Amazon Simple Workflow Service

AWS Flow Framework Recipes

API Version 2012-01-25

Amazon Simple Workflow Service: AWS Flow Framework Recipes

AWS Flow Framework Recipes

The AWS Flow Framework is a programming framework that simplifies the process of implementing workflows that run on Amazon Simple Workflow Service (Amazon SWF). This document describes a set of recipes for the Java programming language, each of which shows how to address a common use case. The recipes are accompanied by JUnit tests that run the workflows.

The Recipes

The following briefly describes the recipes and provides pointers to detailed walkthroughs.

Repeatedly Execute an Activity (p. 6)

- ForLoopInlineRecipeWorkflowImpl (p. 6) shows how to use a `for` loop to repeatedly execute an activity a specified number of times.
- ConditionalLoopActivitiesImpl (p. 8) shows how to use a recursive asynchronous method to repeatedly execute an activity a specified number of times.
- DoWhileWorkflowImpl (p. 8) shows how to use a recursive asynchronous method to repeatedly execute an activity while a condition is satisfied.

Execute Multiple Activities Concurrently (p. 10)

- RunMultipleActivitiesConcurrentlyWorkflowImpl (p. 10) shows how to run a fixed number of activities concurrently and merge the results.
- JoinBranchesWorkflowImpl (p. 11) shows how to run a dynamically determined number of activities concurrently and merge the results.
- PickFirstBranchWorkflowImpl (p. 12) shows how to execute multiple activities concurrently and use the result from the first activity to complete.

Execute Workflow Logic Conditionally (p. 15)

- ExclusiveChoiceWorkflowImpl (p. 16) shows how to execute one of several activities based on a condition.
- MultiChoiceWorkflowImpl (p. 17) shows how to execute multiple activities from a larger group based on a condition.

Complete an Activity Task Manually (p. 18)

- humanActivity (p. 18) shows how to implement an activity that is manually completed by a person.

Handle Exceptions Thrown by Asynchronous Code (p. 21)

- CleanupResourceWorkflowImpl (p. 21) shows how to use `TryCatchFinally` to handle exceptions thrown by asynchronous code such as activities and clean up resources.
- HandleErrorWorkflowImpl (p. 23) shows how to handle exceptions thrown by asynchronous code by invoking other asynchronous code.

Retry Failed Asynchronous Code (p. 26)

- RetryActivityRecipeWorkflowImpl (p. 26) shows how to retry an activity by simply retrying the activity until it either completes or the retry attempts reach a specified limit.
- ExponentialRetryAnnotationActivities (p. 29) shows how to annotate an activity so that the framework retries it automatically by using an exponential retry strategy, which waits for an increasingly long period between each retry, and stops at a specified point.
- DecoratorRetryWorkflowImpl (p. 29) shows how to implement exponential retry by using the `RetryDecorator` class, which allows you to specify the retry policy at run time and change it as needed.
- AsyncExecutorRetryWorkflowImpl (p. 30) shows how to implement exponential retry by using the `AsyncRetryingExecutor` class, which allows you to specify the retry policy at run time. In addition, you use the `AsyncRunnable` abstraction to implement a `run` method, which `AsyncRetryingExecutor` calls to execute the activity for each retry attempt.
- CustomLogicRetryWorkflowImpl (p. 32) shows how to implement a custom retry strategy.

Signal a Workflow (p. 34)

- WaitForSignalWorkflowImpl (p. 34) shows how a workflow can wait a specified time for a signal before proceeding.

The recipes are contained in a set of Java packages, some of which contain multiple recipes. The following table shows the recipes that are in the various packages. All recipes are in the com.amazonaws.services.simpleworkflow.flow.recipes package, which is omitted for brevity.

AWS Flow Framework Recipes

Package	Recipes
branch	JoinBranchesWorkflowImpl (p. 11)
conditionloop	ConditionalLoopActivitiesImpl (p. 8)
choice	ExclusiveChoiceWorkflowImpl (p. 16), MultiChoiceWorkflowImpl (p. 17)
dowhile	DoWhileWorkflowImpl (p. 8)
forloopinline	ForLoopInlineRecipeWorkflowImpl (p. 6)
handleerror	CleanupResourceWorkflowImpl (p. 21), HandleErrorWorkflowImpl (p. 23)
humantask	humanActivity (p. 18)
pickfirstbranch	PickFirstBranchWorkflowImpl (p. 12)

Package	Recipes
retryactivity	RetryActivityRecipeWorkflowImpl (p. 26), ExponentialRetryAnnotationActivities (p. 29), DecoratorRetryWorkflowImpl (p. 29), AsyncExecutorRetryWorkflowImpl (p. 30), CustomLogicRetryWorkflowImpl (p. 32)
runningmultipleactivities	RunMultipleActivitiesConcurrentlyWorkflowImpl (p. 10)
waitforsignal	WaitForSignalWorkflowImpl (p. 34)

Some Background

The recipes depend on four core AWS Flow Framework technologies, which are briefly described in this section. For a detailed discussion, see AWS Flow Framework Developer Guide.

The Promise<T> Type—Promise<T> represents the future result of an activity or asynchronous method, where T is the result's type. For example, an activity could return a Promise<Integer> object which represents an Integer return value. Immediately after the activity returns, the object is simply a placeholder and is in the unready state. When the activity completes, the framework assigns the return value to the Promise<Integer> object and puts it in the ready state. The primary purpose of Promise<T> objects is to manage data flow between asynchronous components. For example, when you pass the Promise<Integer> object returned by an activity to another activity, the second activity defers execution until the first activity completes and the Promise<Integer> object is ready.

The Settable<T> Type—Settable<T> is derived from Promise<T>. If you pass a Settable<T> object to an asynchronous method, it defers execution until the Settable<T> object is ready, much like Promise<T>. You use Settable<T> instead of Promise<T> when you want to manually set the object's value and put it in the ready state instead of letting the framework handle the task.

Activities—Activities are a mechanism for distributing tasks across multiple processes and perhaps across multiple systems. From a programming perspective you call an activity much like would a method. However, activities run asynchronously and Amazon SWF mediates communication between a workflow and its activities. When you call an activity, it immediately returns a Promise<T> object in the unready state, which allows the workflow to continue. You can then pass the Promise<T> object to other activities or asynchronous methods as an argument, which causes them to defer execution. When the activity completes, the framework assigns the return value to the Promise<T> object, changing its status to ready and allowing any dependent activities or asynchronous methods to execute.

Asynchronous Methods—An asynchronous method runs in the workflow implementation's context but executes asynchronously, much like an activity. You designate a method as asynchronous by applying an AWS Flow Framework @Asynchronous annotation, and the method typically takes one or more Promise<T> objects as input. When a workflow implementation calls an asynchronous method, it returns immediately but defers execution until the input Promise<T> objects are ready. Asynchronous methods are commonly used to process the Promise<T> result objects that are returned by activities. If a workflow implementation simply calls the object's get method and the Promise<T> object is unready, get throws an exception. Instead, you pass the Promise<T> object to an asynchronous method. It doesn't execute until the Promise<T> object is in the ready state, allowing you to safely call get to retrieve the result.

Before You Start

The AWS Flow Framework Recipes documentation assumes that you have at least a basic understanding of the Amazon SWF and the AWS Flow Framework. Before you start, you should be familiar with the basics of:

- Amazon SWF, as described in the Amazon Simple Workflow Service Developer Guide's Introduction, Getting Set Up, and Basic Concepts sections.
- The AWS Flow Framework, as described in the AWS Flow Framework Developer Guide's What is AWS Flow Framework (Java), AWS Flow Framework Concepts, Setting up ..., and Feature Details sections.

How to Install, Build, and Run the Recipes

1. Follow the instructions in Setting up ... to set up your development environment.
2. Extract the contents of recipes.zip and copy the `AWSFlowFrameworkRecipes` folder to the AWS SDK for Java `samples` folder, which is under the root folder. For example: `C:\InstalledApps\aws-java-sdk-1.3.22\samples\AWSFlowFrameworkRecipes\....`
3. The recipes' JUnit tests require version 4.10 of JUnit.jar, which you can obtain from Kentbeck/junit.
4. Create a `JUnit-4.10` folder under the AWS SDK for Java `third-party` folder, which is under the root folder, and put a copy of the JUnit 4.10 junit.jar file in the folder. For example: `C:\InstalledApps\aws-java-sdk-1.3.22\third-party\junit-4.10\junit-4.10.jar`.

Each recipe is accompanied by a JUnit test that executes the recipe's workflow. To build and run the recipes from the command line, open a command window and navigate to the `AWSFlowFrameworkRecipes` root folder, which includes a build.xml file.

- To build all the recipes, run:

```
ant -f build.xml
```

- To build and run all the recipes, run:

```
ant -f build.xml testall
```

- To build and run a particular recipe, run:

```
ant -f build.xml -Dtest-class="com.amazonaws.services.simpleworkflow.flow.re
cipes.branch.test_class" test
```

For example, the following command runs the JoinBranchesWorkflowImpl (p. 11) recipe.

```
ant -f build.xml -Dtest-class="com.amazonaws.services.simpleworkflow.flow.re
cipes.branch.JoinBranchesWorkflowTest" test
```

To build and run the recipes by using Eclipse:

1. Click **File->New->Other...**
2. Select **Java Project from Existing Ant Build File**
3. Select build.xml from the `AWSFlowFrameworkRecipes` folder and optionally specify a project name.
4. Select the project in **Project Explorer** and open the **Project Properties** dialog box.
5. In the left pane, select **Java Compiler->Annotation Processing**.
6. Select the **Enable project specific settings** and **Enable annotation processing** check boxes and build the project.

7. Right-click the test of interest and select **Run As->JUnit Test** to run the workflow.

Repeatedly Execute an Activity

With standard Java programs, a `for`, `while`, or `do-while` loop executes its code block multiple times, in sequence. That's not necessarily what happens with asynchronous code. For example, the following code fragment executes the `doNothing` activity n times.

```
for (int i = 0; i < n; i++) {
    client.doNothing();
}
```

Because `client.doNothing` is asynchronous, the `for` loop immediately invokes the `doNothing` activity n times. However, the activity tasks execute later and might do so in parallel rather than in sequence.

This section describes three recipes that show how to repeatedly execute asynchronous code:

Topics

Repeatedly Execute an Activity a Fixed Number of Times by using a Java for Loop

Problem: You need to repeatedly execute an activity a specified number of times.

Solution: Use a `for` loop and pass the `Promise<T>` object returned by the activity to the activity's next invocation.

This recipe is implemented in the forloopinline package. The workflow interface is defined in `ForLoopInlineRecipeWorkflow` and has a single `loop` method, which is the workflow's entry point. The workflow is implemented in `ForLoopInlineRecipeWorkflowImpl`, as follows:

```
public class ForLoopInlineRecipeWorkflowImpl implements ForLoopInlineRecipeWork
```

```
flow {
     ...
    @Override
    public void loop(int n) {
        if (n > 0) {
            Promise<Void> hasRun = Promise.Void();
            for (int i = 0; i < n; i++) {
                hasRun = client.doNothing(hasRun);
            }
        }
    }
}
```

The `for` loop immediately calls the `doNothing` activities client method n times. However, the framework can't schedule the corresponding activity tasks for execution until their input `Promise<T>` objects become ready. Because each `doNothing` invocation depends on a `Promise<T>` object from the preceding invocation, the activities execute one after the other.

Note

For the initial iteration, `loop` uses the static `Promise.Void` method to set `hasRun` to a `Promise<Void>` object in the ready state, so the framework executes the first activity immediately.

Repeatedly Execute an Activity a Fixed Number of Times by using a Recursive Asynchronous Method

Problem: You need to execute an activity a specified number of times in sequence.

Solution: Use a recursive asynchronous method to implement the asynchronous equivalent of a `for` loop.

This recipe processes a set of records by using the following activities:

- `getRecordCount` determines the number of records to be processed. For simplicity, the activity simply returns a random integer from 1 - 20.
- `processRecord` processes a record.

This recipe is implemented in the conditionloop package. The workflow interface is defined in`ConditionalLoopWorkflow` and has a single `startWorkflow` method which is the workflow's entry point. The workflow is implemented in `ConditionalLoopWorkflowImpl`, as follows:

```
public class ConditionalLoopWorkflowImpl implements ConditionalLoopWorkflow {
    ...
    @Override
    public void startWorkflow() {
        Promise<Integer> recordCount = client.getRecordCount();
        processRecords(recordCount);
    }
```

```
    @Asynchronous
    public void processRecords(Promise<Integer> recordCount) {
        processRecords(recordCount.get());
    }

    @Asynchronous
    public void processRecords(int records, Promise<?>... waitFor) {
        if (records >= 1) {
            Promise<Void> nextWaitFor = client.processRecord();
            processRecords(records - 1, nextWaitFor);
        }
    }
}
```

The number of records is provided by an activity, so `startWorkflow` passes the returned `Promise<Integer>` object to an asynchronous method , which defers execution until `recordCount` is ready and then starts the processing loop. The workflow implements the processing loop by recursively calling the asynchronous `processRecords` method, which takes the remaining number of records and an optional `Promise<T>` object.

- `recordCount` starts the processing loop by passing only the number of records to `processRecords`, so the method immediately executes the first instance of the activity.
- For the remaining iterations, `processRecords` calls itself and passes in the remaining number of records and `nextWaitFor`. That call to `processRecords` defers execution until the current activity is complete and `nextWaitFor` is ready, ensuring that the activities execute one after the other.

This approach produces the same overall result as the Java `for` loop described in Repeatedly Execute an Activity a Fixed Number of Times by using a Java for Loop (p. 6)—the activities execute one after the other. However, the details are different.

- A `for` loop invokes all the activities at once and the framework then executes the activities one at a time.
- A recursive asynchronous method invokes and executes one activity with each iteration. The next iteration doesn't start until the previous activity completes, so there is never more than one pending activity task.

Repeatedly Execute an Activity While a Condition is Satisfied

Problem: You need to execute an activity repeatedly while a condition is satisfied.

Solution: Use mutually recursive asynchronous methods to implement the asynchronous equivalent of a `while` or `do-while` loop.

Java `while` or `do-while` loops aren't well suited for repeatedly executing an activity or asynchronous method.

- The activities might execute in parallel instead of in sequence.
- You often want the loop's condition to depend on a `Promise<T>` object returned by the activity. Each iteration must therefore wait until the previous activity completes and the returned `Promise<T>` object represents a valid value. A Java loop condition cannot wait on a `Promise<T>` object.

This recipe is implemented in the dowhile package and uses the `getRandomNumber` activity, which randomly generates an integer from 0 - 3. The workflow interface is defined in `DoWhileWorkflow` and has a single `doWhile` method, which is the workflow's entry point. The workflow is implemented in `DoWhileWorkflowImpl`, as follows:

```
public class DoWhileWorkflowImpl implements DoWhileWorkflow {
    ...
    @Override
    public void doWhile() {
        doBody();
    }

    @Asynchronous
    private void doBody() {
        Promise<Integer> bodyResult = client.getRandomNumber();
        whileNext(bodyResult);
    }

    @Asynchronous
    private void whileNext(Promise<Integer> bodyResult) {
        if (bodyResult.get() >= 1) {
            doBody();
        }
    }
}
```

The processing loop uses mutually recursive pair of asynchronous methods:

- `doBody` executes the `doNothing` activity. Because it can't evaluate the condition until the activity completes and `bodyResult` is ready `doBody` passes `bodyResult` to the asynchronous `whileNext` method.
- `whileNext` determines whether to perform the next iteration. It defers execution until the current `doNothing` activity completes, extracts the value from `bodyResult` and tests whether the condition evaluates to `true`. If so, `whileNext` calls `doBody` to start the next iteration. Otherwise, the loop terminates.

To provide `do-while` functionality, the workflow calls `doBody` before `whileNext`, so the activity executes at least once. You can implement an asynchronous `while` loop in much the same way by testing the condition before executing the activity instead of after.

Execute Multiple Activities Concurrently

Problem: You need to execute multiple activities concurrently.

Solution: Call the activities client methods and use the returned `Promise<T>` values to defer processing the results until the activities complete.

This section describes three recipes that show how to run multiple activities concurrently.

Topics

Execute a Fixed Number of Activities Concurrently

Problem: You need to execute a fixed number of activities concurrently and merge the results.

Solution: Call the activities client methods and pass the `Promise<T>` return values to an asynchronous method to merge the results after the activities complete.

This recipe is implemented in the runningmultipleactivities package and uses the `generateRandomNumber` activity, which returns a randomly generated integer. The workflow interface is defined in `RunMultipleActivitiesConcurrentlyWorkflow` and has a single `runMultipleActivitiesConcurrently` method, which is the workflow's entry point. The workflow is implemented in `RunMultipleActivitiesConcurrentlyWorkflowImpl`, as follows:

```
public class RunMultipleActivitiesConcurrentlyWorkflowImpl implements RunMul
tipleActivitiesConcurrentlyWorkflow {
    ...
    @Override
```

```
    public void runMultipleActivitiesConcurrently() {
        Promise<Integer> result1 = client.generateRandomNumber();
        Promise<Integer> result2 = client.generateRandomNumber();
        processResults(result1, result2);
    }
    @Asynchronous
    public void processResults(Promise<Integer> result1, Promise<Integer> res
ult2) {
        if (result1.get() + result2.get() > 5) {
            client.generateRandomNumber();
        }
    }
}
```

Neither activity client method has a `Promise<T>` input object, so they don't defer execution and both activities execute concurrently. `runMultipleActivitiesConcurrently` passes the activities' `Promise<T>` result objects to the asynchronous `processResults` method for processing, which defers execution until both result objects are ready.

Execute a Dynamically Determined Number of Activities Concurrently

Problem: You need to execute a dynamically determined number of activities concurrently and merge the results.

Solution: Call the activities client methods, package the return values as a collection of `Promise<T>` objects, and pass the collection to an activity or asynchronous method which merges the results after the activities complete.

This recipe is implemented in the branch package and uses the following activities:

- `doSomeWork` performs a simple integer computation and returns a `Promise<Integer>` result.
- `reportResults` reports the result and returns a `Promise<Integer>` result.

The workflow interface is defined in `JoinBranchesWorkflowImpl` interface and has a single `parallelComputing` method, which is the workflow's entry point. The workflow is implemented in `JoinBranchesWorkflowImpl`, as follows:

```
public class JoinBranchesWorkflowImpl implements JoinBranchesWorkflow {
     ...
    @Override
    public Promise<Integer> parallelComputing(int branches) {
         List<Promise<Integer>> results = new ArrayList<Promise<Integer>>();
        for (int i = 0; i < branches; i++) {
            Promise<Integer> result = client.doSomeWork();
            results.add(result);
        }
        Promise<Integer> sum = joinBranches(results);
        return client.reportResult(sum);
    }
```

```
    @Asynchronous
    public Promise<Integer> joinBranches(@Wait List<Promise<Integer>> results)
 {
        int sum = 0;
        for (Promise<Integer> result : results) {
            sum += result.get();
        }
        return Promise.asPromise(sum);
    }
}
```

The workflow works as follows

- `doSomeWork` does not take a `Promise<T>` input object, so it does not defer execution and the `for` loop invokes every instance of the activity concurrently.
- To merge the results after all activities complee, `parallelComputing` packages the returned `Promise<Integer>` objects into a `List<Promise<Integer>>` object and passes it to the asynchronous `joinBranches` method.
- `joinBranches` defers execution until all activities complete. However, `results` is a `List<Promise<T>>` object, which is not a `Promise<T>` type and does not by itself cause an activity or asynchronous method to defer execution. The `@Wait` annotation directs `joinBranches` to defer execution until every `Promise<T>` object in the collection is ready.

Execute Multiple Activities Concurrently and Use the Fastest

Problem: You need to execute multiple activities concurrently, use the result from the first activity to complete, and cancel the rest.

Solution: Create parallel execution branches by calling the activities client methods in separate `TryCatch` blocks. When one of the activities completes, cancel the others by canceling their `TryCatch` blocks.

For example, you need to run a lengthy computation or query as quickly as possible but don't know which of the available systems or clusters will be the fastest. You can implement a "fastest wins" strategy by executing multiple activities in parallel, each of which runs the computation on one of the available systems or clusters. You then use result from the first activity to complete, and cancel the other branches.

Note

You don't need `TryCatch` blocks to implement parallel execution branches, but they allow you to later cancel branches.

This recipe uses two activities, `searchCluster1` and `searchCluster2`, which run the queries. Each activity takes a query string as input, and returns a collection of strings. The workflow interface is defined in `PickFirstBranchWorkflow` and has a single `search` method, which is the workflow's entry point. The workflow is implemented in `PickFirstBranchWorkflowImpl`. The following example shows the `search` method:

```
public class PickFirstBranchWorkflowImpl implements PickFirstBranchWorkflow {
    ...
    @Override
    public Promise<List<String>> search(final String query) {
```

```
        Promise<List<String>> branch1Result = searchOnCluster1(query);
        Promise<List<String>> branch2Result = searchOnCluster2(query);

       OrPromise branch1OrBranch2 = new OrPromise(branch1Result, branch2Result);

        return processResults(branch1OrBranch2);
    }
    ...
}
```

`search` works as follows:

1. It executes the two activities in parallel by calling `searchCluster1` and `searchCluster2`, which are discussed later.
2. It uses the activities' returned `Promise<String>` objects to create an `OrPromise` object, which becomes ready when any of its `Promise<T>` objects becomes ready.
3. It passes the `OrPromise` object to the asynchronous `processResults` method for processing, which defers execution until one of the activities completes.

Note

AWS Flow Framework also includes an `AndPromise` object, which is similar to `OrPromise`, but becomes ready when *all* of its `Promise<T>` objects become ready.

The following example shows `searchOnCluster1`, which creates the execution branch for `cluster1Result`. `searchOnCluster2` is implemented in essentially the same way.

```
public class PickFirstBranchWorkflowImpl implements PickFirstBranchWorkflow {
    private TryCatch branch1;
    ...
    Promise<List<String>> searchOnCluster1(final String query) {
        final Settable<List<String>> result = new Settable<List<String>>();
        branch1 = new TryCatch() {
            @Override
            protected void doTry() throws Throwable {
                Promise<List<String>> cluster1Result = client.search
Cluster1(query);
                result.chain(cluster1Result);
            }
            @Override
            protected void doCatch(Throwable e) throws Throwable {
                if (!(e instanceof CancellationException)) {
                    throw e;
                }
            }
        };
        return result;
    }
}
```

`searchOnCluster1` executes the activity in the `TryCatch` class's `doTry` method, and *chains* the returned `Promise<List<String>>` object to a `Settable<List<String>>` object named `result`. Chaining allows other parts of an application to use a `Promise<T>` object that is returned within the scope of a nested `TryCatch` class.

- If `branch2` completes first, `processResults` cancels `branch1`. If the cancellation attempt takes place before `searchCluster1` completes, `doCatch` handles the resulting exception.
- Otherwise, `searchOnCluster1` returns `result`, which represents the activity's results in the rest of the application.

Note

The `Settable<T>` type is derived from `Promise<T>` but you make a `Settable<T>` object ready by manually setting its value.

When one of the activities completes, the `branch1OrBranch2` object becomes ready and `processResults` handles the results, as shown in the following example.

```
public class PickFirstBranchWorkflowImpl implements PickFirstBranchWorkflow {
    private TryCatch branch1;
    private TryCatch branch2;
    ...
    @Asynchronous
    Promise<List<String>> processResults(OrPromise result) {
        Promise<List<String>> output = null;
       Promise<List<String>> branch1Result = (Promise<List<String>>) result.get
Values()[0];
       Promise<List<String>> branch2Result = (Promise<List<String>>) result.get
Values()[1];
        if (branch1Result.isReady()) {
            output = branch1Result;
            if (!branch2Result.isReady()) {
                branch2.cancel(null);
            }
        }
        else {
            output = branch2Result;
            branch1.cancel(null);
        }
        return output;
    }
    ...
}
```

`processResults` works as follows:

1. It retrieves the activities' result objects from the `OrPromise` object, which are stored in the order that they are passed to the constructor.
2. It uses the `result` object's `isReady` methods to determine which branch completed and attempts to cancel the other branch by calling its `TryCatch.cancel` method.

Execute Workflow Logic Conditionally

Problem: You need to execute a one or more activities from a larger group of activities based on workflow input or a computed value such as a return value from an activity.

Solution: Pass the input or computed value to an asynchronous method, which uses the value to execute the appropriate activities.

Any particular workflow execution might need to use only some of the available activities. For example, a workflow that handles customer orders might have a "handle order" activity for each available product. However, each workflow execution handles a particular customer who will order only some of the available products. The workflow instance therefore needs to execute only the appropriate activities, based on externally provided customer data.

The recipes in this section are implemented in the choice package and use the same activities. The `orderApple`, `orderOrange`, `orderLettuce`, and `orderCabbage` activities represent possible orders and return a `Promise<Void>` object.

- `getItemOrder`, gets an order for a single item and returns a `Promise<OrderChoice>` object, where `OrderChoice` is a private enumeration.
- `getBasketOrder`, gets orders for one or more items and returns the result as a collection of `Promise<OrderChoice>` objects.
- `finishOrder`, which completes the order.

The workflow interface for both recipes is implemented in `OrderChoiceWorkflow` and has a single `processOrder` method, which is the workflow's entry point. The workflow implementations are discussed in the following sections.

Topics

Execute One of Several Activities

Problem: You need to execute one of several activities based on workflow input or a computed value such activity's result.

Solution: Use an asynchronous method to execute the selected activity by using the workflow input or computed value as the condition for a Java branch statement such as `switch`.

The workflow is implemented in `ExclusiveChoiceWorkflowImpl`, as follows:

```
public class ExclusiveChoiceWorkflowImpl implements OrderChoiceWorkflow {

    ...
    @Override
    public void processOrder() {
        Promise<OrderChoice> itemChoice = client.getItemOrder();
        Promise<Void> waitFor = processItemOrder(itemChoice);
        client.finishOrder(waitFor);
    }

    @Asynchronous
    public Promise<Void> processItemOrder(Promise<OrderChoice> itemChoice) {
        OrderChoice choice = itemChoice.get();
        Promise<Void> result = null;
        switch (choice) {
        case APPLE:
            result = client.orderApple();
            break;
        case ORANGE:
            result = client.orderOrange();
            break;
        case LETTUCE:
            result = client.orderLettuce();
            break;
        case CABBAGE:
            result = client.orderCabbage();
            break;
        }
        return result;
    }
}
```

The workflow works as follows:

1. `processOrder` executes the `getItemOrder` activity to get the customer's order. It passes the result object to the asynchronous `processItemOrder` method for processing, which defers execution until `itemChoice` is ready.
2. `processItemOrder` extracts the customer's order from `itemChoice` and uses it to select and execute the appropriate order activity.
3. `processOrder` passes the `processItemOrder` activity's `result` object to the `finishOrder` activity, which defers execution until the order activity completes and then finishes the order.

Execute Multiple Activities from a Larger Group

Problem: You need to execute one or more activities from a larger group based on workflow input or a computed value such activity's result.

Solution: Determine which activities to execute and use an asynchronous method to execute them in parallel. Merge the results by passing a collection containing the returned `Promise<T>` objects to an asynchronous method.

The workflow is implemented in `MultiChoiceWorkflowImpl`, as follows:

```
public class MultiChoiceWorkflowImpl implements OrderChoiceWorkflow {
     ...
    @Override
    public void processOrder() {
        Promise<List<OrderChoice>> basketChoice = client.getBasketOrder();
        Promise<List<Void>> waitFor = processBasketOrder(basketChoice);
        client.finishOrder(waitFor);
    }

    @Asynchronous
    public Promise<List<Void>> processBasketOrder(Promise<List<OrderChoice>>
basketChoice) {

        List<OrderChoice> choices = basketChoice.get();
        List<Promise<Void>> results = new ArrayList<Promise<Void>>();

         for (OrderChoice choice : choices) {
            Promise<Void> result = processSingleChoice(choice);
            results.add(result);
        }
        return Promises.listOfPromisesToPromise(results);
    }
    public Promise<Void> processSingleChoice(OrderChoice choice) {
        ...
    }
}
```

The workflow works as follows:

1. `processOrder` executes the `getBasketOrder` activity to get the customer's order. It passes the result object to the asynchronous `processBasketOrder` method for processing, which defers execution until `basketChoice` is ready.
2. `processBasketOrder` retrieves the list of orders from `basketChoice` and processes them by executing the `processSingleChoice` activity once for each order. `processSingleChoice` does not have any `Promise<T>` input, so the order activities execute concurrently.
3. The order can't be finished until all the activities are completed, so `processOrder` uses the asynchronous `finishOrder` method to finalize the order. However, the result objects are packaged as a `List<Promise<Void>>` collection, which is not itself a `Promise<T>` type and will not cause `finishOrder` to defer execution as required. Instead, `processBasketOrder` uses the static `Promises.listOfPromisesToPromise` method to convert the results collection into a `Promise<List<Void>>` object, which becomes ready when all the order activities have completed.

Complete an Activity Task Manually

Problem: You have an activity that needs to return immediately and be completed later, for example, by a person assigned to perform the task.

Solution: Apply the `@ManualActivityCompletion` annotation to the activity, which allows the activity method to return immediately and be completed later.

By default, when an activity method returns, Amazon SWF considers the activity task to be complete and the framework assigns the return value to the associated `Promise<T>` object and puts the object in the ready state. However, it is sometimes useful to separate return from completion. For example, you might need an activity that assigns a task to a person and returns immediately. The person then manually completes the task when they are finished.

This recipe is implemented in the humantask package and uses a three activities. `automatedActivity`, and `sendNotification` are normal activities but the `humanActivity` activity must be manually completed. The recipe also includes a console application, `HumanTaskConsole`, that completes `humanActivity`.

The workflow interface is defined in `HumanTaskWorkflow` and has a single `startWorkflow` method, which is the workflow's entry point. The workflow implementation runs the three activities in sequence, as shown in the following example.

```
public class HumanTaskWorkflowImpl implements HumanTaskWorkflow {
    ...
    @Override
    public void startWorkflow() {
        Promise<Void> automatedResult1 = client.automatedActivity();
        Promise<String> humanResult = client.humanActivity(automatedResult1);
        client.sendNotification(humanResult);
    }
}
```

Notice that from the workflow implementation's perspective, `humanActivity` is just another activity. It returns a `Promise<T>` object, and the workflow proceeds when the object becomes ready, regardless of how the activity is completed. Manual completion is part of the activity implementation; you can change the activity from manual to normal completion or vice versa without affecting the workflow implementation.

`humanActivity` is implemented as follows:

```
public class HumanTaskActivitiesImpl implements HumanTaskActivities {
   ActivityExecutionContextProvider contextProvider = new ActivityExecutionCon
textProviderImpl();
    ...
    @Override
    @ManualActivityCompletion
    public String humanActivity() {
        ActivityExecutionContext executionContext = contextProvider.getActiv
ityExecutionContext();
        String taskToken = executionContext.getTaskToken();
        System.out.println("Task received, completion token: " + taskToken);
        return null;
    }
}
```

`humanActivity` works as follows:

- The `@ManualActivityCompletion` annotation designates `humanActivity` for manual completion. The framework ignores the return values from such methods, so `humanActivity` simply returns `null` and the returned `Promise<String>` object remains unready.
- Amazon SWF assigns a unique task token to each activity task, which can be used later to complete the task. For simplicity, `humanActivity` just gets the token and prints it to the console. You should copy it for later use.

The `humanActivity` client method returns a `Promise<String>` object, which remains in the unready state after the activity method returns. It doesn't become ready until it is manually completed. This recipe has the assigned person manually complete the task by running the `HumanTaskConsole` application, which is implemented as follows:

```
public class HumanTaskConsole {
    public static void main(String[] args) throws IOException {
        //get task token and result from user
        String taskToken = getTaskToken();
        String result = getResult();

        //complete the activity task
        AmazonSimpleWorkflow swfService = createSWFClient();
        ManualActivityCompletionClientFactory manualCompletionClientFactory =
new ManualActivityCompletionClientFactoryImpl(
                swfService);
        ManualActivityCompletionClient manualCompletionClient = manualComple
tionClientFactory.getClient(taskToken);
        manualCompletionClient.complete(result);
    }
    ...
}
```

For brevity, the example omits several straightforward utility methods: `getTaskToken`, `getResult`, and `createSWFClient`. See the recipe for details.

The application uses an Amazon SWF client and the stored task token to create a manual completion client and completes the task by passing the result value to the completion client's `complete` method. The framework assigns that value to `humanResult` and puts it in the ready state, which allows the `sendNotification` activity to execute and complete the workflow.

You can use the same basic approach to manually complete activities in a variety of ways. For example, you could store the task token in a database and have an automated process retrieve the token and complete the task.

Handle Exceptions Thrown by Asynchronous Components

Problem: When an asynchronous component such as an activity fails, it throws an exception that you must handle and perhaps perform resource cleanup. However, a standard Java `try/catch` or `try/catch/finally` block can't handle exceptions thrown by asynchronous code.

Solution: Use the AWS Flow Framework API's `TryCatch`, `TryFinally`, or `TryCatchFinally` classes to catch and handle the exception and perform any required cleanup.

The following recipes are implemented in the handlerror package and use the same activities:

- `allocateResource` returns a resource ID as a `Promise<Integer>` object.
- `useResource` takes the resource ID and throws one of two exceptions randomly. The exceptions are represented by the following types.
  - `ResourceNoResponseException` extends `Exception` and indicates that `useResource` did not get a response.
  - `ResourceNotAvailableException` extends `Exception` and indicates that the resource is not available.
- `cleanUpResource` cleans up the resource specified by the resource ID.
- `reportBadResource` handles a `ResourceNoResponseException`.
- `refreshResourceCatalog` handles a `ResourceNotAvailableException`.

Topics

Handle Exceptions and Perform Cleanup by using TryCatchFinally

Problem: You need to handle any exceptions that might be thrown by asynchronous code and guarantee that resources are cleaned up.

Solution: Use the `TryCatchFinally` class and override `doTry`, `doCatch`, and `doFinally` to run asynchronous code, handle any exceptions, and perform resource cleanup.

This recipe calls `UseResource` and handles the resulting exceptions. The workflow interface is defined in `CleanupResourceWorkflow` and has one method, `startWorkFlow`, which is the workflow's entry point. The workflow is implemented in `CleanupResourceWorkflowImpl`, as follows:

```
public class CleanupResourceWorkflowImpl implements CleanupResourceWorkflow {
    ...
    @Override
    public void startWorkflow() {
        final Promise<Integer> resourceId = client.allocateResource();
        new TryCatchFinally() {
            @Override
            protected void doTry() throws Throwable {
                client.useResource(resourceId);
            }
            @Override
            protected void doCatch(Throwable e) throws Throwable {
                 client.rollbackChanges(resourceId);
            }
            @Override
            protected void doFinally() throws Throwable {
                if (resourceId.isReady()) {
                    client.cleanUpResource(resourceId);
                }
            }
        };
    }
}
```

To handle asynchronous exceptions and clean up resources, create a nested `TryCatchFinally` class and:

- Override the `doTry` method to execute the asynchronous code.
- Override the `doCatch` method to catch any exceptions. In this example, `doCatch` handles the exception by executing the `rollbackChanges` activity.
- Override the `doFinally` method to clean up resources. The framework always calls `doFinally` to perform resource cleanup after `doTry` and `doCatch` complete. In this example, `resourceID` will be in an unready state only if the exception was raised by `doTry`. In that case, the resource was not successfully created and there is no need to perform cleanup.

You cannot cancel activities that are executed in `doCatch` or `doFinally`. For an example of how to handle exceptions by executing cancellable activities, see Handle Exceptions by using TryCatch and an Asynchronous Method (p. 23).

Activities and asynchronous methods might execute in parallel. This means that `TryCatchFinally` is a sibling of the `allocateResource` activity and `doTry` can execute before or after `allocateResource` completes. If `allocateResource` throws an exception, the framework cancels `TryCatchFinally` by using the following semantics:

- If `doTry` hasn't yet executed, cancellation is immediate and none of the code in `TryCatchFinally` executes.

- If `doTry` has executed, the framework cancels all of its outstanding tasks and calls `doCatch` with a `CancellationException`. Finally, the framework calls `doFinally`, which guarantees that resources are cleaned up.

Handle Exceptions by using TryCatch and an Asynchronous Method

Problem: When an activity fails, you need to handle the exception in a way that allows you to execute cancellable activities.

Solution: Use the `TryCatch` class and override `doTry` to execute the activity. Override `doCatch` to catch any exceptions and use an asynchronous method to handle them.

This recipe calls `UseResource` and handles the resulting exceptions. The workflow interface is implemented in `HandleErrorWorkflow` and has one method, `startWorkFlow`, which is the workflow's entry point. The workflow is implemented in `HandleErrorWorkflowImpl`, as follows:

```
public class HandleErrorWorkflowImpl implements HandleErrorWorkflow {
     ...
    @Override
    public void startWorkflow() throws Throwable {
        final Settable<Throwable> exception = new Settable<Throwable>();
        final Promise<Integer> resourceId = client.allocateResource();

        new TryCatch() {
            @Override
            protected void doTry() throws Throwable {
                Promise<Void> waitFor = client.useResource(resourceId);
                setState(exception, null, waitFor);
            }

            @Override
            protected void doCatch(Throwable e) throws Throwable {
                setState(exception, e, Promise.Void());
            }
        };
        handleException(exception);
    }

    @Asynchronous
    public void handleException(Promise<Throwable> ex, Promise<Integer> re
sourceId) throws Throwable {
        Throwable e = ex.get();
        if (e != null) {
            if (e instanceof ActivityTaskFailedException) {
                Throwable inner = e.getCause();
                if (inner instanceof ResourceNoResponseException) {
                    client.reportBadResource(resourceId.get());
                }
                else if (inner instanceof ResourceNotAvailableException) {
                    client.refreshResourceCatalog(resourceId.get());
                }
                else {
```

```
                    throw e;
                }
            }
            else {
                throw e;
            }
        }
    }

    @Asynchronous
    public void setState(@NoWait Settable<Throwable> exception, Throwable ex,
Promise<Void> WaitFor) {
        exception.set(ex);
    }
}
```

To handle asynchronous exceptions, create a nested `TryCatch` class and:

- Override the `doTry` method to execute the activities.
- Override the `doCatch` method to catch any exceptions.

If you handle exceptions in `doCatch` or perform resource cleanup in `doFinally`, the code should execute promptly. However, if you execute activities in `doCatch` or `doFinally`, they are not cancellable. If, for example, you want to handle exceptions by executing a cancellable activity, have `doCatch` hand off the exception handling to an asynchronous method. That method runs after `TryCatch` completes, which allows you to execute cancellable activities.

`HandleErrorWorkflowImpl` implements this pattern as follows:

- It uses a `Settable<Throwable>` object named `exception` which is set to the exception object if an exception occurred and `null` otherwise. `Settable<T>` is derived from `Promise<T>` and works much the same way, but you make a `Settable<T>` object ready by manually setting its value.
- Both `doCatch` and `doTry` set `exception` by calling the asynchronous `setState` method. As described later, only one of these attempts succeeds, depending on whether an exception was thrown.
- `doCatch` calls the asynchronous `handleException` to perform the actual exception handling, which runs after `TryCatch` completes.

`setState` works as follows

- The `@NoWait` annotation on the first parameter directs `setState` to not defer execution until `exception` is ready. `setState` sets the `exception` object's value so you don't want to defer execution.
- `ex` takes the value to be assigned to `exception`.
  - `doTry` sets `ex` to null, indicating that no exception occurred.
  - `doCatch` sets `ex` to the exception object, indicating that an exception occurred.
- `WaitFor` takes a `Promise<Void>` object, which ensures that `setState` defers execution until the object is ready.
  - `doTry` sets this parameter to `waitFor`, which ensures that `setState` defers execution until `useResource` completes. If `useResource` instead throws an exception, `waitFor` never becomes ready and the `setState` call does not execute.
  - `doCatch` uses the static `Promise.Void` method to set this parameter to a `Promise<Void>` object in the ready state. The activity has failed, so there is no reason to defer execution.

Because `exception` is a `Settable<T>` type, `handleException` defers execution until `exception` is ready, which occurs after `TryCatch` completes. It then runs the appropriate activity to handle the exception.

Retry Failed Asynchronous Code

Problem: You need to retry failed asynchronous code because the cause of failure might be ephemeral.

Solution: Retry the code, perhaps multiple times, using an appropriate strategy.

Asynchronous code such as activities sometimes fails for ephemeral reasons, such as a temporary loss of connectivity. Because it might succeed at another time, the appropriate way to handle the failure is often to execute the code again, perhaps multiple times. There are a variety of retry strategies; the best one depends on the details of your workflow. They fall into three basic categories:

- The retry-until-success strategy keeps retrying the activity until it completes or the retry attempts reach a specified limit such as a maximum number of attempts.
- The exponential retry strategy increases the time interval between retry attempts exponentially until the activity completes or the process reaches a specified limit 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 recipes are implemented in the retryactivity package and all use the `unreliableActivity` activity, which randomly does one of following:

- Completes immediately
- Fails intentionally by exceeding the timeout value
- Fails intentionally by throwing `IllegalStateException`

Topics

Retry-Until-Success Strategy

Problem: You need to retry a failed activity because the cause of failure is probably ephemeral.

Solution: Use a retry-until-success strategy that keeps retrying the activity until it completes or the attempts reach a specified limit.

The recipe's workflow interface is defined in `RetryActivityWorkflow` and has one method, `process`, which is the workflow's entry point. The workflow is implemented in `RetryActivityWorkflowImpl`, as follows:

```
public class RetryActivityWorkflowImpl implements RetryWorkflow {
   private final RetryActivitiesClient client = new RetryActivitiesClientImpl();

    private final int maxRetries = 10;
    private int retryCount;

    @Override
    public void process() {
        final Settable<Boolean> retryActivity = new Settable<Boolean>();
        new TryCatchFinally() {
            @Override
            protected void doTry() throws Throwable {
                client.unreliableActivity();
            }
            @Override
            protected void doCatch(Throwable e) throws Throwable {
                if (++retryCount <= maxRetries) {
                    retryActivity.set(true);
                }
                else {
                    throw e;
                }
            }
            @Override
            protected void doFinally() throws Throwable {
                if (!retryActivity.isReady()) {
                    retryActivity.set(false);
                }
            }
        };
        restartRunUnreliableActivityTillSuccess(retryActivity);
    }
    @Asynchronous
    private void restartRunUnreliableActivityTillSuccess(Settable<Boolean>
retryActivity) {
        if (retryActivity.get()) {
            process();
        }
    }
}
```

`process` implements a nested `TryCatchFinally` class to execute `unreliableActivity` and catch the exception if it fails. For more examples of how to use `TryCatchFinally` to handle exceptions thrown by asynchronous code, see Handle Exceptions Thrown by Asynchronous Components (p. 21).

`process` uses a `Settable<Boolean>` object named `retryActivity` 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 make a `Settable<T>` object ready by setting its value manually.

- If `unreliableActivity` fails, `doCatch` executes. If the number of retries has not exceeded the specified limit, `doCatch` sets `retryActivity` to `true` to indicate that another retry is required.
- The framework always calls `doFinally` after `doTry` and `doCatch` complete. If `unreliableActivity` completed successfully, or the number of retries reached the specified limit, `retryActivity` is still in

the unready state. In that case, doFinally sets retryActivity to false to indicate that no more retries are required.

process 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 TryCatchFinally completes. When retryActivity is ready, restartRunUnreliableActivityTillSuccess extracts the value.

- If the value is false, the retry succeeded or the number of retry attempts reached the specified limit. restartRunUnreliableActivityTillSuccess does nothing and the retry sequence terminates.
- If the value is true, the retry failed. restartRunUnreliableActivityTillSuccess calls process to retry the activity again.

Note

doCatch does not 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 cannot cancel it. Retrying the activity in restartRunUnreliableActivityTillSuccess allows you to execute cancellable activities.

Exponential Retry Strategy

Problem: An activity fails, but the cause might be ephemeral. However, the cause might persist for a period of time and immediate retry might not succeed.

Solution: Use an exponential strategy, which waits for an increasingly long period between each retry, and stops at a specified point.

With the exponential retry strategy, the framework executes a failed activity again after a specified period of time. If that attempt fails the framework continues retrying the activity using a time interval that is based on the following formula, where the back-off coefficient is a user-specified value, in seconds:

```
retryInterval = initialInterval * Math.pow(backoffCoeff, numberOfTries - 2)
```

You typically stop the retry attempts at some point rather than continuing indefinitely.

The framework provides three ways to implement an exponential retry strategy. All approaches support the following retry policy options, where time values are in seconds:

- The initial retry wait time.
- The back-off coefficient. 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 Throwable triggers the retry process.
- The exceptions that will not trigger a retry attempt. By default, no exceptions are excluded.

Topics

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 annotation's arguments.

The recipe implements an exponential retry strategy by applying an `@ExponentialRetry` annotation to the `unreliableActivity` activity's interface definition, as follows:

```
@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 retry policy:

- 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 defined in `RetryWorkflow` and has one method, `process`, which is the workflow's entry point. The workflow is implemented in `ExponentialRetryAnnotationWorkflowImpl`, as follows:

```
public class ExponentialRetryAnnotationWorkflowImpl implements RetryWorkflow {

     ...
    public void process() {
        client.unreliableActivity();
    }
}
```

If `unreliableActivity` fails, the framework automatically retries the activity according to the specified retry policy.

Exponential Retry with the RetryDecorator Class

The `@ExponentialRetry` configuration is static and set at compile time, so the framework uses the same retry policy 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 retry policy at run time and change it as needed.

Important

The `@ExponentialRetry` annotation and the `RetryDecorator` class are mutually exclusive. You cannot use `RetryDecorator` to dynamically override a retry policy specified by an `@ExponentialRetry` annotation.

The recipe workflow uses an `unreliableActivity` activity that does not have an `@ExponentialRetry` annotation. The workflow interface is defined in `RetryWorkflow` and has one method, `process`, which is the workflow's entry point. The workflow is implemented in `DecoratorRetryWorkflowImpl`, as follows:

```
public class DecoratorRetryWorkflowImpl implements RetryWorkflow {
   private RetryActivitiesClient client = new RetryActivitiesClientImpl();

   public void process() {
        long initialRetryIntervalSeconds = 5;
        int maximumAttempts = 5;
        ExponentialRetryPolicy retryPolicy = new ExponentialRetryPolicy(ini
tialRetryIntervalSeconds).withMaximumAttempts(maximumAttempts);

        Decorator retryDecorator = new RetryDecorator(retryPolicy);
        client = retryDecorator.decorate(RetryActivitiesClient.class, client);

        handleUnreliableActivity();
    }

    public void handleUnreliableActivity() {
        client.unreliableActivity();
    }
}
```

`DecoratorRetryWorkflowImpl` implements the retry strategy, as follows:

1. Create and configure `ExponentialRetryPolicy` object, which exposes a set of `with`*XYZ* methods that you can use to specify the retry policy.
2. Create a `RetryDecorator` object and pass the `ExponentialRetryPolicy` object to the constructor.
3. Apply the `RetryDecorator` object to the workflow implementation class. `decorate` takes the original activities client object and returns a decorated object.

If the activity fails, the framework retries it according to the `ExponentialRetryPolicy` object's configuration specified in Step 1. You can change the retry policy at any time by calling the `ExponentialRetryPolicy` object's `set`*XYZ* methods: `setBackoffCoefficient`, `setMaximumAttempts`, `setMaximumRetryIntervalSeconds`, and `setMaximumRetryExpirationIntervalSeconds`.

Exponential Retry with the AsyncRetryingExecutor Class

The `AsyncRetryingExecutor` class allows you to configure the retry process at run time. In addition, you use the `AsyncRunnable` abstraction to implement a `run` method, which `AsyncRetryingExecutor` calls to execute the activity for each retry attempt.

This recipe's workflow interface is implemented in `RetryWorkflow` and has one method, `process`, which is the workflow's entry point. The workflow is implemented in `AsyncExecutorRetryWorkflowImpl`, as follows:

```
public class AsyncExecutorRetryWorkflowImpl implements RetryWorkflow {
   private final RetryActivitiesClient client = new RetryActivitiesClientImpl();

   private final DecisionContextProvider contextProvider = new DecisionContex
tProviderImpl();
   private final WorkflowClock clock = contextProvider.getDecisionContext().get
WorkflowClock();

    public void process() {
        long initialRetryIntervalSeconds = 5;
        int maximumAttempts = 5;
       handleUnreliableActivity(initialRetryIntervalSeconds, maximumAttempts);

    }
    public void handleUnreliableActivity(long initialRetryIntervalSeconds, int
 maximumAttempts) {

        ExponentialRetryPolicy retryPolicy = new ExponentialRetryPolicy(ini
tialRetryIntervalSeconds).withMaximumAttempts(maximumAttempts);
        final AsyncExecutor executor = new AsyncRetryingExecutor(retryPolicy,
clock);

        new TryCatch() {
            @Override
            protected void doTry() throws Throwable {
                executor.execute(new AsyncRunnable() {
                    @Override
                    public void run() throws Throwable {
                        client.unreliableActivity();
                    }
                });
            }
            @Override
            protected void doCatch(Throwable e) throws Throwable {
            }
        };
    }
}
```

`AsyncExecutorRetryWorkflowImpl` first creates an `AsyncRetryExecutor` object and passes it a configured `ExponentialRetryPolicy`—which specifies the retry policy—and an instance of the workflow clock. For details on `ExponentialRetryPolicy`, see Exponential Retry with the RetryDecorator Class (p. 29). `AsyncExecutorRetryWorkflowImpl` then implements a nested `TryCatch` class to execute the activity and catch any errors, as follows:

- Execute the activity in `doTry` by passing an anonymous `AsyncRunnable` object to `AsyncRetryExecutor.execute`. Override `AsyncRunnable.run` to run the activity. For simplicity, `run` just executes the activity, but you can implement more sophisticated approaches as appropriate.
- If you exceed the retry limit, the framework throws an exception which is caught by `doCatch`. This example does nothing, allowing the workflow to proceed.

Note

The `TryCatch` block is an implementation detail, and is not required. For some workflows, it might be preferable to omit `TryCatch` and let the exception propagate, perhaps causing the workflow to fail.

For more discussion of how to use the `TryCatch` class to handle exceptions thrown by asynchronous code, see Handle Exceptions Thrown by Asynchronous Components (p. 21).

Custom Retry Strategy

Problem: You need to retry a failed activity, but retry-until-success and exponential retry strategies don't adequately handle the issue.

Solution: Implement a custom retry strategy.

The most flexible approach to retrying failed activities is a custom strategy, which recursively calls an asynchronous method that implements custom logic to decide whether and how to run each retry attempt.

The recipe's workflow interface is implemented in `RetryWorkflow` and has one method, `process`, which is the workflow's entry point. The workflow is implemented in `CustomLogicRetryWorkflowImpl`, as follows:

```
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 custom strategy implementation is basically similar to that used by the retry-until-success strategy (p. 26), except that it uses custom logic to decide whether and how to run each retry attempt instead of simply executing the activity. It runs the retry attempts by using a pair of mutually recursive asynchronous methods.

`callActivityWithRetry` executes the activity as follows:

1. Create 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.
2. Use a `TryCatchFinally` block to execute the activity.
   - If the activity fails, `doCatch` sets `failure` to the exception object, which puts it in the ready state.
   - `doFinally` checks whether `failure` is ready, which will be true only if the activity failed and `doCatch` set `failure`. If not, `doFinally` sets `failure` to null, indicating that the activity completed.

   For more discussion of how to use the `TryCatch` class to handle exceptions thrown by asynchronous code, see Handle Exceptions Thrown by Asynchronous Components (p. 21).

If the retry attempt fails, `callActivityWithRetry` calls the asynchronous `retryOnFailure` method to decide whether to run another attempt. If so, `retryOnFailure` calls `callActivityWithRetry` to run the next attempt.

Note

`doCatch` does not 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 `TryCatchFinally` completes. The reason for this approach is that, if you retry an activity in `doCatch`, you cannot cancel it. Retrying the activity in `retryOnFailure` allows you to execute cancellable activities.

Wait for a Signal

Problem: Your workflow implementation needs to wait a specified time for a signal before proceeding.

Solution: Include a signal method in your workflow implementation, and use a workflow clock timer to determine to determine how long to wait for the signal to be called.

Much like the workflow entry point, a signal method can have any suitable name and arguments. Just include the method in the workflow interface definition and apply a `@Signal` annotation. The AWS Flow Framework annotation processor creates a corresponding method in the workflow client class, which can be used by applications such as the workflow starter to pass data to the workflow execution.

This recipe is implemented in the waitforsignal package. It is an order processing workflow that defers processing the order for a specified time period, which allows it to change the order's details if it receives a signal with a modified order within that time period. The workflow interface is defined in `WaitForSignalWorkflow`, and includes entry point and signal methods, as follows:

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

The workflow is implemented in `WaitForSignalWorkflowImpl`, as follows:

```
public class WaitForSignalWorkflowImpl implements WaitForSignalWorkflow {
    private Settable<Integer> signalReceived = new Settable<Integer>();
    private final int changeOrderPeriod = 30;
    ...
    public WaitForSignalWorkflowImpl() {
        DecisionContextProvider provider = new DecisionContextProviderImpl();
        DecisionContext context = provider.getDecisionContext();
        clock = context.getWorkflowClock();
    }
    @Override
```

```
    public void placeOrder(int amount) {
        Promise<Void> timer = startDaemonTimer(changeOrderPeriod);
        OrPromise signalOrTimer = new OrPromise(timer, signalReceived);
        processOrder(amount, signalOrTimer);
    }
    @Asynchronous
    private void processOrder(int originalAmount, Promise<?> waitFor) {
        int amount = originalAmount;
        if (signalReceived.isReady())
            amount = signalReceived.get();
        client.processOrder(amount);
    }
    @Override
    public void changeOrder(int amount) {
        if(!signalReceived.isReady()){
            signalReceived.set(amount);
        }
    }
    @Asynchronous(daemon = true)
    private Promise<Void> startDaemonTimer(int seconds) {
        Promise<Void> timer = clock.createTimer(seconds);
        return timer;
    }
}
```

The workflow waits to execute the `processOrder` activity until either a specified time period has passed or an application signals the workflow to change the order amount. It uses two objects to determine how and when to process the order.

- `signalReceived` is a `Settable<Integer>` object. If `changeOrder` is called, it sets `signalReceived` to the new order amount, which puts the object in the ready state. `Settable<T>` is derived from `Promise<T>` and works much the same way, but you make a `Settable<T>` object ready by manually setting its value.
- `timer` is a Promise<Void> object that is returned by a workflow clock timer. It becomes ready when the timer expires.

Note

The timer is created by the asynchronous `startDaemonTimer` method, which has an `@Asynchronous(daemon = true)` annotation that designates the method as a *daemon task*. The framework automatically cancels daemon tasks when the rest of the workflow is complete. For details, see Daemon Tasks.

`placeOrder` creates an `OrPromise` object named `signalOrTimer` and passes `signalReceived` and `timer` to the constructor. An `OrPromise` object becomes ready when any of its `Promise<T>` objects becomes ready.

`processOrder` method defers execution until `signalOrTimer` is in the ready state, which happens when either the timer expires or the workflow receives a signal. `processOrder` then checks whether `signalReceived` is ready. If so, the signal was received before the timer expired and `processOrder` changes the order amount before calling the `processOrder` activity to process the order.

The JUnit test, `WaitForSignalTest`, serves as the workflow starter, as follows:

```
public class WaitForSignalTestimplements WaitForSignalActivities {
```

```
    ...
    @Test
    public void testWorkflowSignaled() {
       WaitForSignalWorkflowClientFactory factory = new WaitForSignalWorkflow
ClientFactoryImpl();
        WaitForSignalWorkflowClient workflowClient = factory.getClient();
        Promise<Void> done = workflowClient.placeOrder(100);

        Promise<?> runId = workflowClient.getRunId();
        sendSignal(workflowClient, 200, runId);
        assertAmount(200, done);
    }...
    @Asynchronous
    private void sendSignal(WaitForSignalWorkflowClient workflowClient, int
amount, Promise<?>... waitFor) {
        workflowClient.changeOrder(amount);
    }
}
```

`testWorkflowSignaled` calls the workflow client's entry point method to start the workflow and place the order. It then:

1. Calls the client’s `getRunId` method, which returns a `Promise<T>` object, `runID`, that becomes ready after the workflow starts executing.
2. Passes `runID` to the asynchronous `sendSignal` method. The method doesn’t use the ID, but the fact that it’s a `Promise<T>` object ensures that `sendSignal` doesn’t execute before the workflow starts.
3. When `runID` is ready, `sendSignal` signals the workflow by calling the workflow client’s signal method. If the signal arrives before the timer expires, the workflow processes the order with the new amount.