:

- A Lambda function to execute. You can use any Lambda function that you've defined. For more information about how to create Lambda functions, see the AWS Lambda Developer Guide.
- An IAM role that provides access to execute Lambda functions from your Amazon SWF workflows.
- Code to schedule the Lambda task from within your workflow.

Set up an IAM role

Before you can invoke Lambda functions from Amazon SWF you must provide an IAM role that provides access to Lambda from Amazon SWF. You can either:

- choose a pre-defined role, AWSLambdaRole, to give your workflows permission to invoke any Lambda function associated with your account.
- define your own policy and associated role to give workflows permission to invoke particular Lambda functions, specified by their Amazon Resource Names (ARNs).

Providing Amazon SWF with access to invoke any Lambda role

You can use the pre-defined role, AWSLambdaRole, to give your Amazon SWF workflows the ability to invoke any Lambda function associated with your account.

To use AWSLambdaRole to give Amazon SWF access to invoke Lambda functions

1. Open the Amazon IAM console.
2. Choose Roles, then Create New Role.
3. Give your role a name, such as swf-lambda and choose Next Step.
4. Under AWS Service Roles, choose Amazon SWF, and choose Next Step.
5. On the **Attach Policy** screen, choose `AWSLambdaRole` from the list.
6. Choose **Next Step** and then **Create Role** once you've reviewed the role.

### Defining an IAM role to provide access to invoke a specific Lambda function

If you want to provide access to invoke a specific Lambda function from your workflow, you will need to define your own IAM policy.

**To create an IAM policy to provide access to a particular Lambda function**

1. Open the Amazon IAM console.
2. Choose **Policies**, then **Create Policy**.
3. Choose **Copy an AWS Managed Policy** and select `AWSLambdaRole` from the list. A policy will be generated for you. Optionally edit its name and description to suit your needs.
4. In the **Resource** field of the **Policy Document**, add the ARN of your Lambda function(s). For example:

   ```json
   {
   "Version": "2012-10-17",
   "Statement": [
       { "Effect": "Allow",
         "Action": [ "lambda:InvokeFunction" ],
       }
   ]
   }
   ```

   **Note**
   For a complete description of how to specify resources in an IAM role, see **Overview of IAM Policies** in *Using IAM*.
5. Choose **Create Policy** to finish creating your policy.

You can then select this policy when creating a new IAM role, and use that role to give invoke access to your Amazon SWF workflows. This procedure is very similar to creating a role with the `AWSLambdaRole` policy. instead, choose your own policy when creating the role.

**To create a Amazon SWF role using your Lambda policy**

1. Open the Amazon IAM console.
2. Choose **Roles**, then **Create New Role**.
3. Give your role a name, such as `swf-lambda-function` and choose **Next Step**.
4. Under **AWS Service Roles**, choose **Amazon SWF**, and choose **Next Step**.
5. On the **Attach Policy** screen, choose your Lambda function-specific policy from the list.
6. Choose **Next Step** and then **Create Role** once you've reviewed the role.

### Schedule a Lambda task for execution

Once you've defined an IAM role that allows you to invoke Lambda functions, you can schedule them for execution as part of your workflow.
To schedule a Lambda task for execution

1. In your workflow implementation, get an instance of LambdaFunctionClient by calling getLambdaFunctionClient() on a DecisionContext instance.

```java
// Get a LambdaFunctionClient instance
DecisionContextProvider decisionProvider = new DecisionContextProviderImpl();
DecisionContext decisionContext = decisionProvider.getDecisionContext();
LambdaFunctionClient lambdaClient = decisionContext.getLambdaFunctionClient();
```

2. Schedule the task using the scheduleLambdaFunction() method on the LambdaFunctionClient, passing it the name of the Lambda function that you created and any input data for the Lambda task.

```java
// Schedule the Lambda function for execution, using your IAM role for access.
String lambda_function_name = "The name of your Lambda function.";
String lambda_function_input = "Input data for your Lambda task.";
lambdaClient.scheduleLambdaFunction(lambda_function_name, lambda_function_input);
```

3. In your workflow execution starter, add the IAM lambda role to your default workflow options by using StartWorkflowOptions.withLambdaRole(), and then pass the options when starting the workflow.

```java
// Workflow client classes are generated for you when you use the @Workflow // annotation on your workflow interface declaration.
MyWorkflowClientExternalFactory clientFactory =
    new MyWorkflowClientExternalFactoryImpl(sdk_swf_client, swf_domain);
MyWorkflowClientExternal workflow_client = clientFactory.getClient();

// Give the ARN of an IAM role that allows SWF to invoke Lambda functions on // your behalf.
String lambda_iam_role = "arn:aws:iam::111111000000:role/swf_lambda_role";
StartWorkflowOptions workflow_options =
    new StartWorkflowOptions().withLambdaRole(lambda_iam_role);

// Start the workflow execution
workflow_client.helloWorld("User", workflow_options);
```

View the HelloLambda sample

A sample that provides an implementation of a workflow that uses a Lambda task is provided in the AWS SDK for Java. To view and/or run it, download the source.

A full description of how to build and run the HelloLambda sample is provided in the README file provided with the AWS Flow Framework for Java samples.
Running Programs Written with the AWS Flow Framework for Java

Topics

• WorkflowWorker (p. 81)
• ActivityWorker (p. 81)
• Worker Threading Model (p. 81)
• Worker Extensibility (p. 82)

The framework provides worker classes to initialize the AWS Flow Framework for Java runtime and communicate with Amazon SWF. In order to implement a workflow or an activity worker, you must create and start an instance of a worker class. These worker classes are responsible for managing ongoing asynchronous operations, invoking asynchronous methods that become unblocked, and communicating with Amazon SWF. They can be configured with workflow and activity implementations, the number of threads, the task list to poll, and so on.

The framework comes with two worker classes, one for activities and one for workflows. In order to run the workflow logic, you use the WorkflowWorker class. Similarly for activities the ActivityWorker class is used. These classes automatically poll Amazon SWF for activity tasks and invoke the appropriate methods in your implementation.

The following example shows how to instantiate a WorkflowWorker and start polling for tasks:

```java
AmazonSimpleWorkflow swfClient = new AmazonSimpleWorkflowClient(awsCredentials);
WorkflowWorker worker = new WorkflowWorker(swfClient, "domain1", "tasklist1");
// Add workflow implementation types
worker.addWorkflowImplementationType(MyWorkflowImpl.class);

// Start worker
worker.start();
```

The basic steps to create an instance of the ActivityWorker and starting polling for tasks are as follows:

```java
AmazonSimpleWorkflow swfClient = new AmazonSimpleWorkflowClient(awsCredentials);
ActivityWorker worker = new ActivityWorker(swfClient, "domain1", "tasklist1");
worker.addActivitiesImplementation(new MyActivitiesImpl());

// Start worker
worker.start();
```

When you want to shut down an activity or decider, your application should shut down the instances of the worker classes being used as well as the Amazon SWF Java client instance. This will ensure that all resources used by the worker classes are properly released.

```java
worker.shutdown();
worker.awaitTermination(1, TimeUnit.MINUTES);
```

In order to start an execution, simply create an instance of the generated external client and call the @Execute method.
MyWorkflowClientExternalFactory factory = new MyWorkflowClientExternalFactoryImpl();
MyWorkflowClientExternal client = factory.getClient();
client.start();

**WorkflowWorker**

As the name suggests, this worker class is intended for use by the workflow implementation. It is configured with a task list and the workflow implementation type. The worker class runs a loop to poll for decision tasks in the specified task list. When a decision task is received, it creates an instance of the workflow implementation and calls the `@Execute` method to process the task.

**ActivityWorker**

For implementing activity workers, you can use the `ActivityWorker` class to conveniently poll a task list for activity tasks. You configure the activity worker with activity implementation objects. This worker class runs a loop to poll for activity tasks in the specified task list. When an activity task is received, it looks up the appropriate implementation that you provided and calls the activity method to process the task. Unlike the `WorkflowWorker`, which calls the factory to create a new instance for every decision task, the `ActivityWorker` simply uses the object you provided.

The `ActivityWorker` class uses the AWS Flow Framework for Java annotations to determine the registration and execution options.

**Worker Threading Model**

In the AWS Flow Framework for Java, the embodiment of an activity or decider is an instance of the worker class. Your application is responsible for configuring and instantiating the worker object on each machine and process that should act as a worker. The worker object then automatically receives tasks from Amazon SWF, dispatches them to your activity or workflow implementation and reports results to Amazon SWF. It is possible for a single workflow instance to span many workers. When Amazon SWF has one or more pending activity tasks, it assigns a task to the first available worker, then the next one, and so on. This makes it possible for tasks belonging to the same workflow instance to be processed on different workers concurrently.

Moreover, each worker can be configured to process tasks on multiple threads. This means that the activity tasks of a workflow instance can run concurrently even if there is only one worker.
Decision tasks behave similarly with the exception that Amazon SWF guarantees that for a given workflow execution only one decision can be executed at a time. A single workflow execution will typically require multiple decision tasks; hence, it may end up executing on multiple processes and threads as well. The decider is configured with the type of the workflow implementation. When a decision task is received by the decider, it creates an instance (object) of the workflow implementation. The framework provides an extensible factory pattern for creating these instances. The default workflow factory creates a new object every time. You can provide custom factories to override this behavior.

Contrary to deciders, which are configured with workflow implementation types, activity workers are configured with instances (objects) of the activity implementations. When an activity task is received by the activity worker, it is dispatched to the appropriate activity implementation object.

The workflow worker maintains a single pool of threads and executes the workflow on the same thread that was used to poll Amazon SWF for the task. Since activities are long running (at least when compared to the workflow logic), the activity worker class maintains two separate pools of threads; one for polling Amazon SWF for activity tasks and the other for processing tasks by executing the activity implementation. This allows you to configure the number of threads to poll for tasks separate from the number of threads to execute them. For example, you can have a small number of threads to poll and a large number of threads to execute the tasks. The activity worker class polls Amazon SWF for a task only when it has a free poll thread as well as a free thread to process the task.

This threading and instancing behavior implies that:

1. Activity implementations must be stateless. You should not use instance variables to store application state in activity objects. You may, however, use fields to store resources such as database connections.
2. Activity implementations must be thread safe. Since the same instance may be used to process tasks from different threads at the same time, access to shared resources from the activity code must be synchronized.
3. Workflow implementation can be stateful, and instance variables may be used to store state. Even though a new instance of the workflow implementation is created to process each decision task, the framework will ensure that state is properly recreated. However, the workflow implementation must be deterministic. See the section Under the Hood (p. 119) for more details.
4. Workflow implementations don't need to be thread safe when using the default factory. The default implementation ensures that only one thread uses an instance of the workflow implementation at a time.

**Worker Extensibility**

The AWS Flow Framework for Java also contains a couple of low-level worker classes that give you fine-grained control as well as extensibility. Using them, you can completely customize workflow and activity type registration and set factories for creating implementation objects. These workers are GenericWorkflowWorker and GenericActivityWorker.

The GenericWorkflowWorker can be configured with a factory for creating workflow definition factories. The workflow definition factory is responsible for creating instances of
the workflow implementation and for providing configuration settings such as registration options. Under normal circumstances, you should use the WorkflowWorker class directly. It will automatically create and configure implementation of the factories provided in the framework, POJOWorkflowDefinitionFactoryFactory and POJOWorkflowDefinitionFactory. The factory requires that the workflow implementation class must have a no argument constructor. This constructor is used to create instances of the workflow object at run time. The factory looks at the annotations you used on the workflow interface and implementation to create appropriate registration and execution options.

You may provide your own implementation of the factories by implementing WorkflowDefinitionFactory, WorkflowDefinitionFactoryFactory, and WorkflowDefinition. The WorkflowDefinition class is used by the worker class to dispatch decision tasks and signals. By implementing these base classes, you can completely customize the factory and the dispatch of requests to the workflow implementation. For example, you can use these extensibility points to provide a custom programming model for writing workflows, for instance, based on your own annotations or generating it from WSDL instead of the code first approach used by the framework. In order to use your custom factories, you will have to use the GenericWorkflowWorker class. For more details about these classes, see the AWS SDK for Java documentation.

Similarly, GenericActivityWorker allows you to provide a custom activity implementation factory. By implementing the ActivityImplementationFactory and ActivityImplementation classes you can completely control activity instantiation as well as customize registration and execution options. For more details of these classes, see the AWS SDK for Java documentation.

### Execution Context

**Topics**

- Decision Context (p. 83)
- Activity Execution Context (p. 85)

The framework provides an ambient context to workflow and activity implementations. This context is specific to the task being processed and provides some utilities that you can use in your implementation. A context object is created every time a new task is processed by the worker.

### Decision Context

When a decision task is executed, the framework provides the context to workflow implementation through the DecisionContext class. DecisionContext provides context-sensitive information like workflow execution run Id and clock and timer functionality.

#### Accessing DecisionContext in Workflow Implementation

You can access the DecisionContext in your workflow implementation using the DecisionContextProviderImpl class. Alternatively, you can inject the context in a field or property of your workflow implementation using Spring as shown in the Testability and Dependency Injection section.

```java
DecisionContextProvider contextProvider = new DecisionContextProviderImpl();
DecisionContext context = contextProvider.getDecisionContext();
```
Creating a Clock and Timer

The DecisionContext contains a property of type WorkflowClock that provides timer and clock functionality. Since the workflow logic needs to be deterministic, you should not directly use the system clock in your workflow implementation. The currentTimeMills method on the WorkflowClock returns the time of the start event of the decision being processed. This ensures that you get the same time value during replay, hence, making your workflow logic deterministic.

WorkflowClock also has a createTimer method which returns a Promise object that becomes ready after the specified interval. You can use this value as a parameter to other asynchronous methods to delay their execution by the specified period of time. This way you can effectively schedule an asynchronous method or activity for execution at a later time.

The example in the following listing demonstrates how to periodically call an activity.

```java
@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
    defaultTaskStartToCloseTimeoutSeconds = 10)
public interface PeriodicWorkflow {

    @Execute(version = "1.0")
    void periodicWorkflow();
}

@Activities(version = "1.0")
@ActivityRegistrationOptions(defaultTaskScheduleToStartTimeoutSeconds = 300,
    defaultTaskStartToCloseTimeoutSeconds = 3600)
public interface PeriodicActivity {
    void activity1();
}

public class PeriodicWorkflowImpl implements PeriodicWorkflow {

    private DecisionContextProvider contextProvider = new DecisionContextProviderImpl();
    private WorkflowClock clock = contextProvider.getDecisionContext().getWorkflowClock();

    @Override
    public void periodicWorkflow() {
        callPeriodicActivity(0);
    }

    @Asynchronous
    private void callPeriodicActivity(int count, Promise<?>... waitFor) {
        if (count == 100) {
            return;
        }
        PeriodicActivityClient client = new PeriodicActivityClientImpl();
        // call activity
        Promise<Void> activityCompletion = client.activity1();
        Promise<Void> timer = clock.createTimer(3600);
        // Repeat the activity either after 1 hour or after previous activity run
        // if it takes longer than 1 hour
        callPeriodicActivity(count + 1, timer, activityCompletion);
    }
}
```
public class PeriodicActivityImpl implements PeriodicActivity
{
    @Override
    public void activity1()
    {
        ...
    }
}

In the above listing, the callPeriodicActivity asynchronous method calls activity1 and then creates a timer using the current AsyncDecisionContext. It passes the returned Promise as an argument to a recursive call to itself. This recursive call waits until the timer fires (1 hour in this example) before executing.

### Activity Execution Context

Just as the DecisionContext provides context information when a decision task is being processed, ActivityExecutionContext provides similar context information when an activity task is being processed. This context is available to your activity code through ActivityExecutionContextProviderImpl class.

```java
ActivityExecutionContextProvider provider = new ActivityExecutionContextProviderImpl();
ActivityExecutionContext aec = provider.getActivityExecutionContext();
```

Using ActivityExecutionContext, you can perform the following:

#### Heartbeat a Long Running Activity

If the activity is long running, it must periodically report its progress to Amazon SWF to let it know that the task is still making progress. In the absence of such a heartbeat, the task may timeout if a task heartbeat timeout was set at activity type registration or while scheduling the activity. In order to send a heartbeat, you can use the recordActivityHeartbeat method on ActivityExecutionContext. Heartbeat also provides a mechanism for canceling ongoing activities. See the Error Handling (p. 101) section for more details and an example.

#### Get Details of the Activity Task

If you want, you can get all the details of the activity task that were passed by Amazon SWF when the executor got the task. This includes information regarding the inputs to the task, task type, task token, etc. If you want to implement an activity that is manually completed—for example, by a human action—then you must use the ActivityExecutionContext to retrieve the task token and pass it to the process that will eventually complete the activity task. See the section on Manually Completing Activities (p. 75) for more details.

#### Get the Amazon SWF Client Object that is Being Used by the Executor

The Amazon SWF client object being used by the executor can be retrieved by calling getService method on ActivityExecutionContext. This is useful if you want to make a direct call to the Amazon SWF service.

### Child Workflow Executions

In the examples so far, we have started workflow execution directly from an application. However, a workflow execution may be started from within a workflow by calling the workflow entry point method.
on the generated client. When a workflow execution is started from the context of another workflow execution, it is called a child workflow execution. This allows you to refactor complex workflows into smaller units and potentially share them across different workflows. For example, you can create a payment processing workflow and call it from an order processing workflow.

Semantically, the child workflow execution behaves the same as a standalone workflow except for the following differences:

1. When the parent workflow terminates due to an explicit action by the user—for example, by calling the TerminateWorkflowExecution Amazon SWF API, or it is terminated due to a timeout—then the fate of the child workflow execution will be determined by a child policy. You can set this child policy to terminate, cancel, or abandon (keep running) child workflow executions.
2. The output of the child workflow (return value of the entry point method) can be used by the parent workflow execution just like the Promise<T> returned by an asynchronous method. This is different from standalone executions where the application must get the output by using Amazon SWF APIs.

In the following example, the OrderProcessor workflow creates a PaymentProcessor child workflow:

```java
@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
                               defaultTaskStartToCloseTimeoutSeconds = 10)
public interface OrderProcessor {
    @Execute(version = "1.0")
    void processOrder(Order order);
}

public class OrderProcessorImpl implements OrderProcessor {
    PaymentProcessorClientFactory factory
        = new PaymentProcessorClientFactoryImpl();

    @Override
    public void processOrder(Order order) {
        float amount = order.getAmount();
        CardInfo cardInfo = order.getCardInfo();

        PaymentProcessorClient childWorkflowClient = factory.getClient();
        childWorkflowClient.processPayment(amount, cardInfo);
    }
}

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
                               defaultTaskStartToCloseTimeoutSeconds = 10)
public interface PaymentProcessor {
    @Execute(version = "1.0")
    void processPayment(float amount, CardInfo cardInfo);
}

public class PaymentProcessorImpl implements PaymentProcessor {
    PaymentActivitiesClient activitiesClient = new PaymentActivitiesClientImpl();

    @Override
    public void processPayment(float amount, CardInfo cardInfo) {
        Promise<PaymentType> payType = activitiesClient.getPaymentType(cardInfo);
        switch(payType.get()) {
            case Visa:
                activitiesClient.processVisa(amount, cardInfo);
                break;
        }
    }
```
Continuous Workflows

In some use cases, you may need a workflow that executes forever or runs for a long duration, for example, a workflow that monitors the health of a server fleet.

Note
Since Amazon SWF keeps the entire history of a workflow execution, the history will keep growing over time. The framework retrieves this history from Amazon SWF when it performs a replay, and this will become expensive if the history size is too large. In such long running or continuous workflows, you should periodically close the current execution and start a new one to continue processing.

This is a logical continuation of the workflow execution. The generated self client can be used for this purpose. In your workflow implementation, simply call the @Execute method on the self client. Once the current execution completes, the framework will start a new execution using the same workflow Id.

You can also continue the execution by calling the continueAsNewOnCompletion method on the GenericWorkflowClient that you can retrieve from the current DecisionContext. For example, the following workflow implementation sets a timer to fire after a day and calls its own entry point to start a new execution.

```java
public class ContinueAsNewWorkflowImpl implements ContinueAsNewWorkflow {
    private DecisionContextProvider contextProvider
        = new DecisionContextProviderImpl();

    private ContinueAsNewWorkflowSelfClient selfClient
        = new ContinueAsNewWorkflowSelfClientImpl();

    private WorkflowClock clock
        = contextProvider.getDecisionContext().getWorkflowClock();

    @Override
    public void startWorkflow() {
        Promise<Void> timer = clock.createTimer(86400);  // 1 day
        continueAsNew(timer);
    }
}
```
When a workflow recursively calls itself, the framework will close the current workflow when all pending tasks have completed and start a new workflow execution. Note that as long as there are pending tasks, the current workflow execution will not close. The new execution will not automatically inherit any history or data from the original execution; if you want to carry over some state to the new execution, then you must pass it explicitly as input.

Setting Task Priority

By default, tasks on a task list are delivered based upon their arrival time: tasks that are scheduled first are generally run first, as far as possible. By setting an optional task priority, you can give priority to certain tasks: Amazon SWF will attempt to deliver higher-priority tasks on a task list before those with lower priority.

You can set task priorities for both workflows and activities. A workflow’s task priority doesn’t affect the priority of any activity tasks it schedules, nor does it affect any child workflows it starts. The default priority for an activity or workflow is set (either by you or by Amazon SWF) during registration, and the registered task priority is always used unless it is overridden while scheduling the activity or starting a workflow execution.

Task priority values can range from "-2147483648" to "2147483647", with higher numbers indicating higher priority. If you don’t set the task priority for an activity or workflow, it will be assigned a priority of zero ("0").

Topics
• Setting Task Priority for Workflows (p. 88)
• Setting Task Priority for Activities (p. 89)

Setting Task Priority for Workflows

You can set the task priority for a workflow when you register it or start it. The task priority that is set when the workflow type is registered is used as the default for any workflow executions of that type, unless it is overridden when starting the workflow execution.

To register a workflow type with a default task priority, set the defaultTaskPriority option in WorkflowRegistrationOptions when declaring it:

```java
@Workflow
@WorkflowRegistrationOptions(
    defaultTaskPriority = 10,
    defaultTaskStartToCloseTimeoutSeconds = 240)
public interface PriorityWorkflow
{
    @Execute(version = "1.0")
    void startWorkflow(int a);
}
```

You can also set the taskPriority for a workflow when you start it, overriding the registered (default) task priority.
Setting Task Priority for Activities

You can set the task priority for an activity either when registering it or when scheduling it. The task priority that is set when registering an activity type is used as the default priority when the activity is run, unless it is overridden when scheduling the activity.

To register an activity type with a default task priority, set the `defaultTaskPriority` option in `ActivityRegistrationOptions` when declaring it:

```java
@Activities(version = "1.0")
@ActivityRegistrationOptions(
    defaultTaskPriority = 10,
    defaultTaskStartToCloseTimeoutSeconds = 120)
public interface ImportantActivities {
    int doSomethingImportant();
}
```

You can also set the `taskPriority` for an activity when you schedule it, overriding the registered (default) task priority.

```java
ActivitySchedulingOptions activityOptions = new ActivitySchedulingOptions.withTaskPriority(10);
ImportantActivitiesClient activityClient = new ImportantActivitiesClientImpl();
activityClient.doSomethingImportant(activityOptions);
```

DataConverters

When your workflow implementation calls a remote activity, the inputs passed to it and the result of executing the activity must be serialized so they can be sent over the wire. The framework uses the DataConverter class for this purpose. This is an abstract class that you can implement to provide your own serializer. A default Jackson serializer-based implementation, JsonDataConverter, is provided in the framework. For more details, see the AWS SDK for Java documentation. Refer to the Jackson JSON Processor documentation for details about how Jackson performs serialization as well as Jackson annotations that can be used to influence it. The wire format used is considered part of the contract. Hence, you can specify a DataConverter on your activities and workflow interfaces by setting the DataConverter property of the @Activities and @Workflow annotations.

The framework will create objects of the DataConverter type you specified on @Activities annotation to serialize the inputs to the activity and to deserialize its result. Similarly, objects of the
DataConverter type you specify on @Workflow annotation will be used to serialize parameters you pass to the workflow, and in the case of child workflow, to deserialize the result. In addition to inputs, the framework also passes additional data to Amazon SWF—for example, exception details—the workflow serializer will be used for serializing this data as well.

You can also provide an instance of the DataConverter if you don't want the framework to automatically create it. The generated clients have constructor overloads that take a DataConverter.

If you don't specify a DataConverter type and don't pass a DataConverter object, the JsonDataConverter will be used by default.

### Passing Data to Asynchronous Methods

#### Topics
- Passing Collections and Maps to Asynchronous Methods (p. 90)
- Settable<T> (p. 91)
- @NoWait (p. 92)
- Promise<Void> (p. 92)
- AndPromise and OrPromise (p. 92)

The use of Promise<T> has been explained in previous sections. Some advanced use cases of Promise<T> are discussed here.

#### Passing Collections and Maps to Asynchronous Methods

The framework supports passing arrays, collections, and maps as Promise types to asynchronous methods. For example, an asynchronous method may take Promise<ArrayList<String>> as an argument as shown in the following listing.

```java
@Asynchronous
public void printList(Promise<List<String>> list) {
    for (String s: list.get()) {
        activityClient.printActivity(s);
    }
}
```

Semantically, this behaves as any other Promise typed parameter and the asynchronous method will wait until the collection becomes available before executing. If the members of a collection are Promise objects, then you can make the framework wait for all members to become ready as shown in the following snippet. This will make the asynchronous method wait on each member of the collection to become available.

```java
@Asynchronous
public void printList(@Wait List<Promise<String>> list) {
    for (Promise<String> s: list) {
        activityClient.printActivity(s);
    }
}
```

Note that the @Wait annotation must be used on the parameter to indicate that it contains Promise objects.
Note also that the activity `printActivity` takes a `String` argument but the matching method in the generated client takes a `Promise<String>`. We are calling the method on the client and not invoking the activity method directly.

### Settable<T>

`Settable<T>` is a derived type of `Promise<T>` that provides a set method that allows you to manually set the value of a `Promise`. For example, the following workflow waits for a signal to be received by waiting on a `Settable<?>`, which is set in the signal method:

```java
public class MyWorkflowImpl implements MyWorkflow{
    final Settable<String> result = new Settable<String>();

    // Signal
    @Override
    public void manualProcessCompletedSignal(String data) {
        result.set(data);
    }

    @Asynchronous
    public Promise<String> done(Settable<String> result){
        return result;
    }
}
```

A `Settable<?>>` can also be chained to another promise at a time. You can use `AndPromise` and `OrPromise` to group promises. You can unchain a chained `Settable` by calling the `unchain()` method on it. When chained, the `Settable<?>>` automatically becomes ready when the promise that it is chained to becomes ready. Chaining is especially useful when you want to use a promise returned from within the scope of a `doTry()` in other parts of your program. Since `TryCatchFinally` is used as a nested class, you can't declare a `Promise<>` in the parent’s scope and set it in `doTry()`. This is because Java requires variables to be declared in parent scope and used in nested classes to be marked final. For example:

```java
@Asynchronous
public Promise<String> chain(final Promise<String> input) {
    final Settable<String> result = new Settable<String>();

    new TryFinally() {
        @Override
        protected void doTry() throws Throwable {
            Promise<String> resultToChain = activity1(input);
            activity2(resultToChain);

            // Chain the promise to Settable
            result.chain(resultToChain);
        }

        @Override
        protected void doFinally() throws Throwable {
            if (result.isReady()) { // Was a result returned before the exception?
                // Do cleanup here
                return;
            }
        }
    };
```
A `Settable` can be chained to one promise at a time. You can unchain a chained `Settable` by calling the `unchain()` method on it.

@NoWait

When you pass a `Promise` to an asynchronous method, by default, the framework will wait for the `Promise` (s) to become ready before executing the method (except for collection types). You may override this behavior by using the `@NoWait` annotation on parameters in the declaration of the asynchronous method. This is useful if you are passing in `Settable<T>`, which will be set by the asynchronous method itself.

**Promise<Void>**

Dependencies in asynchronous methods are implemented by passing the `Promise` returned by one method as an argument to another. However, there may be cases where you want to return `void` from a method, but still want other asynchronous methods to execute after its completion. In such cases, you can use `Promise<Void>` as the return type of the method. The `Promise` class provides a static `Void` method that you can use to create a `Promise<Void>` object. This `Promise` will become ready when the asynchronous method finishes execution. You can pass this `Promise` to another asynchronous method just like any other `Promise` object. If you are using `Settable<Void>`, then call the set method on it with null to make it ready.

**AndPromise and OrPromise**

`AndPromise` and `OrPromise` allow you to group multiple `Promise<>` objects into a single logical promise. An `AndPromise` becomes ready when all promises used to construct it become ready. An `OrPromise` becomes ready when any promise in the collection of promises used to construct it becomes ready. You can call `getValues()` on `AndPromise` and `OrPromise` to retrieve the list of values of the constituent promises.

**Testability and Dependency Injection**

**Topics**

- [Spring Integration](#)
- [JUnit Integration](#)

The framework is designed to be Inversion of Control (IoC) friendly. Activity and workflow implementations as well as the framework supplied workers and context objects can be configured and instantiated using containers like Spring. Out of the box, the framework provides integration with the Spring Framework. In addition, integration with JUnit has been provided for unit testing workflow and activity implementations.

**Spring Integration**

The com.amazonaws.services.simpleworkflow.flow.spring package contains classes that make it easy to use the Spring framework in your applications. These include a custom Scope and Spring-aware activity and workflow workers: `WorkflowScope`, `SpringWorkflowWorker` and `SpringActivityWorker`. 
These classes allow you to configure your workflow and activity implementations as well as the workers entirely through Spring.

**WorkflowScope**

WorkflowScope is a custom Spring Scope implementation provided by the framework. This scope allows you to create objects in the Spring container whose lifetime is scoped to that of a decision task. The beans in this scope are instantiated every time a new decision task is received by the worker. You should use this scope for workflow implementation beans and any other beans it depends on. The Spring-provided singleton and prototype scopes should not be used for workflow implementation beans because the framework requires that a new bean be created for each decision task. Failure to do so will result in unexpected behavior.

The following example shows a snippet of Spring configuration that registers the WorkflowScope and then uses it for configuring a workflow implementation bean and an activity client bean.

```xml
<!-- register AWS Flow Framework for Java WorkflowScope -->
<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
  <property name="scopes">
    <map>
      <entry key="workflow">
        <bean class="com.amazonaws.services.simpleworkflow.flow.spring.WorkflowScope" />
      </entry>
    </map>
  </property>
</bean>

<!-- activities client -->
<bean id="activitiesClient" class="aws.flow.sample.MyActivitiesClientImpl" scope="workflow">
</bean>

<!-- workflow implementation -->
<bean id="workflowImpl" class="aws.flow.sample.MyWorkflowImpl" scope="workflow">
  <property name="client" ref="activitiesClient"/>
  <aop:scoped-proxy proxy-target-class="false"/>
</bean>
```

The line of configuration: `<aop:scoped-proxy proxy-target-class="false" />`, used in the configuration of the workflowImpl bean, is required because the WorkflowScope doesn't support proxying using CGLIB. You should use this configuration for any bean in the WorkflowScope that is wired to another bean in a different scope. In this case, the workflowImpl bean needs to be wired to a workflow worker bean in singleton scope (see complete example below).

You can learn more about using custom scopes in the Spring Framework documentation.

**Spring-Aware Workers**

When using Spring, you should use the Spring-aware worker classes provided by the framework: SpringWorkflowWorker and SpringActivityWorker. These workers can be injected in your application using Spring as shown in the next example. The Spring-aware workers implement Spring's SmartLifecycle interface and, by default, automatically start polling for tasks when the Spring context is initialized. You can turn off this functionality by setting the disableAutoStartup property of the worker to true.

The following example shows how to configure a decider. This example uses MyActivities and MyWorkflow interfaces (not shown here) and corresponding implementations, MyActivitiesImpl and MyWorkflowImpl. The generated client interfaces and implementations are MyWorkflowClient/MyWorkflowClientImpl and MyActivitiesClient/MyActivitiesClientImpl (also not shown here).
The activities client is injected in the workflow implementation using Spring's auto wire feature:

```java
public class MyWorkflowImpl implements MyWorkflow {
    @Autowired
    public MyActivitiesClient client;

    @Override
    public void start() {
        client.activity1();
    }
}
```

The Spring configuration for the decider is as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
    <!-- register custom workflow scope -->
    <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
        <property name="scopes">
            <map>
                <entry key="workflow">
                    <bean class="com.amazonaws.services.simpleworkflow.flow.spring.WorkflowScope"/>
                </entry>
            </map>
        </property>
    </bean>

    <context:annotation-config/>

    <bean id="accesskeys" class="com.amazonaws.auth.BasicAWSCredentials">
        <constructor-arg value="{AWS.Access.ID}"/>
        <constructor-arg value="{AWS.Secret.Key}"/>
    </bean>

    <bean id="clientConfiguration" class="com.amazonaws.ClientConfiguration">
        <property name="socketTimeout" value="70000"/>
    </bean>

    <!-- Amazon SWF client -->
    <bean id="swfClient" class="com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient">
        <constructor-arg ref="accesskeys"/>
        <constructor-arg ref="clientConfiguration"/>
        <property name="endpoint" value="{service.url}"/>
    </bean>

    <!-- activities client -->
    <bean id="activitiesClient" class="aws.flow.sample.MyActivitiesClientImpl" scope="workflow"/>

    <!-- workflow implementation -->
    <bean id="workflowImpl" class="aws.flow.sample.MyWorkflowImpl" scope="workflow">
        <property name="client" ref="activitiesClient"/>
        <aop:scoped-proxy proxy-target-class="false"/>
    </bean>
</beans>
```
Since the `SpringWorkflowWorker` is fully configured in Spring and automatically starts polling when the Spring context is initialized, the host process for the decider is simple:

```java
public class WorkflowHost {
    public static void main(String[] args){
        ApplicationContext context = new FileSystemXmlApplicationContext("resources/spring/WorkflowHostBean.xml");
        System.out.println("Workflow worker started");
    }
}
```

Similarly, the activity worker can be configured as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    <!-- register custom scope -->
    <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
        <property name="scopes">
            <map>
                <entry key="workflow">
                    <bean class="com.amazonaws.services.simpleworkflow.flow.spring.WorkflowScope"/>
                </entry>
            </map>
        </property>
    </bean>
    <bean id="accesskeys" class="com.amazonaws.auth.BasicAWSCredentials">
        <constructor-arg value="{AWS.Access.ID}"/>
        <constructor-arg value="{AWS.Secret.Key}"/>
    </bean>
    <bean id="clientConfiguration" class="com.amazonaws.ClientConfiguration">
        <property name="socketTimeout" value="70000"/>
    </bean>
</beans>
```
The activity worker host process is similar to the decider:

```java
public class ActivityHost {
    public static void main(String[] args) {
        ApplicationContext context = new FileSystemXmlApplicationContext("resources/spring/ActivityHostBean.xml");
        System.out.println("Activity worker started");
    }
}
```

### Injecting Decision Context

If your workflow implementation depends on the context objects, then you can easily inject them through Spring as well. The framework automatically registers context-related beans in the Spring container. For example, in the following snippet, the various context objects have been auto wired. No other Spring configuration of the context objects is required.

```java
public class MyWorkflowImpl implements MyWorkflow {
    @Autowired
    public MyActivitiesClient client;
    @Autowired
    public WorkflowClock clock;
    @Autowired
    public DecisionContext dcContext;
    @Autowired
    public GenericActivityClient activityClient;
    @Autowired
    public GenericWorkflowClient workflowClient;
    @Autowired
    public WorkflowContext wfContext;
    @Override
    public void start() {
```
If you want to configure the context objects in the workflow implementation through Spring XML configuration, then use the bean names declared in the WorkflowScopeBeanNames class in the com.amazonaws.services.simpleworkflow.flow.spring package. For example:

```xml
<!-- workflow implementation -->
<bean id="workflowImpl" class="asadj.spring.test.MyWorkflowImpl" scope="workflow">
    <property name="client" ref="activitiesClient"/>
    <property name="clock" ref="workflowClock"/>
    <property name="activityClient" ref="genericActivityClient"/>
    <property name="dcContext" ref="decisionContext"/>
    <property name="workflowClient" ref="genericWorkflowClient"/>
    <property name="wfContext" ref="workflowContext"/>
    <aop:scoped-proxy proxy-target-class="false"/>
</bean>
```

Alternatively, you may inject a DecisionContextProvider in the workflow implementation bean and use it to create the context. This can be useful if you want to provide custom implementations of the provider and context.

### Injecting Resources in Activities

You can instantiate and configure activity implementations using an Inversion of Control (IoC) container and easily inject resources like database connections by declaring them as properties of the activity implementation class. Such resources will typically be scoped as singletons. Note that activity implementations are called by the activity worker on multiple threads. Therefore, access to shared resources must be synchronized.

### JUnit Integration

The framework provides JUnit extensions as well as test implementations of the context objects, such as a test clock, that you can use to write and run unit tests with JUnit. With these extensions, you can test your workflow implementation locally inline.

#### Writing a Simple Unit Test

In order to write tests for your workflow, use the WorkflowTest class in the com.amazonaws.services.simpleworkflow.flow.junit package. This class is a framework-specific JUnit MethodRule implementation and runs your workflow code locally, calling activities inline as opposed to going through Amazon SWF. This gives you the flexibility to run your tests as frequently as you desire without incurring any charges.

In order to use this class, simply declare a field of type WorkflowTest and annotate it with the @Rule annotation. Before running your tests, create a new WorkflowTest object and add your activity and workflow implementations to it. You can then use the generated workflow client factory to create a client and start an execution of the workflow. The framework also provides a custom JUnit runner, FlowBlockJUnit4ClassRunner, that you must use for your workflow tests. For example:

```java
@RunWith(FlowBlockJUnit4ClassRunner.class)
public class BookingWorkflowTest {
    @Rule
    public WorkflowTest workflowTest = new WorkflowTest();

    // Rest of your test methods...
}
```
List<String> trace;

private BookingWorkflowClientFactory workflowFactory
    = new BookingWorkflowClientFactoryImpl();

@Before
public void setUp() throws Exception {
    trace = new ArrayList<String>();
    // Register activity implementation to be used during test run
    BookingActivities activities = new BookingActivitiesImpl(trace);
    workflowTest.addActivitiesImplementation(activities);
    workflowTest.addWorkflowImplementationType(BookingWorkflowImpl.class);
}

@After
public void tearDown() throws Exception {
    trace = null;
}

@Test
public void testReserveBoth() {
    BookingWorkflowClient workflow = workflowFactory.getClient();
    Promise<Void> booked = workflow.makeBooking(123, 345, true, true);
    List<String> expected = new ArrayList<String>();
    expected.add("reserveCar-123");
    expected.add("reserveAirline-123");
    expected.add("sendConfirmation-345");
    AsyncAssert.assertEqualsWaitFor("invalid booking", expected, trace, booked);
}

You can also specify a separate task list for each activity implementation that you add to WorkflowTest. For example, if you have a workflow implementation that schedules activities in host-specific task lists, then you can register the activity in the task list of each host:

```java
for (int i = 0; i < 10; i++) {
    String hostname = "host" + i;
    workflowTest.addActivitiesImplementation(hostname,
        new ImageProcessingActivities(hostname));
}
```

Notice that the code in the @Test is asynchronous. Therefore, you should use the asynchronous workflow client to start an execution. In order to verify the results of your test, an AsyncAssert help class is also provided. This class allows you to wait for promises to become ready before verifying results. In this example, we wait for the result of the workflow execution to be ready before verifying the test output.

If you are using Spring, then the SpringWorkflowTest class can be used instead of the WorkflowTest class. SpringWorkflowTest provides properties that you can use to configure activity and workflow implementations easily through Spring configuration. Just like the Spring-aware workers, you should use the WorkflowScope to configure workflow implementation beans. This ensures that a new workflow implementation bean is created for every decision task. Make sure to configure these beans with the scoped-proxy proxy-target-class setting set to false. See the Spring Integration section for more details. The example Spring configuration shown in the Spring Integration section can be changed to test the workflow using SpringWorkflowTest:

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
```
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<!-- register custom workflow scope -->
<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
  <property name="scopes">
    <map>
      <entry key="workflow">
        <bean
class="com.amazonaws.services.simpleworkflow.flow.spring.WorkflowScope" />
      </entry>
    </map>
  </property>
</bean>
<context:annotation-config />
<bean id="accesskeys" class="com.amazonaws.auth.BasicAWSCredentials">
  <constructor-arg value="{AWS.Access.ID}" />
  <constructor-arg value="{AWS.Secret.Key}" />
</bean>
<bean id="clientConfiguration" class="com.amazonaws.ClientConfiguration">
  <property name="socketTimeout" value="70000" />
</bean>

<!-- Amazon SWF client -->
<bean id="swfClient" class="com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient">
  <constructor-arg ref="accesskeys" />
  <constructor-arg ref="clientConfiguration" />
  <property name="endpoint" value="{service.url}" />
</bean>

<!-- activities client -->
<bean id="activitiesClient" class="aws.flow.sample.MyActivitiesClientImpl" scope="workflow">
</bean>

<!-- workflow implementation -->
<bean id="workflowImpl" class="aws.flow.sample.MyWorkflowImpl" scope="workflow">
  <property name="client" ref="activitiesClient" />
  <aop:scoped-proxy proxy-target-class="false" />
</bean>

<!-- WorkflowTest -->
<bean id="workflowTest" class="com.amazonaws.services.simpleworkflow.flow.junit.spring.SpringWorkflowTest">
  <property name="workflowImplementations">
    <list>
      <ref bean="workflowImpl" />
    </list>
  </property>
  <property name="taskListActivitiesImplementationMap">
    <map>
      <entry>
        <key>
          <value>list1</value>
        </key>
        <ref bean="activitiesImplHost1" />
      </entry>
    </map>
  </property>
</bean>
Mocking Activity Implementations

You may use the real activity implementations during testing, but if you want to unit test just the workflow logic, you should mock the activities. This can be done by providing a mock implementation of the activities interface to the WorkflowTest class. For example:

```java
@RunWith(FlowBlockJUnit4ClassRunner.class)
public class BookingWorkflowTest {

    @Rule
    public WorkflowTest workflowTest = new WorkflowTest();

    List<String> trace;

    private BookingWorkflowClientFactory workflowFactory = new BookingWorkflowClientFactoryImpl();

    @Before
    public void setUp() throws Exception {
        trace = new ArrayList<String>();
        // Create and register mock activity implementation to be used during test run
        BookingActivities activities = new BookingActivities() {
            @Override
            public void sendConfirmationActivity(int customerId) {
                trace.add("sendConfirmation-" + customerId);
            }

            @Override
            public void reserveCar(int requestId) {
                trace.add("reserveCar-" + requestId);
            }

            @Override
            public void reserveAirline(int requestId) {
                trace.add("reserveAirline-" + requestId);
            }
        };
        workflowTest.addActivitiesImplementation(activities);
        workflowTest.addWorkflowImplementationType(BookingWorkflowImpl.class);
    }

    @After
    public void tearDown() throws Exception {
        trace = null;
    }

    @Test
    public void testReserveBoth() {
        BookingWorkflowClient workflow = workflowFactory.getClient();
        Promise<Void> booked = workflow.makeBooking(123, 345, true, true);
        List<String> expected = new ArrayList<String>();
        expected.add("reserveCar-123");
        expected.add("reserveAirline-123");
        expected.add("sendConfirmation-345");
        AsyncAssert.assertEqualsWaitFor("invalid booking", expected, trace, booked);
    }
}
```

Alternatively, you can provide a mock implementation of the activities client and inject that into your workflow implementation.
Test Context Objects

If your workflow implementation depends on the framework context objects—for example, the DecisionContext—you don't have to do anything special to test such workflows. When a test is run through WorkflowTest, it automatically injects test context objects. When your workflow implementation accesses the context objects—for example, using DecisionContextProviderImpl—it will get the test implementation. You can manipulate these test context objects in your test code (@Test method) to create interesting test cases. For example, if your workflow creates a timer, you can make the timer fire by calling the clockAdvanceSeconds method on the WorkflowTest class to move the clock forward in time. You can also accelerate the clock to make timers fire earlier than they normally would using the ClockAccelerationCoefficient property on WorkflowTest. For example, if your workflow creates a timer for one hour, you can set the ClockAccelerationCoefficient to 60 to make the timer fire in one minute. By default, ClockAccelerationCoefficient is set to 1.

For more details about the com.amazonaws.services.simpleworkflow.flow.test and com.amazonaws.services.simpleworkflow.flow.junit packages, see the AWS SDK for Java documentation.

Error Handling

Topics

- TryCatchFinally Semantics (p. 102)
- Cancellation (p. 103)
- Nested TryCatchFinally (p. 106)

The try/catch/finally construct in Java makes it simple to handle errors and is used ubiquitously. It allows you to associate error handlers to a block of code. Internally, this works by stuffing additional metadata about the error handlers on the call stack. When an exception is thrown, the runtime looks at the call stack for an associated error handler and invokes it; and if no appropriate error handler is found, it propagates the exception up the call chain.

This works well for synchronous code, but handling errors in asynchronous and distributed programs poses additional challenges. Since an asynchronous call returns immediately, the caller isn't on the call stack when the asynchronous code executes. This means that unhandled exceptions in the asynchronous code can't be handled by the caller in the usual way. Typically, exceptions that originate in asynchronous code are handled by passing error state to a callback that is passed to the asynchronous method. Alternatively, if a Future<? > is being used, it reports an error when you try to access it. This is less than ideal because the code that receives the exception (the callback or code that uses the Future<? >) doesn't have the context of the original call and may not be able to handle the exception adequately. Furthermore, in a distributed asynchronous system, with components running concurrently, more than one error may occur simultaneously. These errors could be of different types and severities and need to be handled appropriately.

Cleaning up resource after an asynchronous call is also difficult. Unlike synchronous code, you can't use try/catch/finally in the calling code to clean up resources since work initiated in the try block may still be ongoing when the finally block executes.

The framework provides a mechanism that makes error handling in distributed asynchronous code similar to, and almost as simple as, Java's try/catch/finally.

```java
ImageProcessingActivitiesClient activitiesClient = new ImageProcessingActivitiesClientImpl();

public void createThumbnail(final String webPageUrl) {  
```
new TryCatchFinally() {

    @Override
    protected void doTry() throws Throwable {
        List<String> images = getImageUrls(webPageUrl);
        for (String image: images) {
            Promise<String> localImage = activitiesClient.downloadImage(image);
            Promise<String> thumbnailFile = activitiesClient.createThumbnail(localImage);
            activitiesClient.uploadImage(thumbnailFile);
        }
    }

    @Override
    protected void doCatch(Throwable e) throws Throwable {
        // Handle exception and rethrow failures
        LoggingActivitiesClient logClient = new LoggingActivitiesClientImpl();
        logClient.reportError(e);
        throw new RuntimeException("Failed to process images", e);
    }

    @Override
    protected void doFinally() throws Throwable {
        activitiesClient.cleanUp();
    }
};

The TryCatchFinally class and its variants, TryFinally and TryCatch, work similar to Java's try/catch/finally. Using it, you can associate exception handlers to blocks of workflow code that may execute as asynchronous and remote tasks. The doTry() method is logically equivalent to the try block. The framework automatically executes the code in doTry(). A list of Promise objects can be passed to the constructor of TryCatchFinally. The doTry method will be executed when all Promise objects passed in to the constructor become ready. If an exception is raised by code that was asynchronously invoked from within doTry(), any pending work in doTry() is canceled and doCatch() is called to handle the exception. For instance, in the listing above, if downloadImage throws an exception, then createThumbnail and uploadImage will be canceled. Finally, dofinally() is called when all asynchronous work is done (completed, failed, or canceled). It can be used for resource cleanup. You can also nest these classes to suit your needs.

When an exception is reported in doCatch(), the framework provides a complete logical call stack that includes asynchronous and remote calls. This can be helpful when debugging, especially if you have asynchronous methods calling other asynchronous methods. For example, an exception from downloadImage will produce an exception like this:

```
RuntimeException: error downloading image
  at downloadImage(Main.java:35)
  at ---continuation---.(repeated:1)
  at errorHandlingAsync$1.doTry(Main.java:24)
  at ---continuation---.(repeated:1)
...```

TryCatchFinally Semantics

The execution of an AWS Flow Framework for Java program can be visualized as a tree of concurrently executing branches. A call to an asynchronous method, an activity, and TryCatchFinally itself creates
a new branch in this tree of execution. For example, the image processing workflow can be viewed as the tree shown in the following figure.

An error in one branch of execution will cause the unwinding of that branch, just as an exception causes the unwinding of the call stack in a Java program. The unwinding keeps moving up the execution branch until either the error is handled or the root of the tree is reached, in which case the workflow execution is terminated.

The framework reports errors that happen while processing tasks as exceptions. It associates the exception handlers (doCatch() methods) defined in TryCatchFinally with all tasks that are created by the code in the corresponding doTry(). If a task fails—for example, due to a timeout or an unhandled exception—then the appropriate exception will be raised and the corresponding doCatch() will be invoked to handle it. To accomplish this, the framework works in tandem with Amazon SWF to propagate remote errors and resurrects them as exceptions in the caller's context.

Cancellation

When an exception occurs in synchronous code, the control jumps directly to the catch block, skipping over any remaining code in the try block. For example:

```java
try {
    a();
    b();
    c();
} catch (Exception e) {
    e.printStackTrace();
}
```

In this code, if b() throws an exception, then c() is never invoked. Compare that to a workflow:

```java
new TryCatch() {
    @Override
    protected void doTry() throws Throwable {
        activityA();
        activityB();
        activityC();
    }

    @Override
    protected void doCatch(Throwable e) throws Throwable {
        e.printStackTrace();
    }
};
```

In this case, calls to activityA, activityB, and activityC all return successfully and result in the creation of three tasks that will be executed asynchronously. Let's say at a later time that the task for activityB results in an error. This error is recorded in the history by Amazon SWF. In order to handle this, the framework will first try to cancel all other tasks that originated within the scope of the same
doTry(); in this case, activityA and activityC. When all such tasks complete (cancel, fail, or successfully complete), the appropriate doCatch() method will be invoked to handle the error.

Unlike the synchronous example, where c() was never executed, activityC was invoked and a task was scheduled for execution; hence, the framework will make an attempt to cancel it, but there is no guarantee that it will be canceled. Cancellation can't be guaranteed because the activity may have already completed, may ignore the cancellation request, or may fail due to an error. However, the framework does provide the guarantee that doCatch() is called only after all tasks started from the corresponding doTry() have completed. It also guarantees that doFinally() is called only after all tasks started from the doTry() and doCatch() have completed. If, for instance, the activities in the above example depend on each other, say activityB depends on activityA and activityC on activityB, then the cancellation of activityC will be immediate because it isn't scheduled in Amazon SWF until activityB completes:

```java
new TryCatch() {
    @Override
    protected void doTry() throws Throwable {
        Promise<Void> a = activityA();
        Promise<Void> b = activityB(a);
        activityC(b);
    }

    @Override
    protected void doCatch(Throwable e) throws Throwable {
        e.printStackTrace();
    }
};
```

### Activity Heartbeat

The AWS Flow Framework for Java's cooperative cancellation mechanism allows in-flight activity tasks to be canceled gracefully. When cancellation is triggered, tasks that blocked or are waiting to be assigned to a worker are automatically canceled. If, however, a task is already assigned to a worker, the framework will request the activity to cancel. Your activity implementation must explicitly handle such cancellation requests. This is done by reporting heartbeat of your activity.

Reporting heartbeat allows the activity implementation to report the progress of an ongoing activity task, which is useful for monitoring, and it lets the activity check for cancellation requests. The recordActivityHeartbeat method will throw a CancellationException if a cancellation has been requested. The activity implementation can catch this exception and act on the cancellation request, or it can ignore the request by swallowing the exception. In order to honor the cancellation request, the activity should perform the desired clean up, if any, and then rethrow CancellationException. When this exception is thrown from an activity implementation, the framework records that the activity task has been completed in canceled state.

The following example shows an activity that downloads and processes images. It heartbeats after processing each image, and if cancellation is requested, it cleans up and rethrows the exception to acknowledge cancellation.

```java
@Override
public void processImages(List<String> urls) {
    int imageCounter = 0;
    for (String url: urls) {
        imageCounter++;
        Image image = download(url);
        process(image);
        try {
            ActivityExecutionContext context
                = contextProvider.getActivityExecutionContext();
        } catch (CancellationException e) {
            // Process clean up...
            throw e;
        }
    }
}
```
Report activity heartbeat isn’t required, but it is recommended if your activity is long running or may be performing expensive operations that you wish to be canceled under error conditions. You should call `heartbeatActivityTask` periodically from the activity implementation.

If the activity times out, the `ActivityTaskTimedOutException` will be thrown and `getDetails` on the exception object will return the data passed to the last successful call to `heartbeatActivityTask` for the corresponding activity task. The workflow implementation may use this information to determine how much progress was made before the activity task was timed out.

**Note**
It isn’t a good practice to heartbeat too frequently because Amazon SWF may throttle heartbeat requests. See the [Amazon Simple Workflow Service Developer Guide](https://docs.aws.amazon.com/swf/latest/developerguide/swf-limits.html) for limits placed by Amazon SWF.

### Explicitly Canceling a Task

Besides error conditions, there are other cases where you may explicitly cancel a task. For example, an activity to process payments using a credit card may need to be canceled if the user cancels the order. The framework allows you to explicitly cancel tasks created in the scope of a `TryCatchFinally`. In the following example, the payment task is canceled if a signal is received while the payment was being processed.

```java
public class OrderProcessorImpl implements OrderProcessor {
    private PaymentProcessorClientFactory factory
        = new PaymentProcessorClientFactoryImpl();
    boolean processingPayment = false;
    private TryCatchFinally paymentTask = null;

    @Override
    public void processOrder(int orderId, final float amount) {
        paymentTask = new TryCatchFinally() {
            @Override
            protected void doTry() throws Throwable {
                processingPayment = true;
                PaymentProcessorClient paymentClient = factory.getClient();
                paymentClient.processPayment(amount);
            }

            @Override
            protected void doCatch(Throwable e) throws Throwable {
                if (e instanceof CancellationException) {
                    paymentClient.log("Payment canceled.");
                } else {
                    throw e;
                }
            }

            @Override
            protected void doFinally() throws Throwable {
                processingPayment = false;
            }
        };
```
Receiving Notification of Canceled Tasks

When a task is completed in canceled state, the framework informs the workflow logic by throwing a CancellationException. When an activity completes in canceled state, a record is made in the history and the framework calls the appropriate doCatch() with a CancellationException. As shown in the previous example, when the payment processing task is canceled, the workflow receives a CancellationException.

An unhandled CancellationException is propagated up the execution branch just like any other exception. However, the doCatch() method will receive the CancellationException only if there is no other exception in the scope; other exceptions are prioritized higher than cancellation.

Nested TryCatchFinally

You may nest TryCatchFinally's to suit your needs. Since each TryCatchFinally creates a new branch in the execution tree, you can create nested scopes. Exceptions in the parent scope will cause cancellation attempts of all tasks initiated by nested TryCatchFinally's within it. However, exceptions in a nested TryCatchFinally don't automatically propagate to the parent. If you wish to propagate an exception from a nested TryCatchFinally to its containing TryCatchFinally, you should rethrow the exception in doCatch(). In other words, only unhandled exceptions are bubbled up, just like Java's try/catch. If you cancel a nested TryCatchFinally by calling the cancel method, the nested TryCatchFinally will be canceled but the containing TryCatchFinally will not automatically get canceled.
Retry Failed Activities

Activities sometimes fail for ephemeral reasons, such as a temporary loss of connectivity. At another time, the activity might succeed, so the appropriate way to handle activity failure is often to retry the activity, perhaps multiple times.

There are a variety of strategies for retrying activities; the best one depends on the details of your workflow. The strategies fall into three basic categories:

- The retry-until-success strategy simply keeps retrying the activity until it completes.
- The exponential retry strategy increases the time interval between retry attempts exponentially until the activity completes or the process reaches a specified stopping point, such as a maximum number of attempts.
- The custom retry strategy decides whether or how to retry the activity after each failed attempt.

The following sections describe how to implement these strategies. The example workflow workers all use a single activity, unreliableActivity, which randomly does one of following:

- Completes immediately
- Fails intentionally by exceeding the timeout value
- Fails intentionally by throwing IllegalStateException

Retry-Until-Success Strategy

The simplest retry strategy is to keep retrying the activity each time it fails until it eventually succeeds. The basic pattern is:

1. Implement a nested TryCatch or TryCatchFinally class in your workflow’s entry point method.
2. Execute the activity in doTry
3. If the activity fails, the framework calls doCatch, which runs the entry point method again.
4. Repeat Steps 2 - 3 until the activity completes successfully.

The following workflow implements the retry-until-success strategy. The workflow interface is implemented in RetryActivityRecipeWorkflow and has one method, runUnreliableActivityTillSuccess, which is the workflow’s entry point. The workflow worker is implemented in RetryActivityRecipeWorkflowImpl, as follows:

```java
public class RetryActivityRecipeWorkflowImpl
```
implements RetryActivityRecipeWorkflow {

    @Override
    public void runUnreliableActivityTillSuccess() {
        final Settable<Boolean> retryActivity = new Settable<Boolean>();

        new TryCatch() {
            @Override
            protected void doTry() throws Throwable {
                Promise<Void> activityRanSuccessfully = client.unreliableActivity();
                setRetryActivityToFalse(activityRanSuccessfully, retryActivity);
            }

            @Override
            protected void doCatch(Throwable e) throws Throwable {
                retryActivity.set(true);
            }
        };
        restartRunUnreliableActivityTillSuccess(retryActivity);
    }

    @Asynchronous
    private void setRetryActivityToFalse(
        Promise<Void> activityRanSuccessfully,
        @NoWait Settable<Boolean> retryActivity)
    {
        retryActivity.set(false);
    }

    @Asynchronous
    private void restartRunUnreliableActivityTillSuccess(
        Settable<Boolean> retryActivity)
    {
        if (retryActivity.get()) {
            runUnreliableActivityTillSuccess();
        }
    }
}

The workflow works as follows:

1. runUnreliableActivityTillSuccess creates a Settable<Boolean> object named retryActivity which is used to indicate whether the activity failed and should be retried. Settable<T> is derived from Promise<T> and works much the same way, but you set a Settable<T> object's value manually.

2. runUnreliableActivityTillSuccess implements an anonymous nested TryCatch class to handle any exceptions that are thrown by the unreliableActivity activity. For more discussion of how to handle exceptions thrown by asynchronous code, see Error Handling (p. 101).

3. doTry executes the unreliableActivity activity, which returns a Promise<Void> object named activityRanSuccessfully.

4. doTry calls the asynchronous setRetryActivityToFalse method, which has two parameters:
   - activityRanSuccessfully takes the Promise<Void> object returned by the unreliableActivity activity.
   - retryActivity takes the retryActivity object.

   If unreliableActivity completes, activityRanSuccessfully becomes ready and setRetryActivityToFalse sets retryActivity to false. Otherwise, activityRanSuccessfully never becomes ready and setRetryActivityToFalse doesn't execute.

5. If unreliableActivity throws an exception, the framework calls doCatch and passes it the exception object. doCatch sets retryActivity to true.
6. `runUnreliableActivityTillSuccess` calls the asynchronous `restartRunUnreliableActivityTillSuccess` method and passes it the `retryActivity` object. Because `retryActivity` is a `Promise<T>` type, `restartRunUnreliableActivityTillSuccess` defers execution until `retryActivity` is ready, which occurs after `TryCatch` completes.

7. When `retryActivity` is ready, `restartRunUnreliableActivityTillSuccess` extracts the value.
   - If the value is `false`, the retry succeeded. `restartRunUnreliableActivityTillSuccess` doesn’t do anything and the retry sequence terminates.
   - If the value is `true`, the retry failed. `restartRunUnreliableActivityTillSuccess` calls `runUnreliableActivityTillSuccess` to execute the activity again.

8. Steps 1 - 7 repeat until `unreliableActivity` completes.

**Note**

`doCatch` doesn’t handle the exception; it simply sets the `retryActivity` object to true to indicate that the activity failed. The retry is handled by the asynchronous `restartRunUnreliableActivityTillSuccess` method, which defers execution until `TryCatch` completes. The reason for this approach is that, if you retry an activity in `doCatch`, you can’t cancel it. Retrying the activity in `restartRunUnreliableActivityTillSuccess` allows you to execute cancellable activities.

### Exponential Retry Strategy

With the exponential retry strategy, the framework executes a failed activity again after a specified period of time, N seconds. If that attempt fails the framework executes the activity again after 2N seconds, and then 4N seconds and so on. Because the wait time can get quite large, you typically stop the retry attempts at some point rather than continuing indefinitely.

The framework provides three ways to implement an exponential retry strategy:

- The `@ExponentialRetry` annotation is the simplest approach, but you must set the retry configuration options at compile time.
- The `RetryDecorator` class allows you to set retry configuration at run time and change it as needed.
- The `AsyncRetryingExecutor` class allows you to set retry configuration at run time and change it as needed. In addition, the framework calls a user-implemented `AsyncRunnable.run` method to run each retry attempt.

All approaches support the following configuration options, where time values are in seconds:

- The initial retry wait time.
- The back-off coefficient, which is used to compute the retry intervals, as follows:

  \[
  \text{retryInterval} = \text{initialRetryIntervalSeconds} \times \text{Math.pow(\text{backoffCoefficient}, \text{numberOfTries} - 2)}
  \]

  The default value is 2.0.
- The maximum number of retry attempts. The default value is unlimited.
- The maximum retry interval. The default value is unlimited.
- The expiration time. Retry attempts stop when the total duration of the process exceeds this value. The default value is unlimited.
- The exceptions that will trigger the retry process. By default, every exception triggers the retry process.
- The exceptions that will not trigger a retry attempt. By default, no exceptions are excluded.
The following sections describe the various ways that you can implement an exponential retry strategy.

**Exponential Retry with @ExponentialRetry**

The simplest way to implement an exponential retry strategy for an activity is to apply an `@ExponentialRetry` annotation to the activity in the interface definition. If the activity fails, the framework handles the retry process automatically, based on the specified option values. The basic pattern is:

1. Apply `@ExponentialRetry` to the appropriate activities and specify the retry configuration.
2. If an annotated activity fails, the framework automatically retries the activity according to the configuration specified by the annotation's arguments.

The `ExponentialRetryAnnotationWorkflow` workflow worker implements the exponential retry strategy by using an `@ExponentialRetry` annotation. It uses an `unreliableActivity` activity whose interface definition is implemented in `ExponentialRetryAnnotationActivities`, as follows:

```java
@Activities(version = "1.0")
@ActivityRegistrationOptions(
    defaultTaskScheduleToStartTimeoutSeconds = 30,
    defaultTaskStartToCloseTimeoutSeconds = 30)
public interface ExponentialRetryAnnotationActivities {
    @ExponentialRetry(
        initialRetryIntervalSeconds = 5,
        maximumAttempts = 5,
        exceptionsToRetry = IllegalStateException.class)
    public void unreliableActivity();
}
```

The `@ExponentialRetry` options specify the following strategy:

- Retry only if the activity throws `IllegalStateException`.
- Use an initial wait time of 5 seconds.
- No more than 5 retry attempts.

The workflow interface is implemented in `RetryWorkflow` and has one method, `process`, which is the workflow's entry point. The workflow worker is implemented in `ExponentialRetryAnnotationWorkflowImpl`, as follows:

```java
public class ExponentialRetryAnnotationWorkflowImpl implements RetryWorkflow {
    public void process() {
        handleUnreliableActivity();
    }

    public void handleUnreliableActivity() {
        client.unreliableActivity();
    }
}
```

The workflow works as follows:

1. `process` runs the synchronous `handleUnreliableActivity` method.
2. `handleUnreliableActivity` executes the `unreliableActivity` activity.

If the activity fails by throwing `IllegalStateException`, the framework automatically runs the retry strategy specified in `ExponentialRetryAnnotationActivities`. 
Exponential Retry with the RetryDecorator Class

@ExponentialRetry is simple to use. However, the configuration is static and set at compile time, so the framework uses the same retry strategy every time the activity fails. You can implement a more flexible exponential retry strategy by using the RetryDecorator class, which allows you to specify the configuration at run time and change it as needed. The basic pattern is:

1. Create and configure an ExponentialRetryPolicy object that specifies the retry configuration.
2. Create a RetryDecorator object and pass the ExponentialRetryPolicy object from Step 1 to the constructor.
3. Apply the decorator object to the activity by passing the activity client's class name to the RetryDecorator object's decorate method.
4. Execute the activity.

If the activity fails, the framework retries the activity according to the ExponentialRetryPolicy object's configuration. You can change the retry configuration as needed by modifying this object.

Note
The @ExponentialRetry annotation and the RetryDecorator class are mutually exclusive. You can't use RetryDecorator to dynamically override a retry policy specified by an @ExponentialRetry annotation.

The following workflow implementation shows how to use the RetryDecorator class to implement an exponential retry strategy. It uses an unreliableActivity activity that doesn't have an @ExponentialRetry annotation. The workflow interface is implemented in RetryWorkflow and has one method, process, which is the workflow's entry point. The workflow worker is implemented in DecoratorRetryWorkflowImpl, as follows:

```java
public class DecoratorRetryWorkflowImpl implements RetryWorkflow {
    ...
    public void process() {
        long initialRetryIntervalSeconds = 5;
        int maximumAttempts = 5;
        ExponentialRetryPolicy retryPolicy = new ExponentialRetryPolicy(
            initialRetryIntervalSeconds).withMaximumAttempts(maximumAttempts);
        Decorator retryDecorator = new RetryDecorator(retryPolicy);
        client = retryDecorator.decorate(RetryActivitiesClient.class, client);
        handleUnreliableActivity();
    }

    public void handleUnreliableActivity() {
        client.unreliableActivity();
    }
}
```

The workflow works as follows:

1. process creates and configures an ExponentialRetryPolicy object by:
   - Passing the initial retry interval to the constructor.
   - Calling the object's withMaximumAttempts method to set the maximum number of attempts to 5. ExponentialRetryPolicy exposes other with objects that you can use to specify other configuration options.
2. process creates a RetryDecorator object named retryDecorator and passes the ExponentialRetryPolicy object from Step 1 to the constructor.
3. process applies the decorator to the activity by calling the retryDecorator.decorate method and passing it the activity client's class name.
4. `handleUnreliableActivity` executes the activity.

If the activity fails, the framework retries it according to the configuration specified in Step 1.

**Note**
Several of the `ExponentialRetryPolicy` class's `with` methods have a corresponding `set` method that you can call to modify the corresponding configuration option at any time: `setBackoffCoefficient`, `setMaximumAttempts`, `setMaximumRetryIntervalSeconds`, and `setMaximumRetryExpirationIntervalSeconds`.

### Exponential Retry with the AsyncRetryingExecutor Class

The `RetryDecorator` class provides more flexibility in configuring the retry process than `@ExponentialRetry`, but the framework still runs the retry attempts automatically, based on the `ExponentialRetryPolicy` object's current configuration. A more flexible approach is to use the `AsyncRetryingExecutor` class. In addition to allowing you to configure the retry process at run time, the framework calls a user-implemented `AsyncRunnable.run` method to run each retry attempt instead of simply executing the activity.

The basic pattern is:

1. Create and configure an `ExponentialRetryPolicy` object to specify the retry configuration.
2. Create an `AsyncRetryingExecutor` object, and pass it the `ExponentialRetryPolicy` object and an instance of the workflow clock.
3. Implement an anonymous nested `TryCatch` or `TryCatchFinally` class.
4. Implement an anonymous `AsyncRunnable` class and override the `run` method to implement custom code for running the activity.
5. Override `doTry` to call the `AsyncRetryingExecutor` object's `execute` method and pass it the `AsyncRunnable` class from Step 4. The `AsyncRetryingExecutor` object calls `AsyncRunnable.run` to run the activity.
6. If the activity fails, the `AsyncRetryingExecutor` object calls the `AsyncRunnable.run` method again, according to the retry policy specified in Step 1.

The following workflow shows how to use the `AsyncRetryingExecutor` class to implement an exponential retry strategy. It uses the same `unreliableActivity` activity as the `DecoratorRetryWorkflow` workflow discussed earlier. The workflow interface is implemented in `RetryWorkflow` and has one method, `process`, which is the workflow's entry point. The workflow worker is implemented in `AsyncExecutorRetryWorkflowImpl`, as follows:

```java
public class AsyncExecutorRetryWorkflowImpl implements RetryWorkflow {
    private final RetryActivitiesClient client = new RetryActivitiesClientImpl();
    private final DecisionContextProvider contextProvider = new DecisionContextProviderImpl();
    private final WorkflowClock clock = contextProvider.getDecisionContext().getWorkflowClock();

    public void process() {
        long initialRetryIntervalSeconds = 5;
        int maximumAttempts = 5;
        handleUnreliableActivity(initialRetryIntervalSeconds, maximumAttempts);
    }

    public void handleUnreliableActivity(long initialRetryIntervalSeconds, int maximumAttempts) {
        ExponentialRetryPolicy retryPolicy = new ExponentialRetryPolicy(initialRetryIntervalSeconds).withMaximumAttempts(maximumAttempts);
        final AsyncExecutor executor = new AsyncRetryingExecutor(retryPolicy, clock);
```

API Version 2012-01-25
The workflow works as follows:

1. process calls the handleUnreliableActivity method and passes it the configuration settings.
2. handleUnreliableActivity uses the configuration settings from Step 1 to create an ExponentialRetryPolicy object, retryPolicy.
3. handleUnreliableActivity creates an AsyncRetryExecutor object, executor, and passes the ExponentialRetryPolicy object from Step 2 and an instance of the workflow clock to the constructor.
4. handleUnreliableActivity implements an anonymous nested TryCatch class and overrides the doTry and doCatch methods to run the retry attempts and handle any exceptions.
5. doTry creates an anonymous AsyncRunnable class and overrides the run method to implement custom code to execute unreliableActivity. For simplicity, run just executes the activity, but you can implement more sophisticated approaches as appropriate.
6. doTry calls executor.execute and passes it the AsyncRunnable object. execute calls the AsyncRunnable object's run method to run the activity.
7. If the activity fails, executor calls run again, according to the retryPolicy object configuration.

For more discussion of how to use the TryCatch class to handle errors, see AWS Flow Framework for Java Exceptions (p. 132).

### Custom Retry Strategy

The most flexible approach to retrying failed activities is a custom strategy, which recursively calls an asynchronous method that runs the retry attempt, much like the retry-until-success strategy. However, instead of simply running the activity again, you implement custom logic that decides whether and how to run each successive retry attempt. The basic pattern is:

1. Create a Settable<T> status object, which is used to indicate whether the activity failed.
2. Implement a nested TryCatch or TryCatchFinally class.
3. doTry executes the activity.
4. If the activity fails, doCatch sets the status object to indicate that the activity failed.
5. Call an asynchronous failure handling method and pass it the status object. The method defers execution until TryCatch or TryCatchFinally completes.
6. The failure handling method decides whether to retry the activity, and if so, when.

The following workflow shows how to implement a custom retry strategy. It uses the same unreliableActivity activity as the DecoratorRetryWorkflow and AsyncExecutorRetryWorkflow workflows. The workflow interface is implemented in
RetryWorkflow and has one method, process, which is the workflow's entry point. The workflow worker is implemented in CustomLogicRetryWorkflowImpl, as follows:

```java
public class CustomLogicRetryWorkflowImpl implements RetryWorkflow {
    ...
    public void process() {
        callActivityWithRetry();
    }
    @Asynchronous
    public void callActivityWithRetry() {
        final Settable<Throwable> failure = new Settable<Throwable>();
        new TryCatchFinally() {
            protected void doTry() throws Throwable {
                client.unreliableActivity();
            }
            protected void doCatch(Throwable e) {
                failure.set(e);
            }
            protected void doFinally() throws Throwable {
                if (!failure.isReady()) {
                    failure.set(null);
                }
            }
        }
        retryOnFailure(failure);
    }
    @Asynchronous
    private void retryOnFailure(Promise<Throwable> failureP) {
        Throwable failure = failureP.get();
        if (failure != null && shouldRetry(failure)) {
            callActivityWithRetry();
        }
    }
    protected Boolean shouldRetry(Throwable e) {
        // custom logic to decide to retry the activity or not
        return true;
    }
}
```

The workflow works as follows:

1. process calls the asynchronous callActivityWithRetry method.
2. callActivityWithRetry creates a Settable<Throwable> object named failure which is used to indicate whether the activity has failed. Settable<T> is derived from Promise<T> and works much the same way, but you set a Settable<T> object's value manually.
3. callActivityWithRetry implements an anonymous nested TryCatchFinally class to handle any exceptions that are thrown by unreliableActivity. For more discussion of how to handle exceptions thrown by asynchronous code, see AWS Flow Framework for Java Exceptions (p. 132).
4. doTry executes unreliableActivity.
5. If unreliableActivity throws an exception, the framework calls doCatch and passes it the exception object. doCatch sets failure to the exception object, which indicates that the activity failed and puts the object in a ready state.
6. doFinally checks whether failure is ready, which will be true only if failure was set by doCatch.
   - If failure is ready, doFinally does nothing.
   - If failure isn't ready, the activity completed and doFinally sets failure to null.
7. callActivityWithRetry calls the asynchronous retryOnFailure method and passes it failure. Because failure is a Settable<T> type, callActivityWithRetry defers execution until failure is ready, which occurs after TryCatchFinally completes.
8. `retryOnFailure` gets the value from failure.
   - If failure is set to null, the retry attempt was successful. `retryOnFailure` does nothing, which terminates the retry process.
   - If failure is set to an exception object and `shouldRetry` returns true, `retryOnFailure` calls `callActivityWithRetry` to retry the activity.

   `shouldRetry` implements custom logic to decide whether to retry a failed activity. For simplicity, `shouldRetry` always returns true and `retryOnFailure` executes the activity immediately, but you can implement more sophisticated logic as needed.

9. Steps 2–8 repeat until `unreliableActivity` completes or `shouldRetry` decides to stop the process.

**Note**

`doCatch` doesn't handle the retry process; it simply sets failure to indicate that the activity failed. The retry process is handled by the asynchronous `retryOnFailure` method, which defers execution until `TryCatch` completes. The reason for this approach is that, if you retry an activity in `doCatch`, you can't cancel it. Retrying the activity in `retryOnFailure` allows you to execute cancellable activities.

---

**Daemon Tasks**

The AWS Flow Framework for Java allows the marking of certain tasks as daemon. This allows you to create tasks that do some background work that should get canceled when all other work is done. For example, a health monitoring task should be canceled when the rest of the workflow is done. You can accomplish this by setting the `daemon` flag on an asynchronous method or an instance of `TryCatchFinally`. In the following example, the asynchronous method `monitorHealth()` is marked as daemon.

```java
public class MyWorkflowImpl implements MyWorkflow {
    MyActivitiesClient activitiesClient = new MyActivitiesClientImpl();

    @Override
    public void startMyWF(int a, String b) {
        activitiesClient.doUsefulWorkActivity();
        monitorHealth();
    }

    @Asynchronous(daemon=true)
    void monitorHealth(Promise<?>... waitFor) {
        activitiesClient.monitoringActivity();
    }
}
```

In the above example, when `doUsefulWorkActivity` completes, `monitoringHealth` will be automatically canceled. This will in turn cancel the whole execution branch rooted at this asynchronous method. The semantics of cancellation are the same as in `TryCatchFinally`. Similarly, you can mark a `TryCatchFinally` daemon by passing a Boolean flag to the constructor.

```java
public class MyWorkflowImpl implements MyWorkflow {
    MyActivitiesClient activitiesClient = new MyActivitiesClientImpl();

    @Override
    public void startMyWF(int a, String b) {
```
A daemon task started within a `TryCatchFinally` is scoped to the context it is created in—that is, it will be scoped to either the `doTry()`, `doCatch()`, or `doFinally()` methods. For example, in the following example the `startMonitoring` asynchronous method is marked daemon and called from `doTry()`. The task created for it will be canceled as soon as the other tasks (`doUsefulWorkActivity` in this case) started within `doTry()` are complete.

```java
public class MyWorkflowImpl implements MyWorkflow {
    MyActivitiesClient activitiesClient = new MyActivitiesClientImpl();

    @Override
    public void startMyWF(int a, String b) {
        new TryFinally() {
            @Override
            protected void doTry() throws Throwable {
                activitiesClient.doUsefulWorkActivity();
                startMonitoring();
            }
            @Override
            protected void doFinally() throws Throwable {
                // Clean up
            }
        };
        @Asynchronous(daemon = true)
        void startMonitoring()
            activitiesClient.monitoringActivity();
    }
}
```

AWS Flow Framework for Java Replay Behavior

This topic discusses examples of replay behavior, using the examples in the What is the AWS Flow Framework for Java? (p. 1) section. Both synchronous (p. 116) and asynchronous (p. 118) scenarios are discussed.

**Example 1: Synchronous Replay**

For an example of how replay works in a synchronous workflow, modify the `HelloWorldWorkflow` (p. 14) workflow and activity implementations by adding `println` calls within their respective implementations, as follows:

```java
public class GreeterWorkflowImpl implements GreeterWorkflow {
```
Example 1: Synchronous Replay

```java
public void greet() {
    System.out.println("greet executes");
    Promise<String> name = operations.getName();
    System.out.println("client.getName returns");
    Promise<String> greeting = operations.getGreeting(name);
    System.out.println("client.greeting returns");
    operations.say(greeting);
    System.out.println("client.say returns");
}
**************
public class GreeterActivitiesImpl implements GreeterActivities {
    public String getName() {
        System.out.println("activity.getName completes");
        return "World";
    }
    public String getGreeting(String name) {
        System.out.println("activity.getGreeting completes");
        return "Hello " + name + "!";
    }
    public void say(String what) {
        System.out.println(what);
    }
}
```

For details about the code, see HelloWorldWorkflow Application (p. 14). The following is an edited version of the output, with comments that indicate the start of each replay episode.

```
//Episode 1
greet executes
client.getName returns
client.greeting returns
client.say returns

activity.getName completes
//Episode 2
//Episode 3
activity.getGreeting completes
//Episode 4
Hello World! //say completes
```

The replay process for this example works as follows:

- The first episode schedules the `getName` activity task, which has no dependencies.
- The second episode schedules the `getGreeting` activity task, which depends on `getName`.
- The third episode schedules the `say` activity task, which depends on `getGreeting`.
The final episode schedules no additional tasks and finds no uncompleted activities, which terminates the workflow execution.

**Note**
The three activities client methods are called once for each episode. However, only one of those calls results in an activity task, so each task is performed only once.

## Example 2: Asynchronous Replay

Similarly to the synchronous replay example (p. 116), you can modify `HelloWorldWorkflowAsync Application (p. 25)` to see how an asynchronous replay works. It produces the following output:

```java
//Episode 1
greet executes
client.name returns
workflow.getGreeting returns
client.say returns
activity.getName completes
//Episode 2
greet executes
client.name returns
workflow.getGreeting returns
client.say returns
workflow.getGreeting completes

Hello World! //say completes
//Episode 3
greet executes
client.name returns
workflow.getGreeting returns
client.say returns
workflow.getGreeting completes
```

HelloWorldAsync uses three replay episodes because there are only two activities. The `getGreeting` activity was replaced by the `getGreeting` asynchronous workflow method, which doesn't initiate a replay episode when it completes.

The first episode doesn't call `getGreeting`, because it depends on the completion of the `name` activity. However, after `getName` completes, replay calls `getGreeting` once for each succeeding episode.

## See Also

- [AWS Flow Framework Basic Concepts: Distributed Execution (p. 38)]
Under the Hood

Topics
- Task (p. 119)
- Order of Execution (p. 119)
- Workflow Execution (p. 120)
- Nondeterminism (p. 122)

Task

The underlying primitive that the AWS Flow Framework for Java uses to manage the execution of asynchronous code is the Task class. An object of type Task represents work that has to be performed asynchronously. When you call an asynchronous method, the framework creates a Task to execute the code in that method and puts it in a list for execution at a later time. Similarly, when you invoke an Activity, a Task is created for it. The method call returns after this, usually returning a Promise<T> as the future result of the call.

The Task class is public and may be used directly. For example, we can rewrite the Hello World example to use a Task instead of an asynchronous method.

```java
@Override
public void startHelloWorld(){
    final Promise<String> greeting = client.getName();
    new Task(greeting) {
        @Override
        protected void doExecute() throws Throwable {
            client.printGreeting("Hello " + greeting.get() +"!");
        }
    };
}
```

The framework calls the doExecute() method when all the Promises passed to the constructor of the Task become ready. For more details about the Task class, see the AWS Java SDK documentation.

The framework also includes a class called Functor which represents a Task that is also a Promise<T>. The Functor object becomes ready when the Task completes. In the following example, a Functor is created to get the greeting message:

```java
Promise<String> greeting = new Functor<String>() {
    @Override
    protected Promise<String> doExecute() throws Throwable {
        return client.getGreeting();
    }
};
client.printGreeting(greeting);
```

Order of Execution

Tasks become eligible for execution only when all Promise<T> typed parameters, passed to the corresponding asynchronous method or activity, become ready. A Task that is ready for execution is logically moved to a ready queue. In other words, it is scheduled for execution. The worker class executes
the task by invoking the code that you wrote in the body of the asynchronous method, or by scheduling an activity task in Amazon Simple Workflow Service (AWS) in case of an activity method.

As tasks execute and produce results, they cause other tasks to become ready and the execution of the program keeps moving forward. The way the framework executes tasks is important to understand the order in which your asynchronous code executes. Code that appears sequentially in your program may not actually execute in that order.

```java
Promise<String> name = getUserName();
printHelloName(name);
printHelloWorld();
System.out.println("Hello, Amazon!");

@Asynchronous
private Promise<String> getUserName(){
    return Promise.asPromise("Bob");
}
@Asynchronous
private void printHelloName(Promise<String> name){
    System.out.println("Hello, " + name.get() + ",!");
}
@Asynchronous
private void printHelloWorld(){
    System.out.println("Hello, World!");
}
```

The code in the listing above will print the following:

Hello, Amazon!
Hello, World!
Hello, Bob

This may not be what you expected but can be easily explained by thinking through how the tasks for the asynchronous methods were executed:

1. The call to `getUserName` creates a Task. Let's call it Task1. Since `getUserName` doesn't take any parameters, Task1 is immediately put in the ready queue.
2. Next, the call to `printHelloName` creates a Task that needs to wait for the result of `getUserName`. Let's call it Task2. Since the requisite value isn't ready yet, Task2 is put in the wait list.
3. Then a task for `printHelloWorld` is created and added to the ready queue. Let's call it Task3.
4. The `println` statement then prints "Hello, Amazon!" to the console.
5. At this point, Task1 and Task3 are in the ready queue and Task2 is in the wait list.
6. The worker executes Task1, and its result makes Task2 ready. Task2 gets added to ready queue behind Task3.
7. Task3 and Task2 are then executed in that order.

The execution of activities follows the same pattern. When you call a method on the activity client, it creates a Task that, upon execution, schedules an activity in Amazon SWF.

The framework relies on features like code generation and dynamic proxies to inject the logic for converting method calls to activity invocations and asynchronous tasks in your program.

**Workflow Execution**

The execution of the workflow implementation is also managed by the worker class. When you call a method on the workflow client, it calls Amazon SWF to create a workflow instance. The tasks in Amazon
SWF should not be confused with the tasks in the framework. A task in Amazon SWF is either an activity task or a decision task. The execution of activity tasks is simple. The activity worker class receives activity tasks from Amazon SWF, invokes the appropriate activity method in your implementation, and returns the result to Amazon SWF.

The execution of decision tasks is more involved. The workflow worker receives decision tasks from Amazon SWF. A decision task is effectively a request asking the workflow logic what to do next. The first decision task is generated for a workflow instance when it is started through the workflow client. Upon receiving this decision task, the framework starts executing the code in the workflow method annotated with `@Execute`. This method executes the coordination logic that schedules activities. When the state of the workflow instance changes—for example, when an activity completes—further decision tasks get scheduled. At this point, the workflow logic can decide to take an action based on the result of the activity; for example, it may decide to schedule another activity.

The framework hides all these details from the developer by seamlessly translating decision tasks to the workflow logic. From a developer's point of view, the code looks just like a regular program. Under the covers, the framework maps it to calls to Amazon SWF and decision tasks using the history maintained by Amazon SWF. When a decision task arrives, the framework replays the program execution plugging in the results of the activities completed so far. Asynchronous methods and activities that were waiting for these results get unblocked, and the program execution moves forward.

The execution of the example image processing workflow and the corresponding history is shown in the following table.

### Execution of thumbnail workflow

<table>
<thead>
<tr>
<th>Workflow program execution</th>
<th>History maintained by Amazon SWF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial execution</strong></td>
<td></td>
</tr>
<tr>
<td>1. Dispatch loop</td>
<td>1. Workflow instance started, id=&quot;1&quot;</td>
</tr>
<tr>
<td>2. <code>getImgaeUrls</code></td>
<td>2. downloadImage scheduled</td>
</tr>
<tr>
<td>3. downloadImage</td>
<td></td>
</tr>
<tr>
<td>4. createThumbnail (task in wait queue)</td>
<td>3. downloadImage completed, return=&quot;foo&quot;</td>
</tr>
<tr>
<td>5. uploadImage (task in wait queue)</td>
<td>4. createThumbnail scheduled</td>
</tr>
<tr>
<td>6. &lt;next iteration of the loop&gt;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Replay</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Dispatch loop</td>
<td>1. Workflow instance started, id=&quot;1&quot;</td>
</tr>
<tr>
<td>2. <code>getImgaeUrls</code></td>
<td>2. downloadImage scheduled</td>
</tr>
<tr>
<td>3. <code>downloadImage image path=&quot;foo&quot;</code></td>
<td>3. downloadImage completed, return=&quot;foo&quot;</td>
</tr>
<tr>
<td>4. createThumbnail</td>
<td>4. createThumbnail scheduled</td>
</tr>
<tr>
<td>5. uploadImage (task in wait queue)</td>
<td></td>
</tr>
<tr>
<td>6. &lt;next iteration of the loop&gt;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Replay</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Dispatch loop</td>
<td>1. Workflow instance started, id=&quot;1&quot;</td>
</tr>
<tr>
<td>2. <code>getImgaeUrls</code></td>
<td>2. downloadImage scheduled</td>
</tr>
<tr>
<td>3. <code>downloadImage image path=&quot;foo&quot;</code></td>
<td>3. downloadImage completed, return=&quot;foo&quot;</td>
</tr>
<tr>
<td>4. <code>createThumbnail thumbnail path=&quot;bar&quot;</code></td>
<td>4. createThumbnail scheduled</td>
</tr>
<tr>
<td>5. uploadImage</td>
<td>5. createThumbnail completed, return=&quot;bar&quot;</td>
</tr>
<tr>
<td>6. &lt;next iteration of the loop&gt;</td>
<td>6. uploadImage scheduled</td>
</tr>
</tbody>
</table>

API Version 2012-01-25

121
When a call to `processImage` is made, the framework creates a new workflow instance in Amazon SWF. This is a durable record of the workflow instance being started. The program executes until the call to the `downloadImage` activity, which asks Amazon SWF to schedule an activity. The workflow executes further and creates tasks for subsequent activities, but they can't be executed until the `downloadImage` activity completes; hence, this episode of replay ends. Amazon SWF dispatches the task for `downloadImage` activity for execution, and once it is completed, a record is made in the history along with the result. The workflow is now ready to move forward and a decision task is generated by Amazon SWF. The framework receives the decision task and replays the workflow plugging in the result of the downloaded image as recorded in the history. This unblocks the task for `createThumbnail`, and the execution of the program continues farther by scheduling the `createThumbnail` activity task in Amazon SWF. The same process repeats for `uploadImage`. The execution of the program continues this way until the workflow has processed all images and there are no pending tasks. Since no execution state is stored locally, each decision task may be potentially executed on a different machine. This allows you to easily write programs that are fault tolerant and easily scalable.

**Nondeterminism**

Since the framework relies on replay, it is important that the orchestration code (all workflow code with the exception of activity implementations) be deterministic. For example, the control flow in your program should not depend on a random number or the current time. Since these things will change between invocations, the replay may not follow the same path through the orchestration logic. This will lead to unexpected results or errors. The framework provides a `WorkflowClock` that you can use to get the current time in a deterministic way. See the section on Execution Context (p. 83) for more details.

**Note**

Incorrect Spring wiring of workflow implementation objects can also lead to nondeterminism. Workflow implementation beans as well as beans that they depend on must be in the workflow scope (`WorkflowScope`). For example, wiring a workflow implementation bean to a bean that keeps state and is in the global context will result in unexpected behavior. See the Spring Integration (p. 92) section for more details.
Troubleshooting and Debugging Tips

Topics

- Compilation Errors (p. 123)
- Unknown Resource Fault (p. 123)
- Exceptions When Calling get() on a Promise (p. 123)
- Non Deterministic Workflows (p. 124)
- Problems Due to Versioning (p. 124)
- Troubleshooting and Debugging a Workflow Execution (p. 124)
- Lost Tasks (p. 125)

This section describes some common pitfalls that you might run into while developing workflows using AWS Flow Framework for Java. It also provides some tips to help you diagnose and debug problems.

Compilation Errors

If you are using the AspectJ compile time weaving option, you may run into compile time errors in which the compiler isn’t able to find the generated client classes for your workflow and activities. The likely cause of such compilation errors is that the AspectJ builder ignored the generated clients during compilation. You can fix this issue by removing AspectJ capability from the project and re-enabling it. Note that you will need to do this every time your workflow or activities interfaces change. Because of this issue, we recommend that you use the load time weaving option instead. See the section Setting up the AWS Flow Framework for Java (p. 3) for more details.

Unknown Resource Fault

Amazon SWF returns unknown resource fault when you try to perform an operation on a resource that isn’t available. The common causes for this fault are:

- You configure a worker with a domain that doesn’t exist. To fix this, first register the domain using the Amazon SWF console or the Amazon SWF service API.
- You try to create workflow execution or activity tasks of types that have not been registered. This can happen if you try to create the workflow execution before the workers have been run. Since workers register their types when they are run for the first time, you must run them at least once before attempting to start executions (or manually register the types using the Console or the service API). Note that once types have been registered, you can create executions even if no worker is running.
- A worker attempts to complete a task that has already timed out. For example, if a worker takes too long to process a task and exceeds a timeout, it will get an UnknownResource fault when it attempts to complete or fail the task. The AWS Flow Framework workers will continue to poll Amazon SWF and process additional tasks. However, you should consider adjusting the timeout. Adjusting the timeout requires that you register a new version of the activity type.

Exceptions When Calling get() on a Promise

Unlike Java Future, Promise is a non-blocking construct and calling get() on a Promise that isn’t ready yet will throw an exception instead of blocking. The correct way to use a Promise is to pass it...
to an asynchronous method (or a task) and access its value in the asynchronous method. AWS Flow Framework for Java ensures that an asynchronous method is called only when all Promise arguments passed to it have become ready. If you believe your code is correct or if you run into this while running one of the AWS Flow Framework samples, then it is most likely due to AspectJ not being properly configured. For details, see the section Setting up the AWS Flow Framework for Java (p. 3).

Non Deterministic Workflows

As described in the section Nondeterminism (p. 122), the implementation of your workflow must be deterministic. Some common mistakes that can lead to nondeterminism are use of system clock, use of random numbers, and generation of GUIDs. Since these constructs may return different values at different times, the control flow of your workflow may take different paths each time it is executed (see the sections AWS Flow Framework Basic Concepts: Distributed Execution (p. 38) and Under the Hood (p. 119) for details). If the framework detects nondeterminism while executing the workflow, an exception will be thrown.

Problems Due to Versioning

When you implement a new version of your workflow or activity—for instance, when you add a new feature—you should increase the version of the type by using the appropriate annotation: @Workflow, @Activities, or @Activity. When new versions of a workflow are deployed, often times you will have executions of the existing version that are already running. Therefore, you need to make sure that workers with the appropriate version of your workflow and activities get the tasks. You can accomplish this by using a different set of task lists for each version. For example, you can append the version number to the name of the task list. This ensures that tasks belonging to different versions of the workflow and activities are assigned to the appropriate workers.

Troubleshooting and Debugging a Workflow Execution

The first step in troubleshooting a workflow execution is to use the Amazon SWF console to look at the workflow history. The workflow history is a complete and authoritative record of all the events that changed the execution state of the workflow execution. This history is maintained by Amazon SWF and is invaluable for diagnosing problems. The Amazon SWF console enables you to search for workflow executions and drill down into individual history events.

AWS Flow Framework provides a WorkflowReplayer class that you can use to replay a workflow execution locally and debug it. Using this class, you can debug closed and running workflow executions. WorkflowReplayer relies on the history stored in Amazon SWF to perform the replay. You can point it to a workflow execution in your Amazon SWF account or provide it with the history events (for example, you can retrieve the history from Amazon SWF and serialize it locally for later use). When you replay a workflow execution using the WorkflowReplayer, it doesn't impact the workflow execution running in your account. The replay is done completely on the client. You can debug the workflow, create breakpoints, and step into code using your debugging tools as usual. If you are using Eclipse, consider adding step filters to filter AWS Flow Framework packages.

For example, the following code snippet can be used to replay a workflow execution:

```java
String workflowId = "testWorkflow";
String runId = "<run id>";
```


AWS Flow Framework also allows you to get an asynchronous thread dump of your workflow execution. This thread dump gives you the call stacks of all open asynchronous tasks. This information can be useful to determine which tasks in the execution are pending and possibly stuck. For example:

```
String workflowId = "testWorkflow";
String runId = "<run id>";
Class<HelloWorldImpl> workflowImplementationType = HelloWorldImpl.class;
WorkflowExecution workflowExecution = new WorkflowExecution();
workflowExecution.setWorkflowId(workflowId);
workflowExecution.setRunId(runId);
WorkflowReplayer<HelloWorldImpl> replayer = new WorkflowReplayer<HelloWorldImpl>(
    swfService, domain, workflowExecution, workflowImplementationType);
try {
    String flowThreadDump = replayer.getAsynchronousThreadDumpAsString();
    System.out.println("Workflow asynchronous thread dump:");
    System.out.println(flowThreadDump);
} catch (WorkflowException e) {
    System.out.println("No asynchronous thread dump available as workflow has failed: " + e);
}
```

**Lost Tasks**

Sometimes you may shut down workers and start new ones in quick succession only to discover that tasks get delivered to the old workers. This can happen due to race conditions in the system, which is distributed across several processes. The problem can also appear when you are running unit tests in a tight loop. Stopping a test in Eclipse can also sometimes cause this because shutdown handlers may not get called.

In order to make sure that the problem is in fact due to old workers getting tasks, you should look at the workflow history to determine which process received the task that you expected the new worker to receive. For example, the DecisionTaskStarted event in history contains the identity of the workflow worker that received the task. The id used by the Flow Framework is of the form: `{processId}@{hostname}`. For instance, following are the details of the DecisionTaskStarted event in the Amazon SWF console for a sample execution:

<table>
<thead>
<tr>
<th>Event Timestamp</th>
<th>Mon Feb 20 11:52:40 GMT-800 2012</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity</td>
<td>2276@ip-0A6C1DF5</td>
</tr>
<tr>
<td>Scheduled Event Id</td>
<td>33</td>
</tr>
</tbody>
</table>
In order to avoid this situation, use different task lists for each test. Also, consider adding a delay between shutting down old workers and starting new ones.
@Activities

This annotation can be used on an interface to declare a set of activity types. Each method in an interface annotated with this annotation represents an activity type. An interface can’t have both @Workflow and @Activities annotations.

The following parameters can be specified on this annotation:

activityNamePrefix

Specifies the prefix of the name of the activity types declared in the interface. If set to an empty string (which is the default), the name of the interface followed by ‘.’ is used as the prefix.

version

Specifies the default version of the activity types declared in the interface. The default value is 1.0.

dataConverter

Specifies the type of the DataConverter to use for serializing/deserializing data when creating tasks of this activity type and its results. Set to NullDataConverter by default, which indicates that the JsonDataConverter should be used.
@Activity

This annotation can be used on methods within an interface annotated with @Activities.

The following parameters can be specified on this annotation:

name

Specifies the name of the activity type. The default is an empty string, which indicates that the
default prefix and the activity method name should be used to determine the name of the activity
type (which is of the form {prefix}{name}). Note that when you specify a name in an @Activity
annotation, the framework will not automatically prepend a prefix to it. You are free to use your own
naming scheme.

version

Specifies the version of the activity type. This overrides the default version specified in the
@Activities annotation on the containing interface. The default is an empty string.

@ActivityRegistrationOptions

Specifies the registration options of an activity type. This annotation can be used on an interface
annotated with @Activities or the methods within. If specified in both places, then the annotation
used on the method takes effect.

The following parameters can be specified on this annotation:

defaultTaskList

Specifies the default task list to be registered with Amazon SWF for this activity type. This
default can be overridden when calling the activity method on the generated client using the
ActivitySchedulingOptions parameter. Set to USE_WORKER_TASK_LIST by default. This
is a special value which indicates that the task list used by the worker, which is performing the
registration, should be used.

defaultTaskScheduleToStartTimeoutSeconds

Specifies the defaultTaskScheduleToStartTimeout registered with Amazon SWF for this activity type.
This is the maximum time a task of this activity type is allowed to wait before it is assigned to a
worker. See the Amazon Simple Workflow Service API Reference for more details.

defaultTaskHeartbeatTimeoutSeconds

Specifies the defaultTaskHeartbeatTimeout registered with Amazon SWF for this activity type.
Activity workers must provide heartbeat within this duration; otherwise, the task will be timed out.
Set to -1 by default, which is a special value that indicates this timeout should be disabled. See the
Amazon Simple Workflow Service API Reference for more details.

defaultTaskStartToCloseTimeoutSeconds

Specifies the defaultTaskStartToCloseTimeout registered with Amazon SWF for this activity type.
This timeout determines the maximum time a worker can take to process an activity task of this
type. See the Amazon Simple Workflow Service API Reference for more details.

defaultScheduleToCloseTimeoutSeconds

Specifies the defaultScheduleToCloseTimeout registered with Amazon SWF for this activity type.
This timeout determines the total duration that the task can stay in open state. Set to -1 by
default, which is a special value that indicates this timeout should be disabled. See the Amazon
Simple Workflow Service API Reference for more details.
@Asynchronous

When used on a method in the workflow coordination logic, indicates that the method should be executed asynchronously. A call to the method will return immediately, but the actual execution will happen asynchronously when all Promise<> parameters passed to the methods become ready. Methods annotated with @Asynchronous must have a return type of Promise<> or void.

demon

Indicates if the task created for the asynchronous method should be a daemon task. False by default.

@Execute

When used on a method in an interface annotated with the @Workflow annotation, identifies the entry point of the workflow.

Important

Only one method in the interface can be decorated with @Execute.

The following parameters can be specified on this annotation:

name

Specifies the name of the workflow type. If not set, the name defaults to {prefix}{name}, where {prefix} is the name of the workflow interface followed by a '.' and {name} is the name of the @Execute-decorated method in the workflow.

version

Specifies the version of the workflow type.

@ExponentialRetry

When used on an activity or asynchronous method, sets an exponential retry policy if the method throws an unhandled exception. A retry attempt is made after a back-off period, which is calculated by the power of the number of attempts.

The following parameters can be specified on this annotation:

initialRetryIntervalSeconds

Specifies the duration to wait before the first retry attempt. This value should not be greater than maximumRetryIntervalSeconds and retryExpirationSeconds.

maximumRetryIntervalSeconds

Specifies the maximum duration between retry attempts. Once reached, the retry interval is capped to this value. Set to -1 by default, which means unlimited duration.

retryExpirationSeconds

Specifies the duration after which exponential retry will stop. Set to -1 by default, which means there is no expiration.

backoffCoefficient

Specifies the coefficient used to calculate the retry interval. Set to 2 by default. You can set this to 1 to get linear back-off or 0 to get a constant back-off.
maximumAttempts

Specifies the number of attempts after which exponential retry will stop. Set to -1 by default, which means there is no limit on the number of retry attempts.

exceptionsToRetry

Specifies the list of exception types that should trigger a retry. Unhandled exception of these types will not propagate further and the method will be retried after the calculated retry interval. By default, the list contains Throwable.

excludeExceptions

Specifies the list of exception types that should not trigger a retry. Unhandled exceptions of this type will be allowed to propagate. The list is empty by default.

@GetState

When used on a method in an interface annotated with the @Workflow annotation, identifies that the method is used to retrieve the latest workflow execution state. There can be at most one method with this annotation in an interface with the @Workflow annotation. Methods with this annotation must not take any parameters and must have a return type other than void.

@ManualActivityCompletion

This annotation can be used on an activity method to indicate that the activity task should not be completed when the method returns. The activity task will not be automatically completed and would need to be completed manually directly using the Amazon SWF API. This is useful for use cases where the activity task is delegated to some external system that isn't automated or requires human intervention to be completed.

@Signal

When used on a method in an interface annotated with the @Workflow annotation, identifies a signal that can be received by executions of the workflow type declared by the interface. Use of this annotation is required to define a signal method.

The following parameters can be specified on this annotation:

name

Specifies the name portion of the signal name. If not set, the name of the method is used.

@SkipRegistration

When used on an interface annotated with the @Workflow annotation, indicates that the workflow type should not be registered with Amazon SWF. One of @WorkflowRegistrationOptions and @SkipRegistrationOptions annotations must be used on an interface annotated with @Workflow, but not both.

@Wait and @NoWait

These annotations can be used on a parameter of type Promise<> to indicate whether the AWS Flow Framework for Java should wait for it to become ready before executing the method. By default, Promise<> parameters passed into @Asynchronous methods must become ready before method
execution occurs. In certain scenarios, it is necessary to override this default behavior. Promise<> parameters passed into @Asynchronous methods and annotated with @NoWait are not waited for.

Collections parameters (or subclasses of) that contain promises, such as List<Promise<Int>>, must be annotated with @Wait annotation. By default, the framework doesn't wait for the members of a collection.

**@Workflow**

This annotation is used on an interface to declare a workflow type. An interface decorated with this annotation should contain exactly one method that is decorated with the @Execute (p. 129) annotation to declare an entry point for your workflow.

**Note**

An interface can't have both @Workflow and @Activities annotations declared at once; they are mutually exclusive.

The following parameters can be specified on this annotation:

**dataConverter**

Specifies which DataConverter to use when sending requests to, and receiving results from, workflow executions of this workflow type.

The default is NullDataConverter which, in turn, falls back to JsonDataConverter to process all request and response data as JavaScript Object Notation (JSON).

**Example**

```java
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 3600)
public interface GreeterWorkflow {
    @Execute(version = "1.0")
    public void greet();
}
```

**@WorkflowRegistrationOptions**

When used on an interface annotated with @Workflow, provides default settings used by Amazon SWF when registering the workflow type.

**Note**

Either @WorkflowRegistrationOptions or @SkipRegistrationOptions must be used on an interface annotated with @Workflow, but you can't specify both.

The following parameters can be specified on this annotation:

**Description**

An optional text description of the workflow type.

**defaultExecutionStartToCloseTimeoutSeconds**

Specifies the defaultExecutionStartToCloseTimeout registered with Amazon SWF for the workflow type. This is the total time that a workflow execution of this type can take to complete.
defaultTaskStartToCloseTimeoutSeconds

Specifies the defaultTaskStartToCloseTimeout registered with Amazon SWF for the workflow type. This specifies the time a single decision task for a workflow execution of this type can take to complete.

If you don't specify defaultTaskStartToCloseTimeout, it will default to 30 seconds.

For more information about workflow timeouts, see Amazon SWF Timeout Types (p. 44).

defaultTaskList

The default task list used for decision tasks for executions of this workflow type. The default set here can be overridden by using StartWorkflowOptions when starting a workflow execution.

If you don't specify defaultTaskList, it will be set to USE_WORKER_TASK_LIST by default. This indicates that the task list used by the worker that is performing the workflow registration should be used.

defaultChildPolicy

Specifies the policy to use for child workflows if an execution of this type is terminated. The default value is ABANDON. The possible values are:

- ABANDON – Allow the child workflow executions to keep running
- TERMINATE – Terminate child workflow executions
- REQUEST_CANCEL – Request cancellation of the child workflow executions

AWS Flow Framework for Java Exceptions

The following exceptions are used by the AWS Flow Framework for Java. This section provides an overview of the exception. For more details, see the AWS SDK for Java documentation of the individual exceptions.

Topics

- ActivityFailureException (p. 133)
- ActivityTaskException (p. 133)
- ActivityTaskFailedException (p. 133)
- ActivityTaskTimedOutException (p. 133)
- ChildWorkflowException (p. 133)
- ChildWorkflowFailedException (p. 133)
- ChildWorkflowTerminatedException (p. 133)
- ChildWorkflowTimedOutException (p. 134)
- DataConverterException (p. 134)
- DecisionException (p. 134)
- ScheduleActivityTaskFailedException (p. 134)
- SignalExternalWorkflowException (p. 134)
- StartChildWorkflowFailedException (p. 134)
- StartTimerFailedException (p. 134)
- TimerException (p. 134)
• WorkflowException (p. 134)

**ActivityFailureException**

This exception is used by the framework internally to communicate activity failure. When an activity fails due to an unhandled exception, it is wrapped in `ActivityFailureException` and reported to Amazon SWF. You need to deal with this exception only if you use the activity worker extensibility points. Your application code will never need to deal with this exception.

**ActivityTaskException**

This is the base class for activity task failure exceptions: `ScheduleActivityTaskFailedException`, `ActivityTaskFailedException`, `ActivityTaskTimedoutException`. It contains the task Id and activity type of the failed task. You can catch this exception in your workflow implementation to deal with activity failures in a generic way.

**ActivityTaskFailedException**

Unhandled exceptions in activities are reported back to the workflow implementation by throwing an `ActivityTaskFailedException`. The original exception can be retrieved from the cause property of this exception. The exception also provides other information that is useful for debugging purposes, such as the unique activity identifier in the history.

The framework is able to provide the remote exception by serializing the original exception from the activity worker.

**ActivityTaskTimedOutException**

This exception is thrown if an activity was timed out by Amazon SWF. This could happen if the activity task could not be assigned to the worker within the require time period or could not be completed by the worker in the required time. You can set these timeouts on the activity using the `@ActivityRegistrationOptions` annotation or using the `ActivitySchedulingOptions` parameter when calling the activity method.

**ChildWorkflowException**

Base class for exceptions used to report failure of child workflow execution. The exception contains the Ids of the child workflow execution as well as its workflow type. You can catch this exception to deal with child workflow execution failures in a generic way.

**ChildWorkflowFailedException**

Unhandled exceptions in child workflows are reported back to the parent workflow implementation by throwing a `ChildWorkflowFailedException`. The original exception can be retrieved from the cause property of this exception. The exception also provides other information that is useful for debugging purposes, such as the unique identifiers of the child execution.

**ChildWorkflowTerminatedException**

This exception is thrown in parent workflow execution to report the termination of a child workflow execution. You should catch this exception if you want to deal with the termination of the child workflow, for example, to perform cleanup or compensation.
ChildWorkflowTimedOutException

This exception is thrown in parent workflow execution to report that a child workflow execution was
timed out and closed by Amazon SWF. You should catch this exception if you want to deal with the
forced closure of the child workflow, for example, to perform cleanup or compensation.

DataConverterException

The framework uses the DataConverter component to marshal and unmarshal data that is sent over
the wire. This exception is thrown if the DataConverter fails to marshal or unmarshal data. This could
happen for various reasons, for example, due to a mismatch in the DataConverter components being
used to marshal and unmarshal the data.

DecisionException

This is the base class for exceptions that represent failures to enact a decision by Amazon SWF. You can
catch this exception to generically deal with such exceptions.

ScheduleActivityTaskFailedException

This exception is thrown if Amazon SWF fails to schedule an activity task. This could happen due to
various reasons—for example, the activity was deprecated, or an Amazon SWF limit on your account
has been reached. The failureCause property in the exception specifies the exact cause of failure to
schedule the activity.

SignalExternalWorkflowException

This exception is thrown if Amazon SWF fails to process a request by the workflow execution to signal
another workflow execution. This happens if the target workflow execution could not be found—that is,
the workflow execution you specified doesn't exist or is in closed state.

StartChildWorkflowFailedException

This exception is thrown if Amazon SWF fails to start a child workflow execution. This could happen due
to various reasons—for example, the type of child workflow specified was deprecated, or a Amazon SWF
limit on your account has been reached. The failureCause property in the exception specifies the
exact cause of failure to start the child workflow execution.

StartTimerFailedException

This exception is thrown if Amazon SWF fails to start a timer requested by the workflow execution. This
could happen if the specified timer ID is already in use, or an Amazon SWF limit on your account has
been reached. The failureCause property in the exception specifies the exact cause of failure.

timerException

This is the base class for exceptions related to timers.

WorkflowException

This exception is used internally by the framework to report failures in workflow execution. You need to
deal with this exception only if you are using a workflow worker extensibility point.
AWS Flow Framework for Java Packages

This section provides an overview of the packages included with the AWS Flow Framework for Java. For more information about each package, see the com.amazonaws.services.simpleworkflow.flow in the AWS SDK for Java API Reference.

com.amazonaws.services.simpleworkflow.flow

Contains components that integrate with Amazon SWF.

com.amazonaws.services.simpleworkflow.flow.annotations

Contains the annotations used by the AWS Flow Framework for Java programming model.

com.amazonaws.services.simpleworkflow.flow.aspectj

Contains AWS Flow Framework for Java components required for features such as @Asynchronous (p. 129) and @ExponentialRetry (p. 129).

com.amazonaws.services.simpleworkflow.flow.common

Contains common utilities such as framework-defined constants.

com.amazonaws.services.simpleworkflow.flow.core

Contains core features such as Task and Promise.

com.amazonaws.services.simpleworkflow.flow.generic

Contains core components, such as generic clients, that other features build on.

com.amazonaws.services.simpleworkflow.flow.interceptors

Contains implementations of framework provided decorators including RetryDecorator.

com.amazonaws.services.simpleworkflow.flow.junit

Contains components that provide Junit integration.

com.amazonaws.services.simpleworkflow.flow.pojo

Contains classes that implement activity and workflow definitions for the annotation-based programming model.

com.amazonaws.services.simpleworkflow.flow.spring

Contains components that provide Spring integration.

com.amazonaws.services.simpleworkflow.flow.test

Contains helper classes, such as TestWorkflowClock, for unit testing workflow implementations.

com.amazonaws.services.simpleworkflow.flow.worker

Contains implementations of activity and workflow workers.
Document History

The following table describes the important changes to the documentation since the last release of the *AWS Flow Framework for Java Developer Guide*.

- **API version:** 2012-01-25
- **Latest documentation update:** June 5, 2017

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>Cleaned up the code examples throughout this guide.</td>
<td>June 5, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Simplified and improved the organization and contents of this guide.</td>
<td>May 19, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Simplified and improved the Making Changes to Decider Code: Versioning and Feature Flags (p. 47) section.</td>
<td>April 10, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Added the new Best Practices (p. 47) section with new guidance on making changes to decider code.</td>
<td>March 3, 2017</td>
</tr>
<tr>
<td>New feature</td>
<td>You can specify Lambda tasks in addition to traditional Activity tasks in your workflows. For more information, see Implementing AWS Lambda Tasks (p. 76).</td>
<td>July 21, 2015</td>
</tr>
<tr>
<td>New feature</td>
<td>Amazon SWF includes support for setting the task priority on a task list, attempting to deliver the tasks with a higher priority before tasks with lower priority. For more information, see Setting Task Priority (p. 88).</td>
<td>December 17, 2014</td>
</tr>
<tr>
<td>Update</td>
<td>Made updates and fixes.</td>
<td>August 1, 2013</td>
</tr>
</tbody>
</table>
| Update      | • Made updates and fixes, including updates of the setup instructions (p. 3) for Eclipse 4.3 and AWS SDK for Java 1.4.7.  
  • Added a new set of tutorials for building starter scenarios | June 28, 2013    |
| New feature | The initial release of the AWS Flow Framework for Java.                     | February 27, 2012 |
AWS Glossary

For the latest AWS terminology, see the AWS Glossary in the AWS General Reference.