AWS Flow Framework for Ruby Developer Guide

Release 2.4.0

Amazon Web Services

Jul 19, 2016

CHAPTER 1

What is the AWS Flow Framework for Ruby?

The AWS Flow Framework for Ruby is a version of AWS Flow Framework designed for the Ruby programming language; it provides all of the benefits of the AWS Flow Framework while remaining true to idiomatic Ruby programming practices. Since the AWS Flow Framework handles the mechanics of coordinating workflow tasks and communicating with Amazon Simple Workflow Service, you can focus instead on developing the code that describes your workflow logic.

Contents

1.1 What's in this guide?

Getting Started Provides basic information for new users about how to set up and use the framework.

Flow Concepts Describes the conceptual components of a flow application. Understanding these will help you to understand how Amazon SWF works and what parts of a workflow you are responsible for registering and implementing.

Basic Workflow Programming Covers the basics of workflow programming, describing how to register domains, program activities and workflows, start task pollers, how to start a workflow execution, and how to set options.

Advanced Topics Covers advanced workflow programming topics, such as setting task priority, programming workflow patterns, error handling, asynchronous programming, retrying workflows and troubleshooting.

Working with Other AWS Products Describes how to use the AWS Flow Framework for Ruby in conjunction with other AWS services, such as CloudWatch, AWS OpsWorks, and Amazon EBS.

Utilities Describes the command-line utilities provided with the AWS Flow Framework for Ruby that you can use to generate application skeletons and to start workers.

Additional Resources Lists additional documentation, code samples, forums and videos that don't fit within this documentation.

Document History Provides a history of this documentation, including a description of changes made with each major release.

genindex Provides an index of terms that can be used to navigate the documentation by keyword.

1.2 Important Notes About the AWS Flow Framework for Ruby

1.2.1 Tested Ruby runtimes

The AWS Flow Framework for Ruby has been tested with the official Ruby 1.9 runtime, also known as *YARV*. Newer versions of the Ruby runtime may work, but have not been tested extensively. Older versions of the Ruby runtime are unsupported.

1.2.2 Using the Framework on Microsoft Windows

Although the AWS Flow Framework for Ruby has been tested on Windows, *forking* does not work unless you are using Cygwin to run Ruby.

If you are not using Cygwin, you will need to set `use_forking` to `false` in your WorkerOptions when using the AWS Flow Framework for Ruby on Windows.

```
AWS::Flow::ActivityWorker.new(
  @domain.client, @domain, ACTIVITY_TASKLIST, klass) { { use_forking: false } }
```

If you are using the **`aws-flow-ruby`** command to spawn workers on Windows, make sure that the *number_of_forks_per_worker* parameter in the `activity_workers` section is set to `0`. The **`aws-flow-utils`** command will generate a suitable setup for you if you use the `-c local` argument on Windows. For more information, see *aws-flow-ruby* and *aws-flow-utils*.

1.2.3 Update :version whenever you update activity or workflow options

Once registered, any workflow or activity type is *immutable*. Because a workflow or activity type is identified by a combination of its name and version, whenever you modify any registration options for the type, you must also update its *version* in order to register it as a new type.

1.2.4 The AWS Management Console and the AWS SDK for Ruby have different region defaults

The AWS Management Console defaults to the `us-west-2` region, but the AWS SDK for Ruby defaults to the `us-east-1` region.

Because of this, be sure to set your AWS Management Console to the same region as the one in which you registered your Amazon SWF domain using the AWS Flow Framework for Ruby, or vice-versa. Otherwise, you won't see your registered domain in the AWS Management Console.

You can set the region used by the AWS SDK for Ruby by setting the :region option in `AWS.config`. For example:

```
AWS.config( {
  :access_key_id => 'ACCESS_KEY_ID',
  :secret_access_key => 'SECRET_ACCESS_KEY',
  :region => 'us-west-2',
} )
```

1.3 Where to Find the Source Code and Samples

The AWS Flow Framework for Ruby is an open-source project. The source code is available on GitHub at:

- http://github.com/aws/aws-flow-ruby

Code samples and recipes for the AWS Flow Framework for Ruby are also available on GitHub at:

- awslabs/aws-flow-ruby-samples

1.4 Framework and SDK References in the Text

All classes and methods used by the AWS Flow Framework for Ruby reside in the `AWS::Flow` namespace. Because of this, `AWS::Flow` is usually dropped from the text when referring to the framework's classes or methods. For example, the `AWS::Flow::Activities` class is simply referred to as the `Activities` class, and the `AWS::Flow#workflow_client` method is simply referred to as the `workflow_client` method.

Classes and their methods that reside within the `AWS::Flow` namespace follow the same rule: `AWS::Flow::Activities#activity` is referred to as `Activities#activity` in the text, and `AWS::Flow::Core::Future#get` is referred to as `Core::Future#get`.

In some cases, references are made to the underlying AWS SDK for Ruby. For references such as these, the full namespace is always used. `AWS::SimpleWorkflow` and `AWS::SimpleWorkflow::Domain` are written in full form to distinguish them from any names in the `AWS::Flow` namespace.

CHAPTER 2

Getting Started

This section provides information about the prerequisites you need to use the AWS Flow Framework for Ruby, how to set up the framework, and provides a short example of using the framework to write a simple workflow.

More information about programming with the AWS Flow Framework for Ruby can be found in *Basic Workflow Programming*.

2.1 Setting Up

To set up and use the AWS Flow Framework for Ruby, you will need to meet the *prerequisites* and then *install the framework*.

Contents

2.1.1 Prerequisites

Before you can install the framework, you must have the following software installed:

- **Ruby 1.9** or greater – The AWS Flow Framework for Ruby relies on *fibers*, which were introduced with Ruby version 1.9.1. To determine the version of Ruby that you have installed, use the following command:

```
ruby --version
```

  For information about installing Ruby for the first time, or about updating your Ruby version, visit http://www.ruby-lang.org/en/downloads/.

- **AWS SDK for Ruby** – The AWS Flow Framework for Ruby is built upon the AWS SDK for Ruby. If you use RubyGems to install the framework, then the SDK for Ruby will be automatically downloaded and installed for you—you can ignore this prerequisite. Otherwise, you will need to obtain and install the AWS SDK for Ruby *before* installing the AWS Flow Framework for Ruby.

2.1.2 Installing the Framework

You can install the AWS Flow Framework for Ruby using RubyGems or by downloading the source code and building it yourself.

Installing with RubyGems

If you have Ruby and RubyGems installed, you can install the framework on your system with the following command:

```
gem install aws-flow
```

This command will also install any additional libraries needed by the framework.

2.1.3 Building and Installing from Source

Before building the AWS Flow Framework for Ruby from source, you will first need to make sure that you have both of the *prerequisites* installed. You should also have Bundler installed on your system, which will make it easy to assemble all of the dependencies for the framework. Then, use the following procedure to get the framework installed on your system.

To build and install the framework from source

1. Download the source code from http://github.com/aws/aws-flow-ruby. There are two ways to do this:

    - Clone the repository on your local system using either of the following `git` commands, depending on whether you authenticate Git with SSH or HTTPS:

SSH	`git clone git@github.com:aws/aws-flow-ruby.git`
HTTPS	`git clone https://github.com/aws/aws-flow-ruby.git`

    - Download the code in a `.zip` archive using the URL http://github.com/aws/aws-flow-ruby/archive/master.zip and unpack the archive on your local system.

2. Using the command line, navigate to the directory where you cloned (or unpacked) the source code, and then enter the `aws-flow` directory. For example, if you downloaded the source and unpacked it in your `Downloads` directory, you would type:

```
    cd Downloads/aws-flow-ruby-master/aws-flow
```

3. Install the framework with Bundler:

```
    bundle install
```

2.2 Providing AWS Credentials

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

The AWS Flow Framework for Ruby is based on the SDK for Ruby; setup and specification of AWS credentials is the same for each. For more information, see Setting up AWS Credentials for Use with the SDK for Ruby in the AWS SDK for Ruby Developer Guide (V1).

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

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

  This file should contain lines in the following format:

```
[default]
aws_access_key_id = your_access_key_id
aws_secret_access_key = your_secret_access_key
```

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

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

  To set these variables in Linux, OS X, or Unix, use `export`:

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

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

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

- To set credentials for an EC2 instance, you should specify an IAM role and then give your EC2 instance access to that role as shown in Using IAM Roles for Amazon EC2 Instances in the AWS SDK for Ruby Developer Guide (V1).

Once you have set your AWS credentials using one of these methods, they can be loaded automatically by the AWS SDK for Ruby by using the default credential provider chain.

2.3 Hello World

To introduce you to programming with the AWS Flow Framework for Ruby, we'll begin with a variant of the famous "Hello, World" application. This version of Hello World will use Amazon SWF to schedule and run an *activity*, which implements some work to be done.

The complete code for the example is presented in this topic, but you will also find it in the awslabs/aws-flow-ruby-samples repository on GitHub along with many other examples of programming with the AWS Flow Framework for Ruby.

Contents

2.3.1 Prerequisites

This example assumes that you meet the following prerequisites:

- Ruby and the AWS Flow Framework for Ruby (at least version *2.4.0*) are installed as described in *Setting Up*.
- Your AWS credentials are configured as described in *Providing AWS Credentials*.

2.3.2 Create an Activity

An activity represents a single unit of work. At its most basic, an activity is a simple Ruby method that is housed within a class. For this example, we'll create a single activity called `hello`.

To create the "hello" activity:

1. Open a command-line (terminal) window and create a new file, `hello.rb`.
2. Add the following code:

```
class HelloWorld
  def hello(input)
    "Hello #{input[:name]}!"
  end
end
```

The method `hello` is our activity implementation—it prints a greeting customized by the value of the *input* parameter, which is provided to the activity by Amazon SWF when the activity is run.

2.3.3 Generate the Application

You can use the **`aws-flow-utils`** command to automatically generate an application skeleton for you based on the activity you just created. For this tutorial, we will create a locally-run application, but you can also create application skeletons ready for use with Elastic Beanstalk. For more information, see *Deploying Workflows with Elastic Beanstalk*.

To create an Amazon SWF application with aws-flow-utils:

- Open a command-line (terminal) window and type:

```
    aws-flow-utils -c local -n HelloWorld -a hello.rb -A HelloWorld
```

An Amazon SWF application configured to use the AWS Flow Framework for Ruby will be created for you in the local directory, called `HelloWorld`. Here is the layout of the project that will be created:

```
HelloWorld/
 |-- Gemfile
 |-- flow/
 |    |-- activities.rb
 |    |-- hello.rb
 |    `-- workflows.rb
 `-- worker.json
```

This is a standard layout for AWS Flow Framework for Ruby applications: a `flow` directory that contains your *activity* and *workflow* classes and methods, and a `worker.json` configuration file used to spawn *workers*.

2.3.4 Start an Activity Worker

To run the `hello` activity and provide it with its necessary input data, we need to start at least one activity worker to receive activity tasks from Amazon SWF. You can start the activity worker right now, and it will begin polling for tasks.

To start the worker:

- Starting within the `HelloWorld` directory, run the **`aws-flow-ruby`** utility and specify the name of your configuration file:

```
    aws-flow-ruby -f worker.json
```

The output of this command lists the process IDs of your worker threads:

```
waiting on workers [10972, 10975] to complete
```

Your worker is now polling for tasks, but to provide it with tasks to process, you need to start a *workflow execution*.

2.3.5 Starting a Workflow Execution

Now that your workers are running, you can start your activity by initiating a `workflow execution`. This signals to Amazon SWF to begin running your workflow (or in this case, a single activity). You can do this from anywhere: Amazon SWF will communicate with the workers you started to run the `HelloWorld.hello` activity.

For example, you can use the following script (call it `starter.rb`) to start the activity:

```
require 'aws/decider'
AWS::Flow::start("HelloWorld.hello", { name: "AWS Flow Framework!" })
```

To run this script, open a new command-line window and run it using Ruby:

```
ruby starter.rb
```

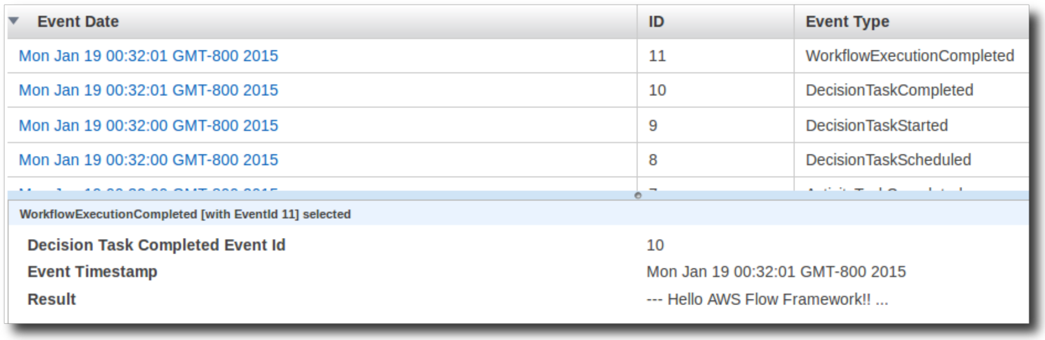
This will begin executing the `hello` activity in the background on the worker.

2.3.6 Viewing your Execution with the AWS Management Console

The application didn't provide any output—how do you know that it ran? When you use the `start` method to run an activity, Amazon SWF runs it as a workflow execution. Since Amazon SWF keeps a history of all workflow executions that you've started, you can view your activity's progress using the AWS Management Console.

To view your activity's execution history:

1. Sign in to the AWS Management Console.
2. Go to the SWF console and select the domain: `FlowDefault`.
3. Click **Workflow Executions**. By default, only *active* workflow executions are listed.
4. Perform *one* of the following actions:
   - If your workflow execution has finished, select an *Execution Status* of **Closed** in the **Workflow Execution List Parameters** view, then click **List Executions** to refresh the list.
   - If your workflow execution is still running, leave the *Execution Status* as **Active**.
5. Click the **Workflow Execution ID** associated with your workflow execution to see the details of the workflow execution.
6. Click the **Events** tab to see a view of individual workflow events, listed in order from most recent to oldest. Once your workflow execution is complete, you will see a *WorkflowExecutionCompleted* event at the top of the history.
7. Click on the date that's associated with the *WorkflowExecutionCompleted* event to view the event details, which include the result of the workflow execution:



Event Date	ID	Event Type
Mon Jan 19 00:32:01 GMT-800 2015	11	WorkflowExecutionCompleted
Mon Jan 19 00:32:01 GMT-800 2015	10	DecisionTaskCompleted
Mon Jan 19 00:32:00 GMT-800 2015	9	DecisionTaskStarted
Mon Jan 19 00:32:00 GMT-800 2015	8	DecisionTaskScheduled

WorkflowExecutionCompleted [with EventId 11] selected

Decision Task Completed Event Id	10
Event Timestamp	Mon Jan 19 00:32:01 GMT-800 2015
Result	--- Hello AWS Flow Framework!! ...

Congratulations, you've run your first Amazon SWF workflow using the AWS Flow Framework for Ruby!

2.3.7 Next Steps

This topic is meant to be only a simple introduction to the way you create workflows with AWS Flow Framework for Ruby. Use the following topics and resources to learn more about the framework:

- To learn how to create a workflow with multiple activities, see *Basic Workflow Example*.
- To learn more about the AWS Flow Framework for Ruby and about how Amazon SWF applications work, see *Flow Concepts*.
- For more information about and examples of programming with the AWS Flow Framework for Ruby, see *Basic Workflow Programming* and *Advanced Topics*.
- For information about how you can use the AWS Flow Framework for Ruby with other AWS products, see *Working with Other AWS Products*.
- To view and download working examples that demonstrate many of the features and techniques described in this documentation, see the aws-flow-ruby-samples repository on GitHub.

2.4 Basic Workflow Example

Continuing from *Hello World*, this topic provides an introduction to creating a basic workflow with the AWS Flow Framework for Ruby, and demonstrates the basic process of creating a multi-step workflow, setting options, and starting a workflow execution.

Note: The complete code for the example is presented in this topic, but you will also find it in the awslabs/aws-flow-ruby-samples repository on GitHub along with many other examples of programming with the AWS Flow Framework for Ruby.

Contents

2.4.1 Prerequisites

This example assumes that you meet the following prerequisites:

- Ruby and the AWS Flow Framework for Ruby (at least version *2.4.0*) are installed as described in *Setting Up*.
- Your AWS credentials are configured as described in *Providing AWS Credentials*.

2.4.2 Create your Application

Just as with the *Hello World* example, we'll use the **aws-flow-utils** command to generate an application skeleton project.

To create the application project:

- Open a command-line (terminal) window and type:

```
aws-flow-utils -c local -n Booking
```

An Amazon SWF application configured to use the AWS Flow Framework for Ruby will be created for you in the local directory, called `Booking`. Here is the layout of the project that will be created:

```
Booking/
 |-- Gemfile
 |-- flow/
 |    |-- activities.rb
 |    `-- workflows.rb
 `-- worker.json
```

2.4.3 Create the Activities

For this example, we'll define a couple of activities that emulate a travel-booking workflow: `reserve_car`, `reserve_air`, and `send_confirmation`.

To define the Booking activities:

1. Open the `flow/activities.rb` file in your generated `Booking` project.
2. Add the following code:

```
# BookingActivity class defines a set of activities for the Booking sample.
class BookingActivity
  extend AWS::Flow::Activities

  # The activity method is used to define activities. It accepts a list of names
  # of activities and a block specifying registration options for those
  # activities
  activity :reserve_car, :reserve_air, :send_confirmation do
    {
      version: "1.0",
    }
  end

  # This activity can be used to reserve a car for a given request_id
  def reserve_car(request_id)
    puts "Reserving car for Request ID: #{request_id}\n"
  end

  # This activity can be used to reserve a flight for a given request_id
  def reserve_air(request_id)
    puts "Reserving airline for Request ID: #{request_id}\n"
  end

  # This activity can be used to send a booking confirmation to the customer
  def send_confirmation(customer_id)
    puts "Sending notification to customer: #{customer_id}\n"
  end
end
```

The activity class is based on Activities, which provides a common interface for defining and working with activity methods. In fact, in the activity defined in the *Hello World* tutorial, AWS Flow Framework for Ruby converted the `HelloWorld` class to an `Activities`-based class behind the scenes, before running it.

In this example, the activities are assigned options using the Activities#activity method, which takes a list of activity names and assigns each of them the set of ActivityRegistrationOptions defined in the block.

As with *HelloWorld*, activities are defined by methods that take a single input parameter, and each one performs a specific job in the workflow.

2.4.4 Create the Workflow

The defined activities comprise a *synchronization* workflow pattern: the customer could either reserve a car, an airline ticket, or both. In any of these cases, a confirmation will be sent.

To implement the Booking workflow:

1. Open the `flow/workflows.rb` file in your generated `Booking` project.
2. Add the following code:

```
  require 'flow/activities'

  # BookingWorkflow class defines the workflows for the Booking sample
  class BookingWorkflow
    extend AWS::Flow::Workflows

    # Use the workflow method to define workflow entry point.
    workflow :make_booking do
      {
        version: "1.0",
        default_execution_start_to_close_timeout: 120
      }
    end

    # Create an activity client using the activity_client method to schedule
    # activities
    activity_client(:client) { { from_class: "BookingActivity" } }

    # This is the entry point for the workflow
    def make_booking options

      puts "Workflow has started\n" unless is_replaying?
      # This array will hold all futures that are created when asynchronous
```

```
        # activities are scheduled
        futures = []

        if options[:reserve_car]
          puts "Reserving a car for customer\n" unless is_replaying?
          # The activity client can be used to schedule activities
          # asynchronously by using the send_async method
          futures << client.send_async(:reserve_car, options[:request_id])
        end
        if options[:reserve_air]
          puts "Reserving air ticket\n" unless is_replaying?
          futures << client.send_async(:reserve_air, options[:customer_id])
        end

        puts "Waiting for reservation to complete\n" unless is_replaying?
        # wait_for_all is a flow construct that will wait on the array of
        # futures passed to it
        wait_for_all(futures)

        # After waiting on the reservation activities to complete, the workflow
        # will call the send_confirmation activity.
        client.send_confirmation(options[:customer_id])

        puts "Workflow has completed\n" unless is_replaying?
      end

      # Helper method to check if Flow is replaying the workflow. This is used to
      # avoid duplicate log messages
      def is_replaying?
        decision_context.workflow_clock.replaying
      end
    end
```

Workflow methods are defined in a class based on Workflows, and each workflow method takes an input parameter, just as the activity methods did. Similarly, you can use the Workflows#workflow method to set registration options for your workflows.

The workflow method, `make_booking`, uses the input parameter to choose whether or not to run the `reserve_car` and `reserve_air` activities. Each booking activity runs asynchronously using GenericClient#send_async, which returns immediately with a *future* that is filled once the activity completes. For more information, see *Executing Tasks Asynchronously*.

Activities and workflows will normally be replayed if an exception occurs, which sets the value of `is_replaying` in the `workflow_clock` attribute of the DecisionContext object held by the Workflows class. In this case, the workflow checks its value to avoid repeating its status messages with every replay.

Finally, the workflow calls Core#wait_for_all to wait for all of the running activities to complete before running the final activity, `send_confirmation`.

2.4.5 Start Workers

For the workflow and activities to run, we need to start *workers* to listen for tasks and start running the appropriate methods in our implementation. As with *HelloWorld*, we'll start the worker using the *aws-flow-ruby* utility.

To write the runner configuration:

1. Open the `worker.json` file in your `Booking` project.
2. Add the following JSON configuration data to the file:

```
{
  "domain":
    {
      "name": "Booking",
      "retention_in_days": 10
    },
  "workflow_workers": [
    {
      "number_of_workers": 5,
      "task_list": "booking_tasklist"
    }
  ],
  "activity_workers": [
    {
      "number_of_workers": 5,
      "number_of_forks_per_worker": 10,
      "task_list": "booking_activity_tasklist"
    }
  ]
}
```

Now, start the workers so they can begin polling for tasks.

To start the workers:

- Starting within the `Booking` directory, run the **`aws-flow-ruby`** utility:

```
aws-flow-ruby -f worker.json
```

The runner will provide you with some output that describes the process IDs of your worker threads:

```
waiting on workers [10972, 10975, ...] to complete
```

Your worker is now polling for tasks, but to provide it with tasks to process, you need to start a *workflow execution*.

2.4.6 Start a Workflow Execution

When executing a workflow instead of a single activity, use the start_workflow method instead of `start`.

To start the workflow execution:

1. Create a new file (you can call it `starter.rb`) and add the following code:

```
require 'aws/decider'

input = {
  request_id: "1234567890",
  customer_id: "1234567890",
  reserve_car: true,
  reserve_air: true
}

opts = {
  domain: "Booking",
  version: "1.0"
}

AWS::Flow::start_workflow("BookingWorkflow.make_booking", input, opts)
```

2. Open a command-line window and run your script using Ruby:

```
ruby starter.rb
```

The call to `start` takes:

- the fully-qualified name (*class.method*) of your workflow method
- input data for the workflow
- a block of StartWorkflowOptions

When you run the script, the `make_booking` workflow and its associated activities will begin running in the background on the previously-started workers.

You can view your workflow execution the same way as for *HelloWorld*, just be sure to select the *Booking* domain in the SWF console.

2.4.7 Next Steps

Use the following topics and resources to learn more about the framework:

- To learn more about the AWS Flow Framework for Ruby and about how Amazon SWF applications work, see *Flow Concepts*.
- To learn how to deploy workflows and workers with Elastic Beanstalk or AWS OpsWorks, see *Deploying Workflows with Elastic Beanstalk* and *Tutorial: Hello AWS OpsWorks!*.
- For more information about and examples of programming with the AWS Flow Framework for Ruby, see *Basic Workflow Programming* and *Advanced Topics*.
- For information about how you can use the AWS Flow Framework for Ruby with other AWS products, see *Working with Other AWS Products*.
- To view and download working examples that demonstrate many of the features and techniques described in this documentation, see the aws-flow-ruby-samples repository on GitHub.

CHAPTER 3

Flow Concepts

Throughout the AWS Flow Framework for Ruby documentation, you will find references to a number of code and conceptual terms specific to Amazon SWF and the flow framework. This section provides topics that discuss the various parts of an Amazon SWF application and other essential concepts that you should understand when designing Amazon SWF applications and workflows.

3.1 Parts of an Amazon SWF Application

An Amazon SWF application comprises various logical elements. Understanding these will help you determine how to build your own flow applications.

Contents

3.1.1 Domains

A *domain* is an identifier (name) that you create to hold workflow processes and data. When you register a workflow type or activity type, you associate it with a domain name, in which all of the workflow activity takes place. Workflows and activities can only communicate with workflows and activities that exist within the same domain, and task lists that are used within a domain are distinct from task lists that exist in a separate domain, even if the task list has the same name as one that is being used in the other domain.

When you register a domain, you provide it with a workflow retention period, which is the minimum number of days that workflow history is retained for closed workflow executions within that domain.

Registering a domain is optional—The AWS Flow Framework for Ruby provides a default domain, `FlowDefault`, which it uses for workflow executions that are started without specifying a domain name to use. The default domain has a retention period of 7 days.

To learn how to register and deprecate domains, see *Registering a Domain*.

3.1.2 Workflows

A *workflow* is the primary element in all Amazon SWF applications. It represents a sequence of steps required to perform a specific task. The steps needn't be strictly sequential; a workflow can consist of tasks that run sequentially, in parallel, synchronously or asynchronously. How your workflow behaves depends largely upon your business logic—the steps that are required to complete a process.

Because workflows contain code that responds to events that are managed by the Amazon SWF service, making decisions about what steps to take and how workflow execution proceeds, a workflow is also commonly referred to as a *decider*. Workflows are also responsible for passing data from and to any activities and child workflows that it runs.

The AWS Flow Framework for Ruby provides a *default decider* for you, so for simple, sequential workflows, you may not need to write any workflow code yourself. For a very simple example of a AWS Flow Framework for Ruby application that uses a default decider, see the *Hello World* topic.

A workflow consists of two parts: a workflow *type registration* and a workflow *implementation*:

- When you *register* a workflow, you provide a name, a version, and a set of options that provide default settings. These settings are applied by default to any workflow that uses the same workflow name and version.
- The workflow *implementation* consists of the code that provides your business logic. This is the part of the workflow that is specifically referred to as the decider. Workflow code is associated with a workflow type registration, but you can use the same code for different workflow types: the registered workflow type controls the default options that will be applied to the workflow when it's run.

3.1.3 Activities

An *activity* represents a step, or single unit of work, in a workflow. An activity can calculate a value based on input data, receive input from a web application, wait for a *human task* to be completed, or perform any other action that represents a step in your workflow.

Similar to *workflows*, an activity consists of an activity *type registration*, uniquely identified by a name and version and which provides default options, and an activity implementation which provides the code that will be executed when the activity is run.

Activities are scheduled by a *workflow implementation*, in response to decision tasks received from the workflow's *task list*.

See *Implementing Activities* to learn how to implement activities with the AWS Flow Framework for Ruby.

3.1.4 Task Lists

A task list is a logical entity used by Amazon SWF to manage events for your workflows and activities. When you register a workflow or activity, you can provide it with a task list name that can be referred to in order to receive tasks for that workflow or activity.

Workflow and activity tasks are polled for separately, even if they use the same task list name. Workflow tasks, for example, are delivered only to pollers that exist within your workflow code, and activity tasks are delivered only to your activities.

3.1.5 Workers

Workflow and Activity workers are responsible for receiving tasks from Amazon SWF and in taking appropriate actions to start a workflow or schedule an activity to be run. They are each configured with a task list to poll on.

With the AWS Flow Framework for Ruby, you can start workers using the ActivityWorker and WorkflowWorker classes, or by using the **`aws-flow-ruby`** command-line utility to spawn a number of workers when provided with activity and workflow classes.

Your workers will not begin receiving workflow or activity task events until a *workflow execution* is started.

For more information about starting workers, see *Starting Workflow and Activity Workers*.

3.1.6 Workflow Execution

A *workflow execution* refers to an individual execution of a workflow using a workflow_client's WorkflowClient#start_execution method (or by any other means, such as starting a workflow from the command line or using the AWS Management Console).

Once you begin executing a workflow, your *workers* will begin receiving task events from Amazon SWF.

3.2 Amazon SWF Timeout Types

This topic provides information about the various *timeouts* that you can set in your workflows and activities to control your workflow behavior.

Contents

3.2.1 About Amazon SWF timeouts

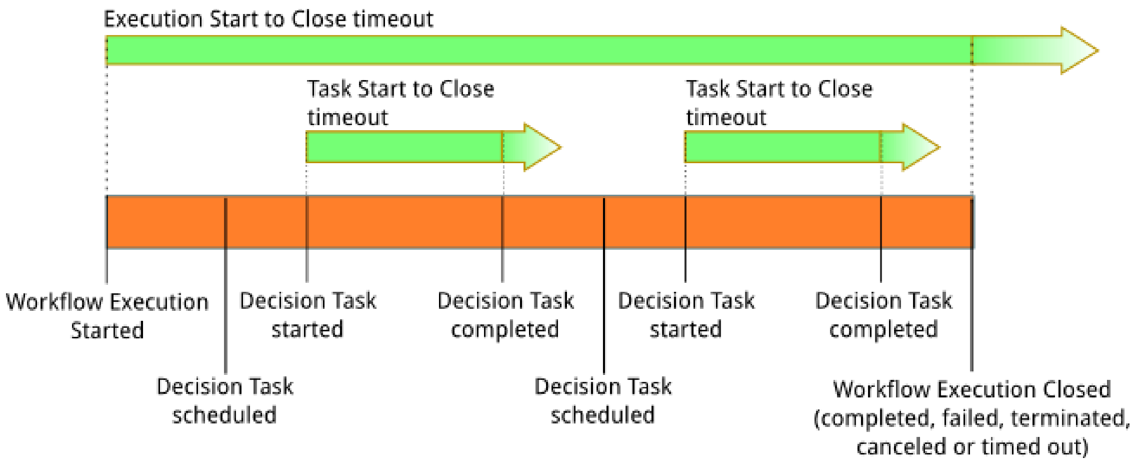
To ensure that workflow executions run correctly, Amazon Simple Workflow Service enables you to set different types of timeouts. Some timeouts specify how long the workflow can run in its entirety. Other timeouts specify how long activity tasks can take before being assigned to a worker and how long 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.

3.2.2 Timeouts for Workflows and Workflow Executions

The following diagram shows how workflow execution and workflow (decider) timeouts are related to the lifetime of a workflow:


Execution Start to Close timeout
Task Start to Close timeout
Task Start to Close timeout
Workflow Execution Started
Decision Task scheduled
Decision Task started
Decision Task completed
Decision Task scheduled
Decision Task started
Decision Task completed
Workflow Execution Closed (completed, failed, terminated, canceled or timed out)

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

Execution 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 cannot take any action on it other than visibility calls.

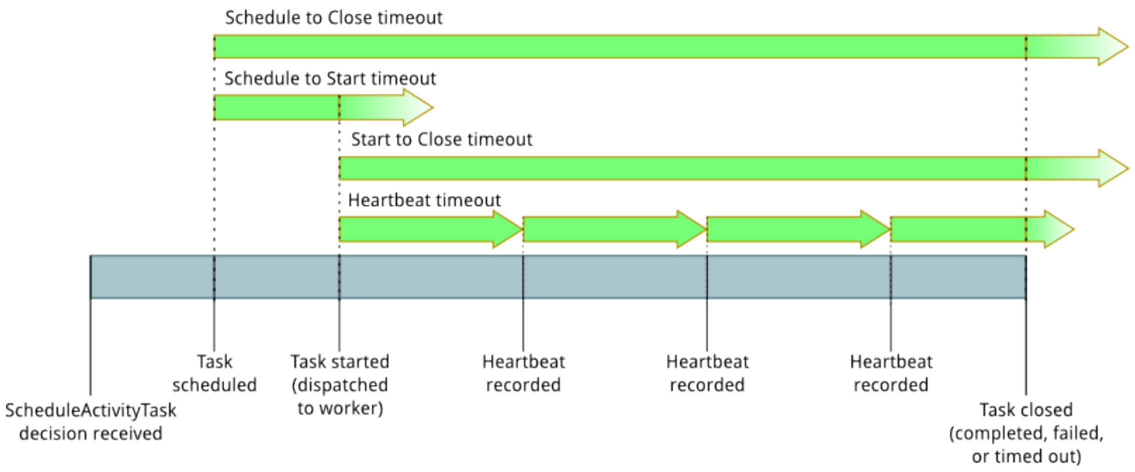
Task Start to Close This timeout specifies the maximum time that the corresponding decider can take to complete a decision task. It is set during workflow type registration. If this timeout is exceeded, the task is marked as timed out in the workflow execution history, and Amazon SWF adds an event of type DecisionTaskTimedOut to the workflow history.

The event attributes will include the IDs for the events that correspond to when this decision 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 this decision task timed out.

After this timeout occurs, an attempt to complete the timed-out decision task using RespondDecisionTaskCompleted will fail.

3.2.3 Timeouts for Activities

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


Schedule to Close timeout
Schedule to Start timeout
Start to Close timeout
Heartbeat timeout
ScheduleActivityTask decision received
Task scheduled
Task started (dispatched to worker)
Heartbeat recorded
Heartbeat recorded
Heartbeat recorded
Task closed (completed, failed, or timed out)

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

Activity Task 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 This timeout specifies the maximum time that a task can run before providing its progress through the RecordActivityTaskHeartbeat action.

Activity Task 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 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.

CHAPTER 4

Basic Workflow Programming

This section covers the basics of workflow programming, describing how to register domains, program activities and workflows, start task pollers, how to start a workflow execution, and how to set options. You can find further information about programming with the AWS Flow Framework for Ruby in the *Advanced Topics* section.

Note: In addition to the examples provided within these topics, code samples that demonstrate many of the features discussed here can be found in the AWS Flow Framework for Ruby samples and recipes repository, available at:

- https://github.com/awslabs/aws-flow-ruby-samples

4.1 Registering a Domain

To register a domain with the AWS Flow Framework for Ruby, use the underlying AWS SDK for Ruby. When you register a domain, you must provide Amazon SWF with the domain's name and retention period, measured in days. The *retention period* is the number of days that workflow execution history will be retained for closed workflows.

Here's a typical method that either retrieves an existing domain, or registers it if the domain name has not yet been registered:

```ruby
require 'aws/decider'

# get a SWF object from the AWS Ruby SDK.
swf = AWS::SimpleWorkflow.new

# attempt to retrieve a domain. If it doesn't already exist, then register it.
domain = swf.domains['ExampleDomain']
unless domain.exists?
  domain = swf.domains.create('ExampleDomain', 10)
end
```

You can also use the *aws-flow-ruby* utility to register a domain—if you specify a domain in its worker configuration file that doesn't yet exist, the AWS Flow Framework for Ruby will attempt to register it for you.

Note: When using the AWS Flow Framework for Ruby, registering a domain is *optional*. If you don't declare a domain to use for your workflows, the framework will use the default domain, `FlowDefault`, with a retention period of 7 days.

For more information about registering domains with the AWS SDK for Ruby, see SWF::Client#register_domain in the AWS SDK for Ruby Reference.

4.1.1 Deprecating a Registered Domain

If you have registered a domain name and you would like to stop any new workflows from being created in it, you can *deprecate* a registered domain. However, once you deprecate a domain:

- You can no longer run any workflows within it.
- You cannot re-register the domain within the same region and using the same account as the deprecated domain.

Given these caveats, you can deprecate a domain by using the AWS::SimpleWorkflow::Client#deprecate_domain method:

```
require 'aws/decider'

# get a SWF object from the AWS Ruby SDK.
swf = AWS::SimpleWorkflow.new

# deprecate the domain
swf.client.deprecate_domain({ name: 'ExampleDomain' })
```

4.2 Implementing Activities

All *activities* in that are run with Amazon SWF use a registered activity type to identify the activity and set its default options. You can have the AWS Flow Framework for Ruby register the activity for you, or you can do this yourself. Either way, the primary attribute of any activity is the code that is run when the activity is run.

Contents

4.2.1 An Activity Implementation is a Ruby Method

In *Hello World*, an activity was defined simply by creating an enclosing class and defining an activity method:

```
class HelloWorld
  def hello(input)
    "Hello #{input[:name]}!"
  end
end
```

The activity takes an input parameter that can receive data supplied to it by Amazon SWF when the activity is run.

Activities that are implemented this way are ideal if you're running a single activity with a default decider and default activity type. For more control over your activity type registration and to set activity options, you should base your activities classes on the Activities class, such as this set of activities from the Booking sample:

```
# BookingActivity class defines a set of activities for the Booking sample.
class BookingActivity
  extend AWS::Flow::Activities

  # The activity method is used to define activities. It accepts a list of names
  # of activities and a block specifying registration options for those
  # activities
  activity :reserve_car, :reserve_air, :send_confirmation do
    {
      version: "1.0",
    }
  end

  # This activity can be used to reserve a car for a given request_id
  def reserve_car(request_id)
    puts "Reserving car for Request ID: #{request_id}\n"
  end

  # This activity can be used to reserve a flight for a given request_id
  def reserve_air(request_id)
    puts "Reserving airline for Request ID: #{request_id}\n"
  end

  # This activity can be used to send a booking confirmation to the customer
  def send_confirmation(customer_id)
    puts "Sending notification to customer: #{customer_id}\n"
  end
end
```

This class defines three activities and sets the same *activity registration options* for each. In your own `Activities`-based classes, you can use this technique to set the same options for multiple activity methods, or you can provide separate options for each activity.

4.2.2 Activity Registration

Amazon SWF must know about your activity type in order to process tasks for it; *registering an activity type* provides the activity's name, version, and default options to Amazon SWF so that the activity can be referenced and run in your workflows.

Activities that you define using the AWS Flow Framework for Ruby are automatically registered by the framework when necessary. Activities that have already been registered are used when they are referenced in your code, and any activities that are not yet registered will be registered for you by the framework when your code is run for the first time.

When the AWS Flow Framework for Ruby registers an activity type for you, its *name* is taken to be a combination of the activity's class and method names. For example, the `reserve_car` method defined in the Booking example's `BookingActivities` class will be named `BookingActivities.reserve_car` in your workflow history.

Whether you define one or not, *version* of an activity is required by Amazon SWF, and is either automatically assigned by the framework (in the case that you define an activity to run as in the Hello World sample) or can be set using an the *version* registration option when the activity is declared in your `Activities`-based class, as with the Booking sample. When an activity version is automatically applied, the default value of `1.0` is used. Activity versions are not restricted to numeric values: "1.0", "1.2a", and "version_three" are all valid version fields.

Like all Amazon SWF types, activity types are scoped to a particular *domain*, AWS account, and region. Activities that are registered in other domains, regions, or to another account are unrelated, even if they share the same name, version or other options.

Within a domain, region and account, an activity type is uniquely identified by the combination of its *name* and *version*. Once registered, an activity type is *immutable*: Any changes you make to an activity's default options must be accompanied by either a change to its name, its version, or both.

4.2.3 Scheduling and Running Activities

Activities are scheduled to be run within your workflow implementation, also known as your workflow's *decider* logic. The AWS Flow Framework for Ruby provides a default decider that can run a single activity that you provide to the start method:

```
require 'aws/decider'
AWS::Flow::start("HelloWorld.hello", { name: "AWS Flow Framework!" })
```

When working with multi-step workflows, you will often want to write the decider logic yourself. The Booking sample implements a *synchronization*-pattern workflow by scheduling two activities asynchronously, and then waiting for all of the futures to be set before completing the workflow:

```
require 'flow/activities'

# BookingWorkflow class defines the workflows for the Booking sample
class BookingWorkflow
  extend AWS::Flow::Workflows

  # Use the workflow method to define workflow entry point.
  workflow :make_booking do
    {
      version: "1.0",
      default_execution_start_to_close_timeout: 120
    }
  end

  # Create an activity client using the activity_client method to schedule
  # activities
  activity_client(:client) { { from_class: "BookingActivity" } }

  # This is the entry point for the workflow
  def make_booking options

    puts "Workflow has started\n" unless is_replaying?
    # This array will hold all futures that are created when asynchronous
    # activities are scheduled
    futures = []

    if options[:reserve_car]
      puts "Reserving a car for customer\n" unless is_replaying?
      # The activity client can be used to schedule activities
      # asynchronously by using the send_async method
      futures << client.send_async(:reserve_car, options[:request_id])
    end
    if options[:reserve_air]
      puts "Reserving air ticket\n" unless is_replaying?
      futures << client.send_async(:reserve_air, options[:customer_id])
    end

    puts "Waiting for reservation to complete\n" unless is_replaying?
    # wait_for_all is a flow construct that will wait on the array of
    # futures passed to it
    wait_for_all(futures)

    # After waiting on the reservation activities to complete, the workflow
    # will call the send_confirmation activity.
    client.send_confirmation(options[:customer_id])

    puts "Workflow has completed\n" unless is_replaying?
  end

  # Helper method to check if Flow is replaying the workflow. This is used to
  # avoid duplicate log messages
  def is_replaying?
    decision_context.workflow_clock.replaying
  end
end
```

Note: For more information about implementing workflow patterns and about writing asynchronous workflows, see the topics *Implementing Workflow Patterns* and *Executing Tasks Asynchronously*, respectively.

Activities are run when an *activity worker* that is polling for activity tasks receives an activity task event from Amazon SWF for a particular activity, runs the activity and then reports the result back to Amazon SWF.

You can *code activity workers* yourself, or you can use the **`aws-flow-ruby`** utility to spawn workers that will automatically run activities for you. Information about how to use each method is provided in the linked topics.

4.2.4 For More Information

More information and further examples of activity implementation, registration and scheduling, refer to the following topics and resources:

- *Specifying Workflow and Activity Options* – Provides information about setting activity options during registration or when scheduling an activity.
- *Amazon SWF Timeout Types* – Provides information about timeouts for activities and what they mean in the context of the activity's life-cycle.
- *Implementing Workflow Patterns* – Provides information about how to design your decider code to replicate many common workflow patterns.
- *Setting Task Priority* – Provides information about how to set a task priority value to your activities to affect which activity tasks are delivered to your workers first.
- *aws-flow-ruby* – Provides information about how to set up and spawn workers for your activities and workflows with a simple configuration file and the **`aws-flow-ruby`** utility.
- awslabs/aws-flow-ruby-samples – A GitHub repository with examples and recipes that provide code examples of activity and workflow implementations using the AWS Flow Framework for Ruby.

4.3 Running Activities

If you need to run *only a single activity* at a time, you don't need to write any workflow code—you can use the start method to automatically create a workflow and start a workflow execution to run your activity.

Note: The `start` method can run only one activity per workflow execution. If your workflow consists of more than one activity, create decider methods of your own. For more information, see *Implementing Workflows*.

To run an activity

1. Write an activity method that takes one parameter, the activity *input*. The activity method must reside within a class. For example:

```
    class HelloWorld
      def hello(input)
        "Hello #{input[:name]}!"
      end
    end
```

2. Call the start method, providing it with the activity's name (a combination of its class name and method name, joined by a period) and a block of optional input data to pass to the activity. For example, to run the `hello` activity method and provide it with input:

```
    require 'aws/decider'
    AWS::Flow::start("HelloWorld.hello", { name: "AWS Flow Framework!" })
```

The `start` method will register your activity if necessary, create and register a workflow to run it, and will start a workflow execution.

If you do not provide any activity options to `start`, it will use the following defaults:

Option	Value
version	"1.0"
data_converter	YAMLDataConverter
exponential_retry	{ maximum_attempts: 3 }
start_to_close_timeout	"NONE"
schedule_to_close_timeout	"NONE"
schedule_to_start_timeout	"NONE"
heartbeat_timeout	"NONE"
task_list	"USE_WORKER_TASK_LIST"

You can modify any of these activity options by passing them as a block after the input data when you call `start`:

```
require 'aws/decider'
AWS::Flow::start("HelloWorld.hello", { name: "AWS Flow Framework!" }) {
  {
    heartbeat_timeout: 10,
    task_priority: 500,
  }
}
```

4.3.1 For More Information

Use the following topics and resources to learn more about running activities:

- *Hello World* – A basic tutorial that leads you through the process of creating and launching an activity using the `start` method.
- *Implementing Activities* – Provides more information about writing activity code.
- *Specifying Workflow and Activity Options* – Provides information about the options that can be set on activities and how to set them.
- *Retrying Failed Tasks* – Shows how to run activities that are automatically retried when they fail.

4.4 Implementing Workflows

A *workflow* represents the path of execution required to perform a sequence of tasks, which are usually activities, but which can also be child workflows (which might also have activities and child workflows of their own).

In some cases, you don't need to implement your own workflow. If you would like to run a single activity, you can use the start method to run a single activity using a default workflow. For more information, see the *Hello World* tutorial for an example, and *Running Activities* for detailed information about the AWS Flow Framework for Ruby-supplied default decider.

Contents

4.4.1 A Decider Implementation Defines Your Workflow

At the center of a workflow implementation is your *decider* logic. Similarly to activities, you provide a workflow implementation by declaring a class based on the Workflows class and provide methods that define your deciders. For example, here is the implementation of the workflow used in the Booking sample:

```
require 'flow/activities'

# BookingWorkflow class defines the workflows for the Booking sample
class BookingWorkflow
  extend AWS::Flow::Workflows

  # Use the workflow method to define workflow entry point.
  workflow :make_booking do
    {
      version: "1.0",
      default_execution_start_to_close_timeout: 120
    }
  end

  # Create an activity client using the activity_client method to schedule
  # activities
  activity_client(:client) { { from_class: "BookingActivity" } }

  # This is the entry point for the workflow
  def make_booking options

    puts "Workflow has started\n" unless is_replaying?
    # This array will hold all futures that are created when asynchronous
    # activities are scheduled
    futures = []
    if options[:reserve_car]
      puts "Reserving a car for customer\n" unless is_replaying?
      # The activity client can be used to schedule activities
      # asynchronously by using the send_async method
      futures << client.send_async(:reserve_car, options[:request_id])
    end
    if options[:reserve_air]
      puts "Reserving air ticket\n" unless is_replaying?
      futures << client.send_async(:reserve_air, options[:customer_id])
    end

    puts "Waiting for reservation to complete\n" unless is_replaying?
    # wait_for_all is a flow construct that will wait on the array of
    # futures passed to it
    wait_for_all(futures)

    # After waiting on the reservation activities to complete, the workflow
    # will call the send_confirmation activity.
    client.send_confirmation(options[:customer_id])

    puts "Workflow has completed\n" unless is_replaying?
  end

  # Helper method to check if Flow is replaying the workflow. This is used to
  # avoid duplicate log messages
  def is_replaying?
    decision_context.workflow_clock.replaying
  end
end
```

This example defines a single decider, `make_booking`, which declares an activity client using the Workflows#activity_client method to schedule activities with. The activity client takes a set of ActivityOptions that it will use when an activity is scheduled using the client.

Your decider can schedule activities synchronously or asynchronously, can spawn child workflows, and can perform many other functions to allow you to customize how your workflow progresses. While you design your workflow classes and decider methods, keep the following points in mind:

- *Do not use decider methods to perform long-running tasks.* The AWS Flow Framework for Ruby replay mechanism will repeat that task multiple times. Even asynchronous workflow methods will typically run more than once. Instead, use *activities* for long running tasks; the replay mechanism executes activities only once.
- *Your workflow logic must be completely deterministic.* Every episode (a single replay of the workflow) must take the same control flow path. For example, the control flow path should not depend on the current time.

4.4.2 Registering Workflows

Amazon SWF workflows are represented by a workflow type that is registered with Amazon SWF. As with *activities*, the AWS Flow Framework for Ruby handles workflow registration automatically for your workflow types when necessary.

Workflow types registered by the framework are named using a combination of the workflow class name and decider method name; in the Booking example, the workflow type registered for the `make_booking` decider method will be `BookingWorkflow.make_booking`.

You can set default options for workflow types when you declare decider methods in your Workflows-based class:

```
workflow :make_booking do
  {
    version: "1.0",
    default_execution_start_to_close_timeout: 120
  }
end
```

The block of options associated with the *make_booking* declaration are used as default options whenever the workflow is run, unless they are overridden when launching the workflow. As with activities, workflow types are *immutable* once registered, so if you need to change the default options for a workflow, you will also need to change its name, version or both in order to keep it from interfering with workflows associated with the previously-registered type.

When using the *default decider*, the AWS Flow Framework for Ruby will register and use its own workflow type, named `RubyFlowDefaultWorkflow.start`, with a version number of `1.0`. You cannot redefine or change the default options associated with this workflow type.

4.4.3 Launching and Running Workflows

Your decider code won't be run until:

1. A workflow execution is started.
2. A workflow worker receives a decision task to start the workflow.

You can start a workflow worker that polls for decision tasks by *implementing one yourself* or by using the **aws-flow-ruby** to spawn workers for you. Information about how to use each method is provided in the linked topics.

To start a workflow execution, you can use the start_workflow method, providing it with your registered workflow name, a block of input data for your workflow, and a set of WorkflowOptions used to start the workflow. For example:

```
require 'aws/decider'

input = {
  request_id: "1234567890",
  customer_id: "1234567890",
  reserve_car: true,
  reserve_air: true
}

opts = {
  domain: "Booking",
  version: "1.0"
}

AWS::Flow::start_workflow("BookingWorkflow.make_booking", input, opts)
```

4.4.4 For More Information

- *Specifying Workflow and Activity Options* – Provides information about setting workflow options during registration or when launching a workflow.
- *Amazon SWF Timeout Types* – Provides information about timeouts for workflows and what they mean in the context of the workflow's life-cycle.
- *Implementing Workflow Patterns* – Provides information about how to design your decider code to replicate many common workflow patterns.
- *Setting Task Priority* – Provides information about how to set a task priority value to your workflows to affect which decider tasks are delivered to your workers first.
- *aws-flow-ruby* – Provides information about how to set up and spawn workers for your workflows and activities with a simple configuration file and the **`aws-flow-ruby`** utility.
- awslabs/aws-flow-ruby-samples – A GitHub repository with examples and recipes that provide code examples of workflow and activity implementations using the AWS Flow Framework for Ruby.

4.5 Starting Workflow and Activity Workers

You can start workflow and activity workers easily using the **`aws-flow-ruby`** utility, or in code.

Contents

4.5.1 About Workers

Workflows and Activities are run by workflow and activity *workers*. A worker is responsible for polling for tasks from Amazon SWF on a *task list*, then starting the appropriate workflow or activity based on the message in the task event.

The AWS Flow Framework for Ruby takes care of managing the workers for you. All you need to do is instantiate and start the workers, passing optional configuration data to control how the workers operate.

You can start the workers in your Ruby code, or start them by using the **aws-flow-ruby** utility. If you are planning on deploying fleets of workers using AWS OpsWorks, you should use **aws-flow-ruby**. For more information about using Amazon SWF with AWS OpsWorks, see *Tutorial: Hello AWS OpsWorks!*.

4.5.2 Using `aws-flow-ruby` to Start Workers

You can begin both activity and workflow workers by providing a small JSON configuration file to the **aws-flow-ruby** utility, also referred to as the *runner*. Here is an example configuration, `worker.json`, that configures a small fleet of workers for the Booking sample:

```
{
  "domain":
    {
      "name": "Booking",
      "retention_in_days": 10
    },
  "workflow_workers": [
    {
      "number_of_workers": 5,
      "task_list": "booking_tasklist"
    }
  ],
  "activity_workers": [
    {
      "number_of_workers": 5,
      "number_of_forks_per_worker": 10,
      "task_list": "booking_activity_tasklist"
    }
  ]
}
```

The runner interprets this file and creates a set of workers as you specify, using the activities and workflows that are defined in `activities.rb` and `workflows.rb` files within the `flow` directory where the configuration file exists. See *aws-flow-ruby* for a complete description of how to configure the runner.

4.5.3 Starting Activity Workers in Code

To start activity workers in your code:

1. Create a new ActivityWorker object, providing it with a AWS::SimpleWorkflow::Client object, the *domain*, the *task list* name to poll for activity tasks on, and an Activities-based class to use to access its activity implementations. For example:

```
    require 'aws/decider'

    swf = AWS::SimpleWorkflow.new

    activity_worker = AWS::Flow::ActivityWorker.new(
      swf.client, "HelloWorldDomain", "hello_world_activity_tasks",
      HelloWorldActivities)
```

2. Call `start` on the `ActivityWorker`. You can set whether or not the activities should be registered first by using the *should_register* parameter.

```
activity_worker.start(true)
```

4.5.4 Starting Workflow Workers in Code

To start workflow workers in your code:

1. Create a new WorkflowWorker object, providing it with similar options as with an `ActivityWorker`: a AWS::SimpleWorkflow::Client object, the *domain*, the *task list* name to poll for workflow tasks on, and a Workflows-based class to use to access its activity implementations. For example:

```
require 'aws/decider'

swf = AWS::SimpleWorkflow.new

workflow_worker = AWS::Flow::WorkflowWorker.new(
  swf.client, "HelloWorldDomain", "hello_world_decision_tasks",
  HelloWorldWorkflow)
```

2. Call `start` on the `WorkflowWorker`. You can set whether or not the workflows should be registered first by using the *should_register* parameter.

```
workflow_worker.start(true)
```

4.5.5 For More Information

For more information about implementing workers, refer to the following topics and resources:

- *aws-flow-ruby* – Provides information about how to set up and spawn workers for your activities and workflows with a simple configuration file and the **`aws-flow-ruby`** utility.
- *Tutorial: Hello AWS OpsWorks!* – A tutorial that leads you through the process of spawning SWF worker fleets with AWS OpsWorks.
- awslabs/aws-flow-ruby-samples – A GitHub repository with examples and recipes that provide code examples of activity and workflow implementations using the AWS Flow Framework for Ruby.

4.6 Starting a Workflow Execution

Once you have defined your workflows and activities, have registered them with Amazon SWF and have started activity and workflow workers, your workflow is ready to run. However, until you start a workflow execution, Amazon SWF will not begin delivering tasks to your workers.

You have a number of options when starting a workflow with the AWS Flow Framework for Ruby:

1. If you are running a *single activity* only, use `AWS::Flow#start`.

2. If you are running *multiple activities*, use `AWS::Flow#start_workflow`.
3. Use a workflow client to start a workflow execution with `AWS::Flow::WorkflowClient#start_execution`.

Since most workflows use multiple activities, this topic will focus on the final two methods. For information about using the `start` method to run a single activity, see *Hello World* and *Running Activities*.

Contents

4.6.1 Starting a Workflow Execution with start_workflow

To start a workflow execution, the preferred method is to use start_workflow, which can be run simply by passing it the workflow name (*class.method*), input data, and a hash of StartWorkflowOptions:

```ruby
require 'aws/decider'

AWS::Flow::start_workflow(
  "BookingWorkflow.make_booking",
  {
    request_id: "1234567890", customer_id: "1234567890",
    reserve_car: true, reserve_air: true,
  },
  {
    domain: "Booking",
    version: "1.0a"
  }
)
```

Important: The *domain* and *version* options are *required*.

You can also pass the workflow class as the first argument, and specify the method to run in the passed-in options. This is equivalent to the preceding method:

```ruby
require 'aws/decider'

AWS::Flow::start_workflow(
  "BookingWorkflow",
  {
    request_id: "1234567890", customer_id: "1234567890",
    reserve_car: true, reserve_air: true,
  },
  {
    domain: "Booking",
    execution_method: "make_booking",
    version: "1.0a"
  }
)
```

Use the *from_class* option to use options set in the workflow class:

```
require 'aws/decider'
require_relative 'flow/workflows.rb'

AWS::Flow::start_workflow(
  "BookingWorkflow",
  {
    request_id: "1234567890", customer_id: "1234567890",
    reserve_car: true, reserve_air: true,
  },
  {
    domain: "Booking",
    from_class: "BookingWorkflow"
  }
)
```

4.6.2 Starting a Workflow Execution with a WorkflowClient

In addition to using `start_workflow`, you can start a workflow execution by using a WorkflowClient object.

To start a workflow execution:

1. Get a WorkflowClient object by calling workflow_client and providing it with an AWS::SimpleWorkflow object, a AWS::SimpleWorkflow::Domain object, and an optional block of StartWorkflowOptions:

```
require 'aws/decider'

swf = AWS::SimpleWorkflow.new
domain = swf.domains['HelloWorldDomain']

workflow_client = AWS::Flow::workflow_client(swf.client, domain) {
  { task_list: "hello_world_decision_tasks" }
}
```

2. Use the WorkflowClient#start_execution method, passing it optional input data for the workflow, and a hash of WorkflowOptions:

```
workflow_input = "Amazon SWF"
workflow_client.start_execution(
  workflow_input, { {  workflow_name: 'my_workflow_execution_name' } } )
```

Once you've started the workflow execution, your workflow and activity pollers will begin receiving events on their respective task lists.

4.6.3 For More Information

- *Specifying Workflow and Activity Options* – Provides information about setting options on a workflow client or when starting a workflow execution.
- awslabs/aws-flow-ruby-samples – A GitHub repository with examples and recipes that provide working code using the AWS Flow Framework for Ruby.

4.7 Specifying Workflow and Activity Options

The AWS Flow Framework for Ruby allows you to set options that affect how your workflows and activities are run. This topic lists each of the options that you can set, as well as providing detail about how, and when, you can set them.

Contents

4.7.1 Activity Registration Options

The following registration options can be set only when *declaring an activity type*. Workers will use these values to register the type with Amazon SWF. Once these values are set, you must set a different version (essentially declaring a new type) if you want to change any of the registration options.

default_task_heartbeat_timeout The time, in seconds, within which an activity must record a heartbeat progress notification (by calling `record_activity_heartbeat` in the ActivityExecutionContext class).

Optional. The default value is `"NONE"`, which will allow tasks to run indefinitely before reporting progress.

default_task_list The name of the task list used for this activity type.

Optional. The default value is `"USE_WORKER_TASK_LIST"`, a restricted string that, when used, will cause the activity to use the same task list that the activity worker is polling on.

default_task_priority The task priority to set, from `-2147483647` to `2147483647`, where higher numbers indicate higher priority. Higher-priority tasks are delivered before lower-priority tasks on the same task list. Tasks that are not assigned a priority are given the default value of 0. For more information, see *Setting Task Priority*.

Optional. The default value is `0`.

default_task_schedule_to_close_timeout The maximum duration, in seconds, of an activity execution from the time it is scheduled to when it is marked as complete.

Optional. The default value is `"NONE"`; the activity has no restriction on when it must complete after being scheduled.

default_task_schedule_to_start_timeout The maximum duration, in seconds, from the time when the activity is scheduled to when it starts.

Optional. The default value is `"NONE"`; the activity has no restriction on when it must start after being scheduled.

default_task_start_to_close_timeout The maximum duration, in seconds, of the activity execution from the time it starts to when it is marked as complete.

Optional. The default value is `"NONE"`; the activity has no restriction on when it must complete after being started.

version The activity version to use. This value can be set *only* when declaring a activity type.

Required. You must always set `version` when you register a activity type, or when changing any activity registration options.

4.7.2 Workflow Registration Options

The following registration options can be set only when *declaring a workflow type*. Workers will use these values to register the type with Amazon SWF. Once these values are set, you must set a different version (essentially declaring a new type) if you want to change any of the registration options.

default_child_policy The optional policy to use for the child workflow executions when a workflow execution of this type is terminated.

Optional. The default value is `"TERMINATE"`, which will automatically terminate all child executions when the parent workflow is terminated.

default_execution_start_to_close_timeout The maximum duration, in seconds, of a workflow execution from the time it starts to when it is marked as complete.

Required. There is no default value. You must set this value either during registration or when executing the workflow.

default_task_list The name of the default task list used for this workflow type.

Optional. The default value is `"USE_WORKER_TASK_LIST"`, a restricted string that, when used, will cause the workflow to use the same task list that the workflow worker is polling on.

default_task_priority The task priority to set, from `-2147483647` to `2147483647`, where higher numbers indicate higher priority. Higher-priority tasks are delivered before lower-priority tasks on the same task list. Tasks that are not assigned a priority are given the default value of 0. For more information, see *Setting Task Priority*.

Optional. The default value is `0`.

default_task_start_to_close_timeout The maximum duration, in seconds, of a workflow task from the time it starts to when it is complete.

Optional. The default value is `30`.

version The workflow version to use. This value can be set *only* when declaring a workflow type.

Required. You must always set `version` when you register a workflow type, or when changing any workflow registration options.

4.7.3 Activity Runtime Options

These options can be set when *declaring an activity*, *initializing a new activity client* or when *scheduling an activity*. They will override any default options with the same name.

data_converter The data converter class to use to interpret data delivered from Amazon SWF. If not specified, then YAMLDataConverter will be used by default.

heartbeat_timeout The time, in seconds, within which an activity must record a heartbeat progress notification (by calling `record_activity_heartbeat` in the ActivityExecutionContext class).

input Input data that will be passed to the activity when it starts. You can also pass input directly as a parameter when scheduling the activity.

manual_completion Set to `true` when you have a *human task* (an activity that will be completed manually). In this case, the activity will return immediately after starting, but it will not complete automatically when it returns.

schedule_to_close_timeout The maximum duration, in seconds, of an activity execution from the time it is scheduled to when it is marked as complete.

schedule_to_start_timeout The maximum duration, in seconds, from the time when the activity is scheduled to when it starts.

start_to_close_timeout The maximum duration, in seconds, of the activity execution from the time it starts to when it is marked as complete.

task_list The name of the task list used for this activity.

task_priority The task priority to set, from `-2147483647` to `2147483647`, where higher numbers indicate higher priority. Higher-priority tasks are delivered before lower-priority tasks on the same task list. Tasks that are not assigned a priority are given the default value of 0. For more information, see *Setting Task Priority*.

4.7.4 Workflow Runtime Options

These options can be set when *declaring a workflow type*, *initializing a new workflow client* or when *starting the workflow*. They will override default options with the same name.

child_policy The optional policy to use for the child workflow executions when a workflow execution of this type is terminated.

data_converter The data converter class to use to interpret data delivered from Amazon SWF. If not specified, then YAMLDataConverter will be used by default.

execution_method The workflow method to call when the workflow begins executing. By default, this method is defined when you use the `workflow` method in the Workflows class to register your workflows.

This option is not required; it is used only if you start a workflow using the `start_execution` method on the client in your workflow class. By default, the client will select the *first* defined workflow in that class.

This will not be used if you start a workflow execution by calling the workflow method directly from the client (for example, `workflow_client.workflow_a`) or by calling `send` (for example, `workflow_client.send(:workflow_a)`).

execution_start_to_close_timeout The maximum duration, in seconds, of a workflow execution from the time it starts to when it is marked as complete.

input Input data that will be passed to the workflow upon execution. You can also pass input directly as a parameter when starting the workflow.

tag_list A list of tags to associate with the workflow. This is an empty list by default.

task_list The name of the default task list used for this workflow.

task_priority The task priority to set, from `-2147483647` to `2147483647`, where higher numbers indicate higher priority. Higher-priority tasks are delivered before lower-priority tasks on the same task list. Tasks that are not assigned a priority are given the default value of 0. For more information, see *Setting Task Priority*.

task_start_to_close_timeout The maximum duration, in seconds, of a workflow task from the time it starts to when it is complete.

workflow_id An optional workflow ID. If you don't set it, the AWS Flow Framework for Ruby will choose one for you.

4.7.5 How to Set Options

You can set options for activities and workflows at the following times:

- *At type declaration* – when you declare a new type, you can specify default options that *will be used for all activities/workflows of that type unless options are overridden on the client or at scheduling*.
- *On an activity/workflow client* – if you set activity options on an activity client, then any activities that are scheduled and launched with that client will inherit the options that it holds. These will act as *overrides for any options set at type declaration*.
- *At scheduling* – when you schedule a workflow or activity for execution, you can specify options that will *override any that were set at type declaration or on the client*.

Setting Registration (Type) Options

To set options in activity or workflow declarations that will be used when the type is registered with Amazon SWF, pass them as a block when you declare the activity or workflow using the `activity` or `workflow` methods in Activities or Workflows, respectively.

Using this method, you can send the same block of options to activities that share settings. For example, to set activity registration options:

```
class BookingActivity
  extend AWS::Flow::Activities

  activity :reserve_car, :reserve_air, :send_confirmation do
    {
      version: "1.0",
      default_task_list: "activity_tasklist",
      default_task_schedule_to_start_timeout: 30,
      default_task_start_to_close_timeout: 30
    }
  end
```

Workflow registration options are set the same way:

```
class BookingWorkflow
  extend AWS::Flow::Workflows

  workflow :make_booking do
    {
      version: "1.0",
      default_task_list: "workflow_tasklist",
      default_execution_start_to_close_timeout: 120
    }
  end
```

Important: *Once an activity or workflow type is registered, its default (registration) options cannot be changed.* If you need to change the default options of a registered activity or workflow type, you will need to register a new type with either a different name or different version to differentiate it from other activity types.

Setting Both Registration and Runtime Options at Declaration

You can set *both* registration and runtime options during declaration. If you do this, then the registration options will be set as the *defaults* for that type (you will see them if you view the type in the console, and they will be used as defaults for any other clients that use the type).

Any runtime options that you set at declaration, however, will *immediately override* these type-defining default options, and will be automatically used when scheduling an activity or workflow of that type in your code, unless they are subsequently overridden on the client or when scheduling. You can freely modify runtime options at type declaration without any need to update the `version` value for the type.

In effect, you can declare a type with different runtime options in different parts of your code, as long as you don't change any of the type's *registration* options. If you do, you will need to also specify a new version.

Setting Options on the Client

If you set options when creating your workflow or activity clients, they will override any options set at type declaration. For example:

```
activity_client(:my_activity_client) {
  {
    heartbeat_timeout: 30,
    start_to_close_timeout: 300
  }
}
```

The same technique is used to set workflow client options:

```
swf = AWS::SimpleWorkflow.new
domain = swf.domains['MyDomain']

workflow_client = AWS::Flow::workflow_client(swf, domain) {
  {
    task_list: 'workflow_task_list',
    execution_start_to_close_timeout: 3600
  }
}
```

You can also use the *from_class:* attribute to copy options from another class. Any options that are set by the class you specify will override those set when the activity was declared (default options).

```
activity_client(:client) { { from_class: "BookingActivity" } }
```

Copying Client Options Using with_opts

You can use the `with_opts` method available in the GenericClient class to create a new client that copies options from an existing client, overriding them with options that are passed to the `with_opts` method in a hash.

Setting Options at Scheduling

If you set options when an activity is scheduled or when starting a workflow, the values will override those that are set on the client *and* any that were set at type declaration.

To set options during activity scheduling, pass the options block to the activity client's `schedule_activity` method or when calling the activity method directly from the client:

```
file_client.process_file(local_source, local_target) do
  { task_list: "new_activity_task_list" }
end
```

Setting workflow options when starting a workflow is similar:

```
workflow_client.start_execution() {
  {
    task_list: "new_workflow_task_list",
    tag_list: [ "orderinfo", "web" ]
  }
}
```

4.7.6 Setting Other Types of Options

While this topic has focused on options that you can set on workflows and activities, there are options for other classes in the AWS Flow Framework for Ruby. For information about setting these options, refer to the sections in which they are discussed:

- Options that can be set when retrying failed tasks are covered in *Retrying Failed Tasks*.

4.8 Handling Errors

The way you handle errors in AWS Flow Framework for Ruby depends on whether the workflow is *synchronous* or *asynchronous*.

Contents

4.8.1 Errors in Synchronous Workflows

If your activities or workflows are synchronous, you can use Ruby's standard `begin/rescue/ensure` pattern to handle errors. For example:

```
# from within a decider
begin
  my_activity_client.my_activity
rescue ActivityTaskTimedOutException => e
  # handle timeout
rescue ActivityTaskFailedException => e
  # handle failure
ensure
  # clean up and stuff
end
```

Tip: AWS Flow Framework for Ruby exception names are based on the event types specified in HistoryEvent in the *Amazon Simple Workflow Service API Reference*. You can predict the exception name by adding "Exception" to the end of the event type. For example, an exception in `TimerFired` will result in a `TimerFiredException`.

4.8.2 Errors in Asynchronous Workflows

If your activities or workflows are asynchronous, (using send_async), use the Core#error_handler core method, which is modeled after Ruby's `begin/rescue/ensure` pattern. Here is an example of its use:

```
error_handler do |t|
  t.begin do
    my_activity_client.send_async :my_activity
  end
  t.rescue ActivityTaskTimedOutException do |e|
    # handle timeout
  end
  t.rescue ActivityTaskFailedException do |e|
    # handle failure
  end
  t.ensure do
    # clean up and stuff
  end
end
```

You pass the `error_handler` method a block that takes a single argument (a Core::BeginRescueEnsure object). The `BeginRescueEnsure` class has three methods: `begin`, `rescue`, and `ensure`, which take parts of your error handling logic.

For more information about writing code for asynchronous tasks, see *Executing Tasks Asynchronously*.

4.8.3 Additional Error Handling Examples

The handle_error recipe in the public aws-flow-ruby-samples project on GitHub provides a number of examples of handling errors for both synchronous and asynchronous tasks.

4.9 Troubleshooting and Debugging Workflows

This section provides information about how to troubleshoot your workflow executions. It includes strategies for examining and replaying workflows, and lists some common causes of errors in workflow executions.

Contents

4.9.1 Examining Workflow Executions with the AWS Management Console

The first step in troubleshooting a workflow execution is to use the AWS Management 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.

To learn more about using the AWS Management Console with Amazon SWF, see Managing Your Workflow Executions in the Amazon SWF Developer Guide.

4.9.2 Using the WorkflowReplayer Class

The AWS Flow Framework provides a Replayer::WorkflowReplayer 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.

When you replay a workflow execution using `WorkflowReplayer`, it does not 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.

For example, the following code snippet can be used to replay a workflow execution:

```ruby
require 'replayer'

# Create an instance of the replayer with the required options
replayer = AWS::Flow::Replayer::WorkflowReplayer.new(
  domain: '<domain_name>',
  execution: {
    workflow_id: "<workflow_id",
    run_id: "<run_id>"
  },
  workflow_class: YourWorkflowClass
)

# Call the replay method with the replay_upto event_id number -
decision = replayer.replay(20)

puts decision.inspect
```

4.9.3 Common Causes of Errors in Workflow Executions

Unknown Resource Fault

Amazon SWF returns an unknown resource fault when you try to perform an operation on a resource that is not available. The common causes for this fault are:

- *You configure a worker with a domain that does not exist.* To fix this, first register the domain using the Amazon SWF console or with 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 AWS Management 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.

Non Deterministic Workflows

The implementation of your workflow must be deterministic. Some common mistakes that can lead to nondeterminism are:

- Use of the system clock
- Use of random numbers
- 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. 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 change the version string of the type by providing a new version when declaring your workflow or activity type.

When new versions of a workflow are deployed, you might have executions of the existing version that are still running. Therefore, you need to make sure that workers get tasks that match the correct version of your workflow and activities. One way to accomplish this is by using a different set of task lists for each version. For example, you can append the version string to the name of a task list. This ensures that tasks belonging to different versions of the workflow and activities are assigned to the appropriate workers.

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.

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 the workflow history contains the identity of the workflow worker that received the task. The id used by the AWS Flow Framework is of the form: *{processId}@{host name}*. Here is an example of the details for a `DecisionTaskStarted` event in the Amazon SWF console for a sample execution:

Event Timestamp	Mon Feb 20 11:52:40 GMT-800 2012
Identity	`2276@ip-0A6C1DF5`
Scheduled Event Id	33

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.

CHAPTER 5

Advanced Topics

This topic advanced workflow programming topics, such as setting task priority, programming workflow patterns, error handling, asynchronous programming, retrying workflows and troubleshooting. For basic information about programming with the AWS Flow Framework for Ruby, see *Basic Workflow Programming*.

Note: In addition to the examples provided within these topics, code samples that demonstrate many of the features discussed here can be found in the AWS Flow Framework for Ruby samples and recipes repository, available at:

- https://github.com/awslabs/aws-flow-ruby-samples

5.1 Setting Task Priority

By default, tasks on a task list are delivered based upon their *arrival time*: tasks that are scheduled first are run first. 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. Tasks with the same priority are ordered by arrival time.

You can set a task priority for both workflows and activities. A workflow's task priority does not 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").

Contents

5.1.1 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, use the *default_task_priority* option when declaring it:

```
workflow :priority_workflow do
  {
    default_task_list: "workflow_tasks",
    default_task_priority: 10,
    version: "1.0",
  }
end
```

You can override a workflow type's registered (default) task priority by setting *task_priority* when you start the workflow execution:

```
workflow_client.start_execution() {
  {
    task_list: "workflow_tasks",
    tag_list: [ "lowpriority" ],
    task_priority: -5
  }
}
```

5.1.2 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.

Just as with workflow types, to register an activity type with a default task priority use the *default_task_priority* option when declaring it:

```
activity :do_something_important do
{
  version: "1.5",
  default_task_list: "activity_list",
  default_task_priority: 10,
  default_task_schedule_to_start_timeout: 30,
  default_task_start_to_close_timeout: 30
}
end
```

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

```
important_activity_client.send_async(
  :do_something_important, { task_priority: 20 } )
```

5.1.3 For More Information

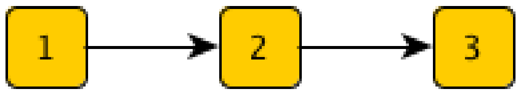
- *Specifying Workflow and Activity Options*

5.2 Implementing Workflow Patterns

This section demonstrates how to implement common workflow patterns using the AWS Flow Framework for Ruby. Much more information about commonly-used workflow patterns can be found on the Workflow Patterns page, presented by the Eindhoven and Queensland Universities of Technology.

Contents

5.2.1 Sequence

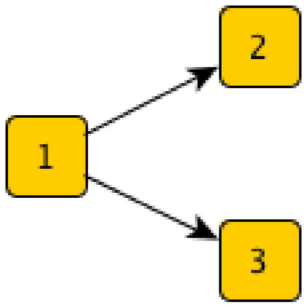


A *sequence* pattern refers to a workflow in which one task follows another in sequential order. It is implemented by calling activities synchronously:

```
client.activity1
client.activity2
client.activity3
```

Since each activity blocks execution of the main thread when it runs, `activity2` will run only after `activity1` has completed. Likewise, `activity3` won't run until after `activity2` is complete.

5.2.2 Parallel Split



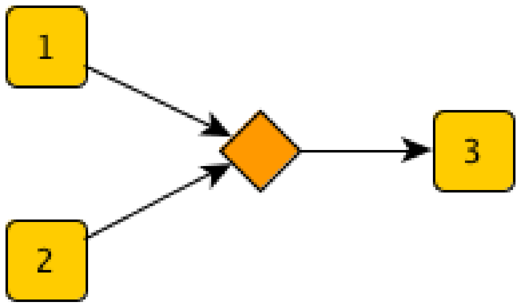
A *parallel split* pattern refers to a workflow in which one or more tasks are executed concurrently. It is implemented by calling activities asynchronously with GenericClient#send_async:

```
client.activity1
client.send_async(:activity2)
client.send_async(:activity3)
```

The `send_async` method launches a fiber to run the activity on and returns immediately. In this case, `activity3` will be run immediately, without waiting for `activity2` to complete. Likewise, execution of the main thread will continue without waiting for either `activity2` or `activity3` to complete.

If you want your main thread to wait for one or both activities to finish before proceeding, see *simple merge* or *synchronization*.

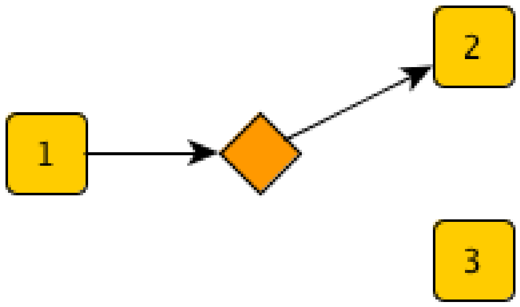
5.2.3 Synchronization



A *synchronization* pattern refers to a workflow in which the main thread waits for one or more concurrently-running tasks to complete before continuing. It is implemented by calling Core#wait_for_all with the futures that are returned from the activities that you want to synchronize:

```
first_future = client.send_async(:activity1)
second_future = client.send_async(:activity2)
wait_for_all(first_future, second_future)
client.activity3
```

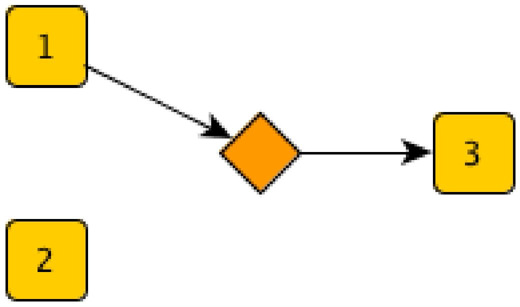
5.2.4 Exclusive Choice



An *exclusive choice* pattern refers to a workflow in which the results of one activity are used to select one subsequent activity to run from a set of two or more activities. It is implemented by choosing the activity to run based on the value returned by a predicate function acting on a decision value:

```
decision_value = client.activity1
if (predicate_function(decision_value))
  client.activity2
else
  client.activity3
end
```

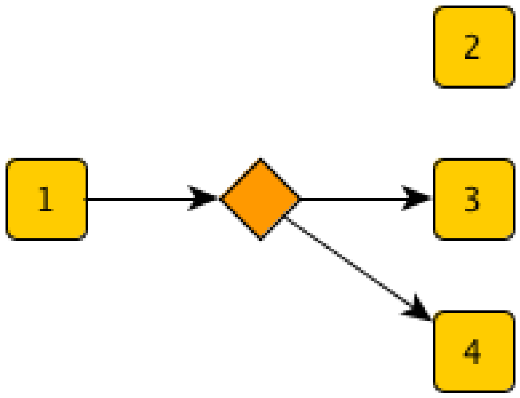
5.2.5 Simple Merge



A *simple merge* pattern refers to a workflow in which the completion of one or more activities triggers the next activity in the sequence. It is implemented by calling Core#wait_for_any with the futures that are returned from the activities that you want to merge:

```
first_future = client.send_async(:activity1)
second_future = client.send_async(:activity2)
wait_for_any(first_future, second_future)
client.activity3
```

5.2.6 Multi Choice



A *multi choice* pattern refers to a workflow in which the results of one activity are used to select one or more subsequent activities to run from a set of two or more activities. This is very similar to *exclusive choice*, but more than one branch in the workflow may be run. Like *exclusive choice*, it is commonly implemented with a predicate function that chooses one or more activities to run based on a decision value:

```
decision_value = client.activity1
if (predicate_function(decision_value))
  client.activity2
elsif (predicate_function2(decision_value))
  client.activity3
else
  client.send_async(:activity3)
  client.send_async(:activity4)
end
```

5.3 Executing Tasks Asynchronously

You can schedule tasks to run asynchronously in a number of different ways:

- Use the GenericClient class method `send_async` to schedule an asynchronous task using an activity or workflow client.
- Use the Core instance method `task` to execute any block of code asynchronously in the context of the AWS Flow Framework for Ruby.

Whichever method you use, the Framework will return an instance of the Core::Future class, which is used to determine when the asynchronous task has been completed.

Contents

5.3.1 Waiting for a Future

A Core::Future provides a mechanism for determining whether or not the task that it is tracking has completed. When the task is complete, the `Future` becomes *set*. The `Future` class itself provides three methods that can be used to determine when the task has completed:

- `get`, which blocks until the future is ready.
- `set?`, which returns `true` when the future is ready.
- `on_set`, which takes a callback block that will be executed once the task has been completed.

There are also a number of methods in the `Core` namespace that operate on Futures:

- Core#wait_for_all takes a list of Future instances and does not return until all of them are set.
- Core#wait_for_any takes a list of Future instances and returns as soon as any one of them is set.

To obtain a `Future`, you can use either the client's `send_async` method or the `Core` namespace's `task` method.

5.3.2 Scheduling an Asynchronous Task Using a Workflow or Activity Client

To schedule an asynchronous task using a workflow or activity client, use the client's GenericClient#send_async method (provided by the parent class, GenericClient) method to schedule the activity. **send_async** returns a Core::Future immediately.

```
def make_booking options
  puts "Workflow has started\n" unless is_replaying?
  futures = []

  if options[:reserve_car]
    puts "Reserving a car for customer\n" unless is_replaying?
    futures << client.send_async(:reserve_car, options[:request_id])
  end

  if options[:reserve_air]
    puts "Reserving air ticket\n" unless is_replaying?
    futures << client.send_async(:reserve_air, options[:customer_id])
  end

  puts "Waiting for reservation to complete\n" unless is_replaying?
  wait_for_all(futures)
  client.send_confirmation(options[:customer_id])
  puts "Workflow has completed\n" unless is_replaying?
end
```

In the preceding code, `send_async` is used to schedule two asynchronous activities, and then `wait_for_all` is used to wait for both activities to complete before scheduling a third activity.

5.3.3 Executing a Block Asynchronously

Using the Core#task method, you can execute any block of code asynchronously. Like `send_async`, the `task` method returns a Core::Future immediately and then begins executing the asynchronous code. When the code has completed, the returned `Future` instance will be set.

5.3.4 Handling Errors in Asynchronous Code

Asynchronous code requires special consideration when handling errors. The AWS Flow Framework for Ruby provides an Core#error_handler method that provides a Ruby-like way, using `begin`/`rescue`/`end`-like semantic for error handling. You pass `error_handler` a block of the following form:

```
error_handler do |t|
  t.begin do
    my_activity_client.send_async :my_activity
  end
  t.rescue ActivityTaskTimedOutException do |e|
    # handle timeout
  end
  t.rescue ActivityTaskFailedException do |e|
    # handle failure
  end
  t.ensure do
    # clean up and stuff
  end
end
```

See *Handling Errors* for more information about handling errors in the AWS Flow Framework for Ruby.

You can also use `error_handler` to provide custom logic for retry attempts on failed tasks. For more information, see *Providing your own Retry Logic*.

5.3.5 Additional Examples

For additional examples of working with asynchronous tasks, see the AWS Flow Framework for Ruby Samples project. Many of the recipes and samples demonstrate the use of asynchronous tasks to create various workflow patterns.

The AWS Flow Framework for Ruby samples are hosted on GitHub at:

- http://github.com/awslabs/aws-flow-ruby-samples/

5.4 Retrying Failed Tasks

There are a number of ways to retry failed activity or workflow tasks, or even arbitrary methods or blocks of code with the AWS Flow Framework for Ruby:

- If you have an activity or workflow task that you want to configure for automatic retries, you can set ExponentialRetryOptions during activity/workflow registration, or when scheduling a task.
- You can retry any method by using one of the generic client methods: GenericClient#retry or GenericClient#exponential_retry.
- You can use the `AWS::Flow` method with_retry to retry any block of code.

Contents

5.4.1 Configuring a Task for Automatic Exponential Retries

To configure an activity or workflow to automatically retry when it fails, pass in a block of ExponentialRetryOptions in the *exponential_retry* section of the options block when you declare the type. For example:

```
activity :unreliable_activity_with_retry_options do
  {
    version: "1.0",
    default_task_list: "activity_tasklist",
    default_task_schedule_to_start_timeout: 30,
    default_task_start_to_close_timeout: 30,
    exponential_retry: { maximum_attempts: 5 }
  }
end
```

In the preceding snippet, the activity will automatically be retried (up to 5 times) using an exponential retry algorithm if any exception occurs.

You can also pass exponential retry options when scheduling the task:

```
client.send(:unreliable_activity_without_retry_options) do
  {
    exponential_retry: {
      maximum_attempts: 5,
      exceptions_to_include: [ArgumentError],
    }
  }
end
```

In this example, the optional parameter *exceptions_to_include* was specified, restricting retry attempts to occur only in the case of an *ArgumentError*. This overrides the default behavior, which attempts a retry after *any* exception.

5.4.2 Exponential Retry Attempts and Jitter Logic

When you specify exponential retry options, the AWS Flow Framework for Ruby applies a *jitter function* by default, to add some randomization to the retry attempts. This helps to reduce the chance that many activities will be retried at exactly the same time.

If you want to use your own jitter logic when using exponential retries, you can use the *jitter_function* option to set your own jitter function:

```
activity_client(:client) do
  {
    from_class: "RetryActivities",
    exponential_retry: {
      should_jitter: true,
      jitter_function: lambda do |seed, max_value|
        Random.new(seed.to_i).rand(max_value)
      end,
      maximum_attempts: 5,
      exceptions_to_include: [StandardError],
    }
  }
end
```

Tip: If you don't want *any* jitter function applied to exponential retry attempts, you can turn it off by specifying `False` for the *should_jitter* option.

Retrying Methods Using an Activity or Workflow Client

You can retry tasks that were not configured *at declaration* by using the client methods:

You can add exponential retry options using `send`, as described in *Configuring a Task for Automatic Exponential Retries*, or by using the `exponential_retry` or `retry` methods of the GenericClient class (inherited by both GenericActivityClient and WorkflowClient). You can retry the method with either the built-in exponential retry algorithm or by supplying your own retry method.

To use the exponential retry algorithm, use `exponential_retry` with a method to retry, arguments for the method, and a block of ExponentialRetryOptions:

```
activity_client.exponential_retry(:my_activity_method, activity_input) {
  exponential_retry: {
    maximum_attempts: 2,
    exceptions_to_include: [ArgumentError],
  }
}
```

If you want to define your own retry algorithm, use the `retry` method by sending it the method to retry, your own retry function, arguments for the method to retry, and a block of RetryOptions:

```
retry_time_secs = lambda do |first_attempt_time, failure_time, num_retries|
  secs_in_day = 3600 * 24
  if ((failure_time - first_attempt_time) > secs_in_day) then
    return -1
  else
    # Constant rate: just divide the total number of seconds by the number of
    # retries.
    return secs_in_day / num_retries
  end
end

activity_client.retry(:my_activity_method, retry_time_secs, activity_input) {
  exponential_retry: {
    maximum_attempts: 2,
    exceptions_to_include: [ArgumentError],
  }
}
```

Retrying an Arbitrary Block of Code

Using the with_retry method, you can execute any block of code with retries in the AWS Flow context, by supplying a set of RetryOptions and the block of code to execute.

In this example, `with_retry` is used to add retry options to an activity that was registered without them:

```
def handle_unreliable_activity
  retry_options = {
    exponential_retry: {
      maximum_attempts: 5,
      exceptions_to_include: [ArgumentError],
    }
  }

  AWS::Flow::with_retry(retry_options) do
    client.unreliable_activity_without_retry_options
  end
end
```

Providing your own Retry Logic

Although you can provide a list of errors to automatically retry in the *exceptions_to_include* RetryOptions, and a list of errors to automatically exclude from retry attempts in the *exceptions_to_exclude* option, there might be times where you want more control over which conditions will initiate a retry attempt.

To provide your own retry logic, use an exception handling strategy and initiate your own retries in a function called by the code that handles the exception.

5.4.3 Synchronous Example

In a *synchronous workflow*, you can use the standard `begin`/`rescue`/`ensure` pattern:

```ruby
def handle_unreliable_activity
  begin
    client.unreliable_activity_without_retry_options
  rescue StandardError => e
    retry_on_failure(e)
  end
end

def retry_on_failure(ex)
  handle_unreliable_activity if should_retry(ex)
end

def should_retry(ex)
  # custom logic to decide to retry the activity or not according to 'ex'
  return true
end
```

5.4.4 Asynchronous Example

For an *asynchronous workflow*, you can use a similar technique, using `error_handler` and `wait_for_all` to handle the details of error handling for, and waiting for the results of, an asynchronous task.

```ruby
def handle_unreliable_activity
  failure = Future.new
  error_handler do |t|
    t.begin do
      client.send_async(:unreliable_activity_without_retry_options)
    end
    t.rescue Exception do |e|
      failure.set(e)
    end
    t.ensure do
      failure.set unless failure.set?
    end
  end
  wait_for_all(failure)
  retry_on_failure(failure)
end

def retry_on_failure(failure)
  ex = failure.get
  handle_unreliable_activity if !ex.nil? && should_retry(ex)
end

def should_retry(ex)
  # insert your custom logic here.
  return true
```

```
end
```

Additional Information and Examples

Refer to the following resources for more information about the subjects in this topic:

- For more information about error handling in the AWS Flow Framework for Ruby, see *Handling Errors*.
- For more information about programming asynchronous tasks, see *Executing Tasks Asynchronously*.
- To view additional examples of retrying tasks, see the retry_activity recipe in the public AWS Flow Framework for Ruby Samples project on GitHub.

5.5 Using S3DataConverter to Manage Large Workflow Data

By default, the AWS Flow Framework for Ruby uses YAMLDataConverter to serialize Ruby objects that you pass to (as input data), and that is returned from, your workflows and activities.

The Amazon SWF service-defined limit for input or output data from activities and workflows is 32,768 (32K) characters. Any data structure that you want to pass as data directly must fit within this limit.

If you want to pass *more* than 32K characters of data to a workflow or activity, use S3DataConverter. Data larger than 32K characters is stored in an Amazon S3 bucket, and `S3DataConverter` passes a hash containing an Amazon S3 path to the data for the workflow instead of passing the data itself.

`S3DataConverter` locally caches data that it serializes or deserializes and uses the cached data if it exists; it only downloads data from S3 when necessary.

To use S3DataConverter

1. *Activate S3Dataconverter* by setting the `AWS_SWF_BUCKET_NAME` environment variable to an Amazon S3 bucket name.
2. (Optional) *Set a bucket lifecycle* on the Amazon S3 bucket used to store your SWF data.

5.5.1 Activate S3DataConverter

S3DataConverter will be used automatically instead of YAMLDataConverter if you set the `AWS_SWF_BUCKET_NAME` environment variable. For example, on Linux, OS X or unix, use:

```
export AWS_SWF_BUCKET_NAME="bucketname"
```

On Windows, use **`set`** instead of **`export`**.

Note: If you deploy your Ruby applications using AWS Elastic Beanstalk, see Customizing and Configuring a Ruby Environment in the Elastic Beanstalk Developer Guide for information about how to set environment variables on your instances.

5.5.2 Set a bucket lifecycle

The AWS Flow Framework for Ruby doesn't delete files from S3 in order to prevent loss of data. It is recommended that you use Object Lifecycle Management in Amazon S3 to automatically delete objects after a certain period of time.

For example, here is an Amazon S3 bucket lifecycle policy that deletes objects automatically after three days:

```
{
    "Rules": [
        {
            "Status": "Enabled",
            "Prefix": "",
            "Expiration": {
                "Days": 3
            },
            "ID": "swf-bucket-rule"
        }
    ]
}
```

You should set your bucket lifecycle so that it respects the run-time of your workflows. For more information about setting bucket lifecycle configurations, see Specifying a Lifecycle Configuration in the Amazon S3 Developer Guide.

CHAPTER 6

Working with Other AWS Products

This section contains guidance about how to use the AWS Flow Framework for Ruby in conjunction with other Amazon Web Services products, such as CloudWatch, AWS OpsWorks, and Elastic Beanstalk.

6.1 Deploying Workflows with Elastic Beanstalk

You can use Elastic Beanstalk to deploy and run your AWS Flow Framework for Ruby workflows, activities and workers. This topic will lead you through the procedure to do so, using the example provided in *Hello World*.

Contents

6.1.1 Prerequisites

This example assumes that you meet the following prerequisites:

- Ruby and the AWS Flow Framework for Ruby (at least version *2.4.0*) are installed as described in *Setting Up*.
- Your AWS credentials are configured as described in *Providing AWS Credentials*.

6.1.2 Create the Workflow Application

Creating a workflow application that can be deployed and run on Elastic Beanstalk is similar to creating a local application with **`aws-flow-utils`**.

To create the application:

1. Open a command-line window and use **aws-flow-utils** to create an application skeleton, specifying `-c eb` to create an Elastic Beanstalk-compatible application. You also need to set the region to run the EC2 instances in, using the `-r` argument:

```
aws-flow-utils -n HelloBeanstalk -c eb -r us-west-2
```

2. This will create a project in the `HelloBeanstalk` directory in the path where you ran **aws-flow-utils**, and will output a 1-Click URL that you can use to create your Elastic Beanstalk application. It will look something like this:

```
---
Your AWS Flow Framework for Ruby application will be located at:
  /path/to/HelloBeanstalk/
AWS Elastic Beanstalk 1-Click URL:
  http://console.aws.amazon.com/elasticbeanstalk/?region=...
---
```

   Save the generated URL—you will use it to configure your application.

   The project directory will contain the following directories and files:

```
HelloBeanstalk
 |-- Gemfile
 |-- config.ru
 |-- flow
 |    |-- activities.rb
 |    `-- workflows.rb
 `-- worker.json
```

3. In the `flow/activities.rb` file, add the following code:

```ruby
class HelloWorld
  def hello(input)
    "Hello #{input[:name]}!"
  end
end
```

4. Create a .zip archive of the `HelloBeanstalk` directory in a way that is supported by your operating system. On Linux, Unix or OS X systems, you can type:

```
zip -r HelloBeanstalk.zip HelloBeanstalk
```

That's all you need in order to create your workflow application. Now you can deploy it with Elastic Beanstalk.

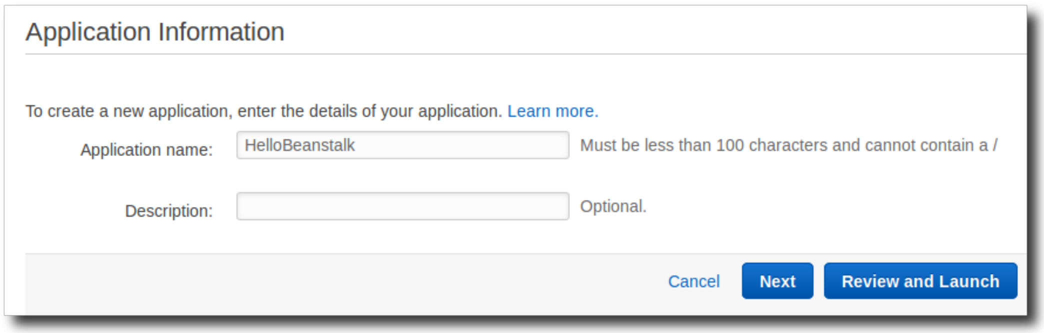
6.1.3 Configure your Application using the Elastic Beanstalk Console

The 1-Click URL that was printed to the screen as output when you ran **aws-flow-utils** will be used now to configure your application on the Elastic Beanstalk console so that you can deploy it.

To configure your application on the AWS Elastic Beanstalk console:

1. Sign in to the AWS Management Console, and follow the 1-Click URL provided in the output of the **aws-flow-utils** command. The URL will bring you to Elastic Beanstalk's **Create New**

Application page, pre-configured for your application.

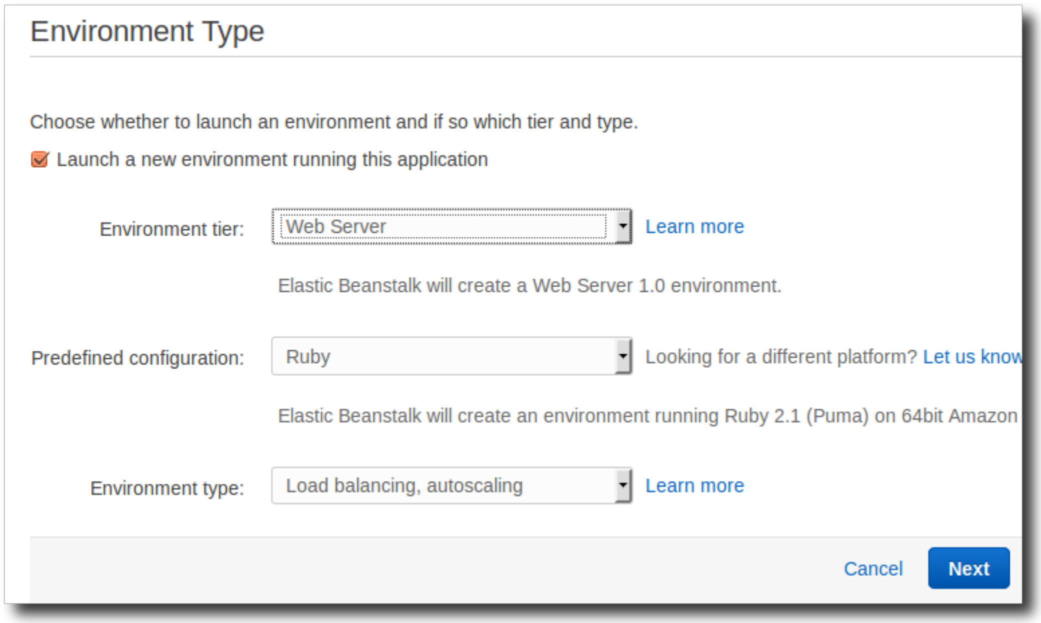

Application Information

To create a new application, enter the details of your application. Learn more.

Application name: HelloBeanstalk — Must be less than 100 characters and cannot contain a /

Description: — Optional.

2. Enter an optional **Description** and click **Next**.
3. On the **Environment Type** page, choose *Web Server* for the **Environment tier** option.


Environment Type

Choose whether to launch an environment and if so which tier and type.

✓ Launch a new environment running this application

Environment tier: Web Server — Learn more

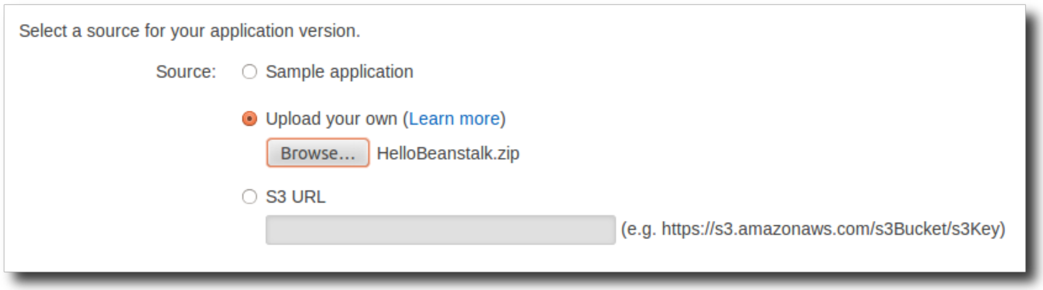
Elastic Beanstalk will create a Web Server 1.0 environment.

Predefined configuration: Ruby — Looking for a different platform? Let us know

Elastic Beanstalk will create an environment running Ruby 2.1 (Puma) on 64bit Amazon

Environment type: Load balancing, autoscaling — Learn more

4. Ensure that the **Predefined Configuration** is *Ruby*, and click **Next**.
5. On the **Application Version** page, choose *Upload your own*, and click **Browse...**, choosing the `HelloBeanstalk.zip` file that you created earlier.


Select a source for your application version.

Source: ○ Sample application

● Upload your own (Learn more)

Browse... HelloBeanstalk.zip

○ S3 URL

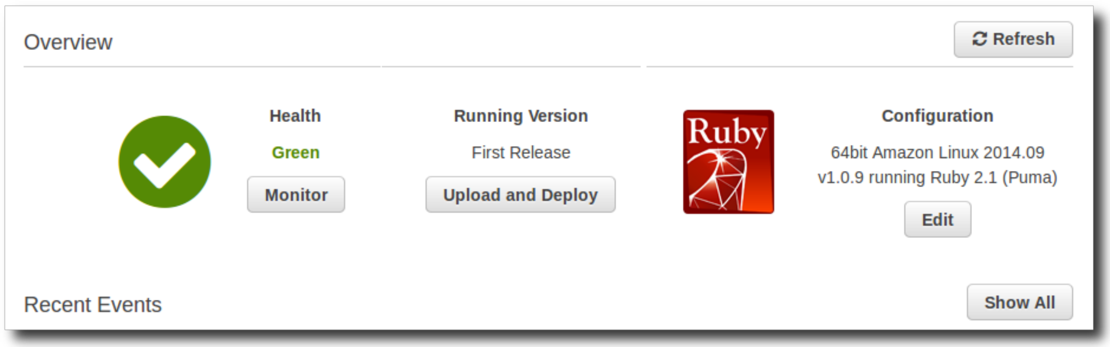
(e.g. https://s3.amazonaws.com/s3Bucket/s3Key)

6. Click **Next** to proceed to the **Environment Information** page. There are no options that need to be

set on this page.

7. Click **Next** to proceed to the **Additional Resources** page. Again, there are no options that need to be set on this page.
8. Click **Next** to proceed to the **Configuration Details** page.
9. On the **Configuration Details** page, choose an EC2 key pair, or open a new browser window and configure one now, using the **IAM Console**.
10. Choose an **Instance Profile** that has access to Amazon EC2, Elastic Beanstalk, and Amazon SWF.
11. Click **Next** to proceed to the **Environment Tags** page. You can leave the tags empty.
12. Click **Next** to proceed to the **Review** page.
13. Review your application's settings, and click **Launch** to begin creating your deployment.

Elastic Beanstalk will take some time to fully launch your application. When it is ready, you'll see **Health** of your Elastic Beanstalk deployment turn to a green circle:



Your workflow is now deployed!

6.1.4 Start a Workflow Execution

Lastly, you need to start a workflow execution so that your workers receive tasks to process.

To start a workflow execution:

1. Open a command-line window and create a local script (call it `starter.rb`). Add the following code:

```
require 'aws/decider'
AWS::Flow::start("HelloWorld.hello", { name: "AWS Flow Framework!" })
```

2. Run the script using Ruby:

```
ruby starter.rb
```

This will begin executing the `hello` activity on your Elastic Beanstalk-deployed application.

6.2 Tutorial: Hello AWS OpsWorks!

This tutorial will show you how to use the AWS Flow Framework for Ruby layer for AWS OpsWorks to deploy and run the Hello World sample application that is described in detail in *Hello World*.

Contents

6.2.1 Amazon SWF support for AWS OpsWorks

Amazon SWF now provides a dedicated layer in AWS OpsWorks that simplifies deployment of workflows and activities written using AWS Flow Framework for Ruby. Using AWS OpsWorks with Amazon SWF, you can easily set up a worker fleet that is cloud-deployable and can use advanced Amazon EC2 features such as load-based auto scaling.

Amazon SWF support for AWS OpsWorks includes updates to the Opsworks console, allowing you to deploy workflow and activity workers from AWS OpsWorks. It also includes updates to the AWS Flow Framework for Ruby to make it easy to specify the details necessary to spawn workers with a simple JSON file, registering any necessary workflow and activity types and starting the activity and workflow workers. This component is called the *runner*, and is provided by a new command-line utility: **`aws-flow-ruby`**.

The typical steps for deploying a new AWS Flow Framework for Ruby application on AWS OpsWorks are the following:

1. Develop your AWS Flow Framework for Ruby workflows and activities normally.
2. Test your application by using *the runner* to check that your workflow runs as expected. AWS Flow Framework for Ruby version 2.0.1 or greater is required to run this step.

   Note: AWS Flow Framework for Ruby version 2.4.0 introduces changes that are incompatible with the current version of the flow layer in AWS OpsWorks. For now, you should use a version of the framework previous to 2.4.0.

3. Set up your application on AWS OpsWorks using the AWS Management Console by creating a stack, layer, and application to deploy.
4. Deploy your application using AWS OpsWorks and monitor your workflow's progress.

The following sections walk through the full set of steps to learn how to configure and use the runner to test how your application will run with AWS OpsWorks. However, if you're interested only in learning how to set up and deploy a working AWS Flow Framework for Ruby application with AWS OpsWorks, you can skip ahead to *Deploying and Running Hello World on AWS OpsWorks*.

6.2.2 Developing and Testing an AWS Flow Framework for Ruby Application using AWS OpsWorks

In this section, we'll use AWS OpsWorks and a local utility (**`aws-flow-ruby`**) to deploy and test an AWS Flow Framework for Ruby application based on the *Hello World* sample used in *Hello World*.

Prerequisites

To deploy the sample on AWS OpsWorks, you must have an AWS account with access to both Amazon SWF and AWS OpsWorks. If you haven't yet signed up for AWS, go to http://aws.amazon.com and click the *Sign Up* link to get started.

To run the sample locally using the **`aws-flow-ruby`** command-line utility (frequently referred to as *the runner*):

- You will need to have at least version 2.0.1 of the AWS Flow Framework for Ruby gem installed. For more information about setting up the framework, see *Getting Started*.
- Make sure that you have provided your AWS credentials using the AWS CLI or by setting the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables. For more information, see *Providing AWS Credentials*.

Get the Tutorial Source Code

The source code used in this tutorial is provided on GitHub at:

- http://github.com/awslabs/aws-flow-ruby-opsworks-helloworld

You can view the code there, or download it using the following link:

- http://github.com/awslabs/aws-flow-ruby-opsworks-helloworld/archive/master.zip

This code is a slightly modified version of the *Hello World* sample from the awslabs/aws-flow-ruby-samples project on GitHub.

Once you have the sample installed in a local directory on your system, open a terminal (command-line) window and change to the directory where you unzipped the sample code.

The sample's top-level directory contains the following files and directories:

Name	Description
`Gemfile`	The gem specification. Sets the AWS Flow Framework gem version to use (must be at least *2.0.1*)
`helloworld.json`	Describes the runner (**`aws-flow-ruby`**) configuration.
`flow` directory	Contains files that are used by the runner to find the activities and workflows.
`lib` directory	Contains the workflow worker, activity worker, and workflow starter code that is run to start the workflow. These files are the same files that are described in Hello World, but have been modified to run in the context of the runner.

The contents of these files will be described fully in later sections of the tutorial.

Add the Required Gemfile

AWS OpsWorks requires that you add a `Gemfile` to your code in the root directory of your project to identify which version of the AWS Flow Framework for Ruby gem to use when deploying and running your code.

Here is the `Gemfile` for the *Hello World* sample:

```
source "http://www.rubygems.org"

gem 'aws-flow', '~> 2', '>= 2.0.1'
```

Set Up the Runner Configuration

To discover details about how the workflows and activities will be run, **aws-flow-ruby** reads a JSON-formatted configuration file, as described in *aws-flow-ruby*.

Note: While using AWS OpsWorks, you don't need to specify a configuration file—the AWS Flow Framework for Ruby layer creates its own configuration based on values that you specify in the AWS Management Console. However, to test your workflow setup locally before deploying it to AWS OpsWorks, you will need to create a local runner configuration file.

To set up the runner configuration:

1. If you have not already done so, open a terminal window and change to the location where you unzipped the sample code (`aws-flow-ruby-opsworks-helloworld`).
2. In the sample's root directory, create or view the file called `helloworld.json`. It contains the following lines:

```
{
  "domain":
    {
      "name": "HelloWorld"
    },
  "workflow_workers": [
    {
      "task_list": "workflow_tasklist"
    }
  ],
  "activity_workers": [
    {
      "task_list": "activity_tasklist"
    }
  ]
}
```

This file specifies the domains to register (if necessary) and use for the workflow and activity workers. It also specifies the number of workflow and activity workers to spawn and the number of process forks allowed when running the activities.

3. In the flow directory within the `aws-flow-ruby-opsworks-helloworld` directory, create or view *activities.rb* and *workflows.rb*. The runner loads these files to discover where the activity and workflow code is.

   The *flow/activities.rb* file provides the activity code:

```
require 'lib/helloworld_activity'
```

   The *flow/workflows.rb* file provides the workflow code:

```
require 'lib/helloworld_workflow'
```

   In the sample, these files simply require files that exist in the *lib* directory. The runner will look in these files for classes that extend Activities and Workflows, which it will take to be the activities and workflows to run, respectively.

Note: If you want to explicitly specify which activity and workflow classes to run, add them to the runner configuration file instead. For more information, see *Runner specification file*.

The contents of the files are similar to the original files from the *Hello World* sample, but have been simplified. The runner takes care of most of the configuration for you. Here are the contents of each:

lib/helloworld_activity.rb

```
require 'aws/decider'

# The HelloWorldActivity class defines a set of activities for the HelloWorld sample.
class HelloWorldActivity
  extend AWS::Flow::Activities

  # Define which activities to run.
  activity :say_hello do
    {
      version: '1.0',
    }
  end

  # This activity will say hello when invoked by the workflow
  def say_hello(name)
    puts "Hello, #{name}!"
  end
end
```

lib/helloworld_workflow.rb

```
require_relative "helloworld_activity"

# HelloWorldWorkflow class defines the workflows for the HelloWorld sample
class HelloWorldWorkflow
  extend AWS::Flow::Workflows

  # Define which workflows to run.
  workflow :hello do
```

```
    {
      version: '1.0',
      default_execution_start_to_close_timeout: 120
    }
  end

  # Create an activity client using the activity_client method to schedule
  # activities
  activity_client(:client) { { from_class: "HelloWorldActivity" } }

  # This is the entry point for the workflow
  def hello(name)
    # Use the activity client 'client' to invoke the say_hello activity
    client.say_hello(name)
  end
end
```

Verify the Code and Configuration

You can now execute **aws-flow-ruby** locally to make sure the runner can find and launch your workflow and activity. Because AWS OpsWorks also uses the AWS Flow Framework for Ruby runner to run your workflow code, this is a good way to test your code's readiness for AWS OpsWorks.

To verify the setup locally:

1. If you have not already done so, open a terminal window and change to the `aws-flow-ruby-opsworks-helloworld` directory where you cloned or unzipped the sample code.
2. Execute the runner from within the `aws-flow-ruby-opsworks-helloworld` directory, passing it the name of the configuration file you created. For example:

```
aws-flow-ruby -f helloworld.json
```

   This will start the workflow worker and activity worker using the domain that is specified in `helloworld.json`. When it runs, you will see output from the runner, such as:

```
waiting on workers [4573, 4574] to complete
```

   The workflow and activity workers are waiting for Amazon SWF tasks to schedule the workflows and activities for execution. Next, you'll need to start a workflow execution to begin workflow and activity scheduling.

3. Run the `helloworld_workflow_starter.rb` script to start the workflow execution:

```
bundle exec ruby lib/helloworld_workflow_starter.rb helloworld.json
```

   After a short time, you should see the results of the *Hello World* activity running:

```
Hello, AWS Flow Framework!
```

Now you're ready to run the sample on AWS OpsWorks instead of the local machine.

6.2.3 Deploying and Running Hello World on AWS OpsWorks

In this section, we'll set up and deploy the AWS OpsWorks-enabled version of the Hello World application on AWS OpsWorks. If you completed the previous section, *Developing and Testing an AWS Flow Framework for Ruby Application using AWS OpsWorks*, you will soon note that these steps use the same settings that you used when *configuring the runner*. If you didn't download and test the application locally, don't worry—the application we'll be using has been set up for AWS OpsWorks already.

Set up a Stack

First, you will need to set up an AWS OpsWorks stack. A stack may consist of a number of layers, each of which can have apps that are deployed to the layers.

To set up the stack:

1. Sign in to the AWS Management Console and open the AWS OpsWorks console at opsworks.
2. Add a new stack by clicking the *Add Stack* button or by selecting *Add Stack* in the *Select Stack* menu.
3. You can use the default values that are provided to create your AWS OpsWorks stack, or set your own. Here are suggested values for each of the fields on the screen:

Field Name	Choose...
Name	a unique name, or leave blank to allow AWS OpsWorks to choose a name for you.
Region	*US East (N. Virginia)*
VPC	*No VPC*
Default Availability Zone	*us-east-1a*
Default operating system	*Amazon Linux*
Default root device type	*Instance store*
IAM role	an existing role with access to Amazon SWF, such as "aws-opsworks-service-role", or choose a new role and give that role access to Amazon SWF.
Default SSH key	an SSH key that you've created, or create a new one. For information about using key pairs to log in to an AWS OpsWorks instance, see Using SSH to Log In to a Linux Instance in the AWS OpsWorks User Guide.
Default IAM instance profile	an existing instance profile with access to Amazon SWF, such as "aws-opsworks-ec2-role", or choose a new instance profile and give the profile access to Amazon SWF.
Hostname theme	any hostname theme you like.
Stack color	any stack color you like.

Important: Be sure that Chef version 11.10 is selected. You can click *Advanced* on the *Add Stack* screen to see what the Chef version is or to change it. If you've already created a stack, you can change the Chef version by editing the existing stack.

4. When you're finished configuring your layer, click the *Add Stack* button at the bottom of the configuration screen to create your stack and continue.

Add a Layer to the Stack

Now that the stack is set up, add a new layer to the stack.

To add the layer:

1. Select your new AWS OpsWorks stack if it is not already selected, and then click *Add a Layer*.
2. Set the *Layer Type* to *AWS Flow*. You'll find this type in the *Other* category in the drop-down list.
3. Since you selected a default IAM profile for your layer, it will be automatically selected for the layer's EC2 instance profile.
4. When you have finished setting your layer options, click *Add Layer* to complete adding the layer to your stack.

Add an Instance

Next, add an Amazon EC2 instance to the layer using the Amazon EC2 instance profile that you set. The instance provides the computing platform to run your workflow code.

To add an instance to the layer:

1. Choose the *Instances* view on the left sidebar if it is not already selected.
2. Click *Add an instance* to add a new instance to the stack.

   You can accept the default options (copied from your layer), or modify the *Hostname* and *Instance Size*.

   Important: Running an Amazon EC2 instance will incur AWS costs. For information about the costs of running various Amazon EC2 instance types, see the Amazon EC2 Pricing page.

3. Click the *Add Instance* button to finalize your settings and add the instance to your layer. Your instance will initially be in the stopped state.
4. If you'll be proceeding with adding the Hello World application and running it, start your instance now. It may take a few minutes for your instance to start.

Add the Hello World Application

Next, you will add the modified Hello World application to your stack. This will instruct AWS OpsWorks to deploy your application to the Amazon EC2 instances that it starts on your behalf.

To add the Hello World application to your stack:

1. Click the *Navigation* heading and choose *Apps* in the list. If your browser window is large enough, you'll find the *Apps* link positioned on the left sidebar.
2. Click *Add an app* to begin adding a new application.
3. In the *Settings* section, choose a name that is memorable, such as *helloworld*. Verify that the *Type* is set to *AWS Flow (Ruby)*.
4. Under *AWS Flow (Ruby) Settings*, choose an Amazon SWF domain name to run your workflows and activities under. This doesn't need to be the same domain that you used in the local test of the Hello World application. The runner will register the domain if necessary, so feel free to choose a new domain name if you would like to.

   You can accept the default workflow retention period of *10 days*.
5. Under *Run Workflows and Run Activities*, set the name of the workflow and activity task lists to use. If you chose to use the same domain that you used in the local test, you should pick new workflow and activity task names so that your workflows and activities don't receive tasks from the local test.

   Note: The workflow starter that you initiate later in *Start your workflow execution* will need to use the same task list name as the one you set for your workflow worker in order to start the workflow.
6. Under *Data Sources*, choose *None* for the *Data Source Type*.
7. Under *Application Source*, choose the *Git* repository type and then set the *Repository URL* to the clone URL of the Hello World AWS OpsWorks sample: http://github.com/awslabs/aws-flow-ruby-opsworks-helloworld.git.

   You can leave the *Repository SSH Key* and *Branch/Revision* settings blank.

   Note: Alternatively, you can choose the *Http Archive* repository type, download or create a `.zip` or `.tar.gz` archive of the sample, and publish it in a accessible location (such as Amazon S3). You can then provide the public URL of the sample archive for this setting.

   For more information, see Adding Apps in the AWS OpsWorks User Guide.
8. Under *Environment Variables*, add a key named *AWS_REGION* with its value set to the AWS region that you plan to deploy the worker to, such as *us-west-2*.
9. Click *Add App* to complete adding the Hello World application to your layer.

Note: The `helloworld.json` file that exists within the repository is ignored. When AWS OpsWorks deploys the application to your Amazon EC2 instances, it will create its own JSON configuration file that contains the application settings that you have just set.

Deploy Hello World

To see if your instance is running, click the *Instances* item on the left sidebar or in the *Navigation* menu. Once the instance is running, you can deploy the Hello World application to it.

To deploy the Hello World application to your EC2 instance:

1. Once your instance is running, click the *Apps* item and click *Deploy* on the Hello World application that you've just finished setting up.
2. Verify that the *Command* setting is *Deploy*, and then click the *Deploy* button at the bottom of the screen. The view will automatically proceed to *Deployments*, and you can monitor the progress of your deployment. It may take a few minutes before it is ready.

Verify Your Deployment

Now that you have deployed Hello World to the Amazon EC2 instance, you can log in and verify your deployment. In order to do this, you will need the SSH key-pair that you associated with your Amazon EC2 instance.

To verify your deployment on the EC2 instance:

1. If you are not already on the *Instances* view in your AWS OpsWorks console, select *Instances* in the menu or in the left sidebar.
2. Click the SSH link. If you have not yet associated the SSH key with your account, the next screen will provide instructions and a link to do so. Otherwise, you will be presented with a list of ways to connect to your instance, through your browser or by using the command-line to SSH to the instance.

   For example, from the command-line, you can specify your private key file and the instance address, as shown in the *Connect Directly* section of the page you receive when clicking the SSH link:

```
ssh -i ~/.ssh/YOUR-KEYFILE my-account@INSTANCE-DNS
```

   Replace *YOUR-KEYFILE* and *INSTANCE-DNS* with your key file name and the IP address of your instance, respectively.

3. Once you are logged in to your instance, view the runner configuration file by typing the following commands (assuming that the application name you chose was *helloworld*):

```
cd srv/www/helloworld
cat runner_config.json
```

Important: The contents of `runner_config.json` must match the values that you entered when setting up your application.

Start your workflow execution

When Hello World is deployed, AWS OpsWorks starts your workflow and activity workers, which start polling on the task lists that you specified when setting up your application.

However, since a workflow execution has not started, the workers won't receive any tasks and no activities will be run. You can start a workflow execution on the command line as was performed in *Verify the Code and Configuration*, or you can start the workflow execution using the AWS Management Console. Both methods will be shown.

Starting a workflow execution on the command line

To start a workflow execution using the command line:

1. Open a terminal (command-line) window and locate the directory where you unpacked or cloned the sample code, as per *Get the Tutorial Source Code*. You could also clone or unpack a new instance of the code if you don't want to modify the existing code.
2. Whichever method you chose (whether it was to use the existing code or download a new copy), change to the tutorial code directory. For example:

```
cd aws-flow-ruby-opsworks-helloworld
```

3. Edit `helloworld.json` and make sure that it contains the same values that you chose when setting up your application in *Add the Hello World Application*. If you chose different values when setting up the application, change the values in `helloworld.json` to match.

   Note: For convenience, the AWS OpsWorks version of `helloworld_workflow_starter.rb` uses the same *helloworld.json* file that the runner uses to find the domain and task lists to use. If you're curious to see how this is done, you can open the file to examine it.

4. If you made any modifications to *helloworld.json*, save the file and then run hello_workflow_starter.rb just as you did when testing the code in *Verify the Code and Configuration*:

```
bundle exec ruby lib/helloworld_workflow_starter.rb helloworld.json
```

5. Open the AWS Management Console, navigate to the *SWF* section and click *Workflow Executions* to monitor the progress of your workflow execution. You should be able to see your workflow events in the *Events* tab, and your running activities in the *Activities* tab of the displayed workflow execution.

Starting a workflow execution using the AWS Management Console

Alternatively to executing the workflow on the command line, you can start it using the AWS Management Console.

To start a workflow execution using the console:

1. Using the AWS Management Console, navigate to the Amazon SWF section by selecting *SWF* in the *Services* menu.
2. Click *Dashboard* and then select the same domain that you used when setting up your application in *Add the Hello World Application*.

3. Click *Start a new Workflow Execution* in the *Quick Links* section of the page.
4. Set the *Workflow Type Name* and *Workflow Type Version* that was specified in the Hello World application. You can verify these settings by viewing the `helloworld_utils.rb` file in the sample. If you did not modify it, these values will be *HelloWorldWorkflow* and *1.0*. Enter any value you like for the *Workflow Execution ID*.
5. Click *Advanced Options* and set *Task List* to be the same task list that you used when adding the application on AWS OpsWorks. Choose reasonable values for the *Execution Start to Close Timeout* and the *Task Start to Close Timeout*.

   For reference, the values used in the sample code are 3600 and 30 seconds, respectively.
6. Click *Continue* to proceed to the next screen.
7. Enter any values you like for the *Execution Input*, or leave it blank. Click the *Review* button to review the values that you've entered.
8. When you're satisfied with the values, click *Start Execution* to start your workflow execution and begin executing your workflow and activity tasks.
9. Click *Workflow Executions* and then click the name of the workflow execution you just initiated to monitor its progress as with the command-line initiated execution.

6.2.4 Experiment with your own application

You've successfully created an AWS OpsWorks-ready stack, layer, instance, and application and have completed a deployment to the Amazon EC2 instance you created.

Now that you've proceeded through an entire AWS Flow Framework for Ruby deployment on AWS OpsWorks, you can try deploying your own workflow code. For the best results, follow the steps as they were presented in this guide, substituting your own application in place of the Hello World sample.

With AWS OpsWorks and the AWS Flow Framework for Ruby, you can deploy as many workflow and activity workers as you like on the AWS cloud with minimal setup!

6.2.5 For More Information

For more information about working with AWS Flow Framework for Ruby and AWS OpsWorks, refer to these topics:

- *aws-flow-ruby*
- Adding Apps: AWS Flow (Ruby) in the AWS OpsWorks User Guide

6.3 Amazon SWF Metrics for CloudWatch

Amazon SWF now provides metrics for Amazon CloudWatch that you can use to track your workflows and activities and set alarms on threshold values that you choose.

You can view metrics using the AWS Management Console or using the AWS CLI. For more information, see Viewing, Graphing, and Publishing Metrics in the CloudWatch Developer Guide.

Contents

6.3.1 Working with Metrics

Metrics that Report a Time Interval

Amazon SWF metrics for CloudWatch that report *time intervals* are always measured in milliseconds. These metrics generally correspond to stages of your workflow execution for which you can set workflow and activity timeouts, and have similar names.

For example, the *DecisionTaskStartToCloseTime* metric measures the time it took for the decision task to complete after it began executing, which is the same time period for which you can set a *DecisionTaskStartToCloseTimeout* value.

For a diagram of each of these workflow stages and to learn when they occur over the workflow and activity lifecycles, see *Amazon SWF Timeout Types*.

Metrics that Report a Count

Some of the Amazon SWF metrics for CloudWatch report results as a *count*. For example, *WorkflowsCanceled*, records a result as either *one* or *zero*, indicating whether or not the workflow was canceled, respectively.

Note: A value of *zero* indicates only that the condition described by the metric did not occur, not that the metric isn't reported.

For count metrics, *minimum* and *maximum* will always be either *zero* or *one*, but *average* will be a value *ranging from zero to one*.

6.3.2 Amazon SWF Workflow Metrics

The following metrics are available for Amazon SWF workflows:

Metric	Description
DecisionTaskScheduleToStartTime	the time interval in milliseconds between the time that the decision task was scheduled and the time it was picked up by a worker and started
DecisionTaskStartToCloseTime	the time interval in milliseconds between the time that the decision task was started and the time it was closed
DecisionTasksCompleted	the count of decision tasks that have been completed
StartedDecisionTasksTimedOutOnClose	the count of decision tasks that started but timed out on closing
WorkflowStartToCloseTime	the time in milliseconds between the time the workflow started and the time it closed
WorkflowsCanceled	the count of workflows that were canceled
WorkflowsCompleted	the count of workflows that completed
WorkflowsContinuedAsNew	the count of workflows that continued as new
WorkflowsFailed	the count of workflows that failed
WorkflowsTerminated	the count of workflows that were terminated
WorkflowsTimedOut	The count of workflows that timed out for any reason

Dimensions for Amazon SWF Workflow Metrics

Dimension	Description
Domain	The Amazon SWF domain that the workflow is running in
WorkflowTypeName	The name of the workflow type for this workflow execution
WorkflowTypeVersion	The version of the workflow type for this workflow execution

6.3.3 Amazon SWF Activity Metrics

The following metrics are available for Amazon SWF activities:

Metric	Description
ActivityTaskScheduleToCloseTime	The time interval in milliseconds between the time when the activity was scheduled to when it closed
ActivityTaskScheduleToStartTime	The time interval in milliseconds between the time when the activity task was scheduled and when it started
ActivityTaskStartToCloseTime	The time interval in milliseconds between the time when the activity task started and when it was closed
ActivityTasksCanceled	The count of activity tasks that were canceled
ActivityTasksCompleted	The count of activity tasks that completed
ActivityTasksFailed	The count of activity tasks that failed
ScheduledActivityTasksTimedOutOnClose	The count of activity tasks that were scheduled but timed out on close
ScheduledActivityTasksTimedOutOnStart	The count of activity tasks that were scheduled but timed out on start
StartedActivityTasksTimedOutOnClose	The count of activity tasks that were started but timed out on close
StartedActivityTasksTimedOutOnHeartbeat	The count of activity tasks that were started but timed out due to a heartbeat timeout

Dimensions for Amazon SWF Activity Metrics

Dimension	Description
Domain	The Amazon SWF domain that the activity is running in
ActivityTypeName	The name of the activity type
ActivityTypeVersion	The version of the activity type

CHAPTER 7

Utilities

The AWS Flow Framework for Ruby is packaged with two utilities, `aws-flow-ruby`, which can be used to spawn and manage *workers*, and `aws-flow-utils`, which can generate AWS Flow Framework for Ruby application skeletons for you.

7.1 `aws-flow-ruby`

`aws-flow-ruby` (also referred to as *the runner*) is a command-line utility that you can use to spawn workflow and activity workers according to a specification that you provide in a JSON configuration file. It is provided with the AWS Flow Framework for Ruby beginning with version *1.3.0*.

Note: While **`aws-flow-ruby`** will start activity and workflow workers, it is not designed to start the workflow execution itself. See *Starting a Workflow Execution* for more information.

Contents

7.1.1 Starting workers with `aws-flow-ruby`

To use **`aws-flow-ruby`** to launch your activity and workflow workers, provide it with the JSON configuration file as its sole argument:

```
aws-flow-ruby -f runnerspec.json
```

The JSON file that you provide must adhere to the format specified in *Runner specification file*.

Note: The runner will start the workflow and activity workers that are defined in the file, and they'll start polling for tasks. It does *not start a workflow execution*. You must perform that step separately. For more information, see *Tutorial: Hello AWS OpsWorks!*.

The runner is configured by passing it a JSON-formatted configuration file. Here is a minimal example, providing only the required fields:

```
{
  "domain": { "name": "ExampleDomain" },
  "activity_workers": [
    { "task_list": "example_activity_tasklist" }
  ],
  "workflow_workers": [
    { "task_list": "example_workflow_tasklist" }
  ]
}
```

You do not need to specify either the workflow or activity classes using this minimal setup. The runner will automatically look for the presence of the `activities.rb` and `workflows.rb` files in the `flow` subdirectory in the location that you start **`aws-flow-ruby`**, and will use those as the activity and workflow classes, respectively.

If the activities within the `activities.rb` file are not based on the Activities class, then an Activities class will be generated for you. However, in this case you must explicitly list the class names in the `activity_classes` option.

Tip: You don't need to implement your activities and workflows in the activities.rb and workflows.rb files. You can use these files to simply `require` activity and workflow code that is located elsewhere.

If you want to override the use of these files, specify the `activity_paths`, `workflow_paths`, and related `activity_classes` and `workflow_classes` fields in the runner configuration file.

7.1.2 Runner specification file

Here is a complete list of the sections and fields that can be set in the runner configuration file.

domain Provides the domain name that will be used (or registered, if necessary) by **`aws-flow-ruby`**, and optionally, the domain retention period. If *domain* is not provided, then the domain `FlowDefault` will be used by default.

Parameter	Description
name	*Required*. The domain name to register. This domain name must be unique to your account and region (Two domains in different regions that share the same name are still considered to be wholly different domains).
retention_in_days	*Optional*. The number of days for which the workflow history will be preserved. If this is not specified, a default retention period of `7` days is used.

Example

```
    "domain": {
      "name": "MyExampleDomain",
      "retention_in_days": 10
    }
```

activity_paths *Optional*. Specifies a list of paths to Ruby source files containing activity classes based on the Activities class.

If not specified, **`aws-flow-ruby`** will attempt to load the file `flow/activities.rb`, which will typically contain `require` lines that load the activity source files. For example:

```
    require 'lib/helloworld_activity.rb'
```

The paths that are specified should be relative to the location of the configuration file (where the runner is executed).

Example

```
    "activity_paths": [
      "aws-flow-ruby-samples/Samples/hello_world/lib/helloworld_activity.rb"
    ]
```

activity_workers Specifies a list of activity worker groups to spawn. Each worker takes the following options:

Parameter	Description
activity_classes	*Optional*. A list of activity class names that the activity worker will run. If not provided, then the activity classes to be run will be auto-discovered by looking in the files specified in the activity_paths member (or, alternatively, `flow/activities.rb`) for classes that are based on Activities. **Note:** Any activities that are *not* based on AWS::Flow::Activities must be listed here, or they will not be used.
number_of_forks_per_worker	*Required* on Windows; *Optional* on other platforms. The number of forked processes that are spawned per activity worker. This sets the number of activity tasks that an activity worker can work on in parallel. If not specified, a default value of `20` will be used. You can set this to zero (`0`) to turn forking off, which is *required* on Windows. See *Using the Framework on Microsoft Windows* for more information.
number_of_workers	*Optional*. The number of activity workers (AWS::Flow::ActivityWorker) to spawn. For each activity worker spawned, a default workflow implementation (decider) will be generated, as well. If not specified, a default value of `1` will be used. you can override this value in the `default_deciders` section.
task_list	*Optional*. The task list to use for the activity execution. If this is not specified, then a task list name will be generated for you, based on the name of the first activity class found.

Example

```
"activity_workers": {
  "number_of_workers": 1,
  "number_of_forks_per_worker": 10,
  "activity_classes": ["HelloWorldActivity"],
  "task_list": "activity_tasklist"
}
```

default_deciders *Optional*. Specifies behavior when the AWS Flow Framework for Ruby automatically generates a workflow implementation for your activities.

A single option can be set, *number_of_workers*, which sets how many workers are launched. This can be used to override the value set in the `activity_workers` section.

Example

```
"default_deciders": {
  "number_of_workers" : 3,
}
```

Note: Generated workflows and workers use the task list "flow_default_ruby".

workflow_paths *Optional*. Specifies a list of paths to Ruby source files containing workflow classes based on the Workflows class.

If not specified, **`aws-flow-ruby`** will attempt to load the file flow/workflows.rb, which will typically contain require lines that load the workflow source files. For example:

```
require 'lib/helloworld_workflow.rb'
```

The paths that are specified should be relative to the location of the configuration file (where **`aws-flow-ruby`** is executed).

Example

```
"workflow_paths": [
  "aws-flow-ruby-samples/Samples/hello_world/lib/helloworld_workflow.rb"
]
```

workflow_workers *Required*. Specifies a list of workflow worker groups to spawn. These take the following options:

Parameter	Description
number_of_workers	*Optional*. The number of workflow workers (AWS::Flow::WorkflowWorker) to spawn. If not specified, a default value of `1` will be used.
task_list	*Required*. The task list to use for the workflow execution.
workflow_classes	*Optional*. The list of workflow class names that the workflow worker will run. If not provided, then the workflow classes to be run will be auto-discovered by looking in the files specified in the workflow_paths member (or, alternatively, `flow/workflows.rb`) for classes that are based on Workflows. **Note:** Any workflows that are *not* based on AWS::Flow::Workflows must be listed here, or they will not be used.

Example

```
    "workflow_workers": {
      "number_of_workers": 1,
      "workflow_classes": ["HelloWorldWorkflow"],
      "task_list": "workflow_tasklist"
    }
```

7.2 `aws-flow-utils`

The **`aws-flow-utils`** utility can be used to generate application skeletons suitable for running locally or hosted on Amazon EC2 using Elastic Beanstalk. It is provided with the AWS Flow Framework for Ruby beginning with version *2.4.0*.

Basic syntax:

```
aws-flow-utils -c TYPE -n APPNAME [OPTIONS]
```

At the minimum, you must specify the *type* of application to create (either `local` or `eb`) and give it a name. If you specify `eb`, then **`aws-flow-utils`** will create an application that you can deploy using Elastic Beanstalk. For more information about deploying a workflow using Elastic Beanstalk, see *Deploying Workflows with Elastic Beanstalk*.

7.2.1 Options Reference

There are a number of arguments that you can specify when running **`aws-flow-utils`**:

Option	Description
`-c`, `--command` *TYPE*	*Required*. Create a project of the specified *TYPE*. You can specify either: • `local` to build a locally-executable Amazon SWF application. • `eb` to build an Amazon SWF application configured for use with AWS Elastic Beanstalk. **Note:** on Windows, `local` will create an activity worker that sets *number_of_forks_per_worker* in the resulting `worker.json` file to zero, turning forking off. For more information, see *Using the Framework on Microsoft Windows*.
`-n`, `--name` *NAME*	*Required*. Set the name of the application.
`-r`, `--region` *REGION*	*Optional*. Set the AWS Region. If this argument is not specified, the default value is taken from the environment variable *AWS_REGION*. If *AWS_REGION* is not set, then this argument is *required*.
`-p`, `--path` *PATH*	*Optional*. Set the location where the application will be created. The default is in the local directory (`.`).
`-a`, `--act_path` *PATH*	*Optional*. Sets the path to an activity class that will be copied into your project. If this argument is not specified, then an empty `activity.rb` file will be generated that you can fill in yourself.
`-w`, `--wf_path` *PATH*	*Optional*. A path to a workflow class that will be copied into your project.
`-A`, `--activities` *x,y,z*	*Optional*. Set the names of activity classes within the file set using the `--act_path` argument. This argument is only necessary if your activity classes are not based on the Activities class.
`-W`, `--workflows` *x,y,z*	*Optional*. Set the names of workflow classes within the file set using the `--wf_path` option. This option is only necessary if your workflow classes are not based on the Workflows class.

Tip: There's no need to memorize this list; you can use `aws-flow-utils --help` to get a list of the command syntax and available options.

CHAPTER 8

Additional Resources

In addition to using the contents of this guide, you can learn more about the AWS Flow Framework for Ruby by using the online resources listed in this topic.

Contents

8.1 AWS Flow Framework for Ruby API Reference

The AWS Flow Framework for Ruby Reference provides details about each of the classes, methods and data structures that make up the framework.

8.2 Amazon Simple Workflow Service Forums

The Amazon SWF forums are a great place to post questions and read answers from the Amazon SWF team and other coders working with the AWS Flow Framework and other Amazon SDKs.

8.3 Videos

The video, Introduction to Programming the AWS Flow Framework for Ruby (video), introduces viewers to the AWS Flow Framework for Ruby and walks through the *Hello World* tutorial, providing details about the code and how to run it.

8.4 Samples and Recipes

A set of AWS Flow Framework for Ruby code samples and recipes is available on GitHub, at awslabs/aws-flow-ruby-samples.

CHAPTER 9

Document History

This topic lists important changes to the documentation over the history of the AWS Flow Framework for Ruby Developer Guide.

- **API version:** 2012-01-25
- **Latest documentation update:** Jul 19, 2016

September 3, 2015 A number of issues in the guide have been fixed, and the topics *Handling Errors* and *Troubleshooting and Debugging Workflows* have been moved to *Basic Workflow Programming* from *Advanced Topics*.

January 22, 2015 The AWS Flow Framework for Ruby provides a new utility, **`aws-flow-utils`**, which can generate application skeletons that you can run locally or on Elastic Beanstalk. Many of the tutorials and related content has been updated. For details, refer to the following walkthroughs, which are new (or, in the case of the Hello World tutorial, has been completely rewritten):

- *Hello World*
- *Basic Workflow Example*
- *Deploying Workflows with Elastic Beanstalk*

Additionally, the rest of the documentation has been re-organized and refreshed. For an overview of the new layout, see *What is the AWS Flow Framework for Ruby?*.

December 17, 2014 Amazon SWF now includes support for setting the priority of tasks on a task list, and will attempt to deliver those with higher priority before tasks with lower priority. Information about this feature is provided in *Setting Task Priority*.

November 10, 2014 A number of topics in the Programming Guide section have been revised for better clarity:

- *Specifying Workflow and Activity Options*
- *Retrying Failed Tasks*
- *Executing Tasks Asynchronously*
- *Handling Errors*
- *Troubleshooting and Debugging Workflows*

Additionally, *Troubleshooting and Debugging Workflows* provides information about using the new `WorkflowReplayer` class.

September 8, 2014 Added content and a new tutorial about *how to use AWS OpsWorks with Amazon SWF*.

August 19, 2014 Added documentation about *the runner*, a new command-line utility that helps to configure and launch activities and workflows.

August 18, 2014 The guide has been restructured to more closely resemble other AWS guides. Some of the changes include the following:

- The *Introduction* topic has been renamed *What is the AWS Flow Framework for Ruby?*, and now includes the sections *Important Notes* and *Where to Find the Source Code and Samples*.
- The *Sample Code* and *Recipes* sections have been moved into the *Programming Guide*.
- The *Additional Resources* section has been promoted to a top-level chapter, instead of being hidden within the introduction.

April 8, 2014 A new chapter has been added to the documentation: *Recipes*. This chapter provides recipes for common use cases.

The official AWS Flow Framework for Ruby samples are now described in the *Sample Code* chapter. These are fully-functional Amazon SWF applications that use the AWS Flow Framework for Ruby.

The *Hello World* tutorial has been updated to match the version that currently exists in the AWS Flow Framework for Ruby samples.

November 13, 2013 The code for the basic code example was missing from its topic—this has been fixed.

August 1, 2013 Initial release of the AWS Flow Framework for Ruby Developer Guide.

CHAPTER 10

About Amazon Web Services

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