Entwicklerhandbuch

AWS Flow Framework für Java

API-Version 2021-04-28

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Flow Framework für Java: Entwicklerhandbuch

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Die Marken und Handelsmarken von Amazon dürfen nicht in einer Weise in Verbindung mit nicht von Amazon stammenden Produkten oder Services verwendet werden, die geeignet ist, Kunden irrezuführen oder Amazon in irgendeiner Weise herabzusetzen oder zu diskreditieren. Alle anderen Marken, die nicht im Besitz von Amazon sind, gehören den jeweiligen Besitzern, die möglicherweise mit Amazon verbunden sind oder von Amazon gesponsert werden.

Table of Contents

Was ist das AWS Flow Framework für Java? 1
Was ist in diesem Handbuch enthalten? 1
Erste Schritte 3
Einrichtung des Frameworks 3
Fügen Sie das Flow-Framework mit Maven hinzu 4
HelloWorld Bewerbung 4
HelloWorld Implementierung der Aktivitäten 5
HelloWorld Workflow-Mitarbeiter 6
HelloWorld Workflow-Starter 7
HelloWorldWorkflow Bewerbung 8
HelloWorldWorkflow Aktivitäten Arbeiter 10
HelloWorldWorkflow Workflow-Worker 12
HelloWorldWorkflow Implementierung von Arbeitsabläufen und Aktivitäten 17
HelloWorldWorkflow Vorspeise 21
HelloWorldWorkflowAsyncBewerbung 26
HelloWorldWorkflowAsync Implementierung der Aktivitäten 28
HelloWorldWorkflowAsync Workflow-Implementierung 28
HelloWorldWorkflowAsyncArbeitsablauf und Aktivitäten: Host und Starter 30
HelloWorldWorkflowDistributed Bewerbung 31
HelloWorldWorkflowParallelBewerbung 34
HelloWorldWorkflowParallelAktivitäten Arbeiter 35
HelloWorldWorkflowParallelWorkflow-Mitarbeiter 36
HelloWorldWorkflowParallel Arbeitsablauf und Aktivitäten: Host und Starter 38
Verstehen AWS Flow Framework 39
Anwendungsstruktur 39
Rolle des Aktivitäts-Workers 41
Rolle des Workflow-Workers 41
Rolle des Workflow-Starters 42
So interagiert Amazon SWF mit Ihrer Anwendung 42
Weitere Informationen 43
Zuverlässige Ausführung 43
Bereitstellen von zuverlässiger Kommunikation 43
Sicherstellen, dass Ergebnisse nicht verloren gegangen sind 44
Verarbeitung fehlgeschlagener verteilter Komponenten 45

Verteilte Ausführung ... 45
Workflow-Replay ... 45
Replay und asynchrone Workflow-Methoden ... 47
Replay und die Workflow-Implementierung ... 47
Aufgabenlisten und Aufgabenausführung ... 48
Skalierbare Webanwendungen ... 50
Datenaustausch zwischen Aktivitäten und Workflows ... 51
Die Promise <T> Type ... 51
Datenkonverter und Marshaling ... 53
Datenaustausch zwischen Anwendungen und Workflow-Ausführungen ... 53
Zeitüberschreitungstypen ... 54
Zeitüberschreitungen in Workflow- und Entscheidungsaufgaben ... 54
Zeitüberschreitungen in Aktivitätsaufgaben ... 56
Aufgaben verstehen ... 58
Aufgabe ... 58
Reihenfolge der Ausführung ... 59
Workflow-Ausführung ... 61
Nichtdeterminismus ... 64
Programming Guide ... 65
Implementieren von Workflow-Anwendungen ... 65
Workflow- und Aktivitäts-Verträge ... 67
Registrierung von Workflow- und Aktivitätstypen ... 70
Workflow-Typname und Version ... 71
Signalname ... 71
Aktivitätstypname und Version ... 72
Standardaufgabenliste ... 72
Weitere Registrierungsoptionen ... 72
Aktivitäts- und Workflow-Clients ... 73
Workflow-Clients ... 73
Aktivitäts-Clients ... 82
Planungsoptionen ... 87
Dynamische Clients ... 87
Workflow-Implementierung ... 89
Entscheidungskontext ... 91
Offenlegen des Ausführungsstatus ... 91
Workflow-Lokale ... 93

Implementierung von Aktivitäten ... 94
Aktivitäten manuell abschließen ... 95
Implementierung von Lambda-Aufgaben ... 97
Über AWS Lambda ... 97
Vorteile und Einschränkungen der Verwendung von Lambda-Aufgaben ... 98
Verwenden von Lambda-Aufgaben in Ihren AWS Flow Framework Workflows für Java ... 99
Sehen Sie sich das Beispiel an HelloLambda ... 104
Ausführen von Programmen, die mit dem AWS Flow Framework für Java geschrieben wurden ... 104
WorkflowWorker ... 106
ActivityWorker ... 106
Worker-Threading-Modell ... 106
Worker-Erweiterbarkeit ... 109
Ausführungskontext ... 110
Entscheidungskontext ... 110
Aktivitätsausführungskontext ... 112
Untergeordnete Workflow-Ausführungen ... 113
Fortlaufende Workflows ... 116
Einstellung der Aufgabenpriorität ... 117
Einrichten der Aufgabenpriorität für Workflows ... 118
Einrichten der Aufgabenpriorität für Aktivitäten ... 119
DataConverters ... 119
Datenübergabe an asynchrone Methoden ... 120
Übergabe von Collections und Maps an asynchrone Methoden ... 121
Einstellbare <T> ... 121
@NoWait ... 123
Promise <Void> ... 123
AndPromise und OrPromise ... 124
Prüfbarkeit und Dependency Injection ... 124
Spring-Integration ... 124
JUnit Integration ... 131
Fehlerbehandlung ... 137
TryCatchFinally Semantik ... 140
Abbruch ... 140
Verschachtelt TryCatchFinally ... 145
Wiederholen fehlgeschlagener Aktivitäten ... 147

Retry-Until-Success Strategie 147
Exponentielle Wiederholungsstrategie 150
Benutzerdefinierte Wiederholungsstrategie 157
Daemon-Aufgaben 160
Replay-Verhalten 162
Beispiel 1: Synchrones Replay 163
Beispiel 2: Asynchrones Replay 165
Weitere Informationen finden Sie unter: 165
Bewährte Methoden 166
Vornehmen von Änderungen am Entscheidercode 166
Wiedergabe und Codeänderungen 166
Beispielszenario 167
Lösungen 174
Fehlerbehebung 180
Fehler beim Kompilieren 180
Unbekannter Ressourcenfehler 181
Ausnahmen beim Aufrufen von get () für ein Promise 181
Nichtdeterministische Workflows 182
Probleme aufgrund der Versionierung 182
Problembehandlung und Debuggen einer Workflow-Ausführung 182
Verlorene Aufgaben 184
Die Überprüfung ist aufgrund von Längenbeschränkungen für API-Parameter fehlgeschlagen . 185
Referenz 186
Anmerkungen 186
@Aktivität 186
@Aktivität 187
@ActivityRegistrationOptions 188
@Asynchron 189
@Execute 189
@ExponentialRetry 190
@GetState 191
@ManualActivityCompletion 191
@Signal 191
@SkipRegistration 192
@Wait und @ NoWait 192
@Workflow 192

@WorkflowRegistrationOptions 193
Ausnahmen 195
ActivityFailureException 195
ActivityTaskException 196
ActivityTaskFailedException 196
ActivityTaskTimedOutException 196
ChildWorkflowException 196
ChildWorkflowFailedException 196
ChildWorkflowTerminatedException 197
ChildWorkflowTimedOutException 197
DataConverterException 197
DecisionException 197
ScheduleActivityTaskFailedException 197
SignalExternalWorkflowException 198
StartChildWorkflowFailedException 198
StartTimerFailedException 198
TimerException 198
WorkflowException 198
Pakete 198
Dokumentverlauf 201
..... cciii

Was ist das AWS Flow Framework für Java?

Mit dem können Sie AWS Flow Framework sich auf die Implementierung Ihrer Workflow-Logik konzentrieren. Hinter den Kulissen verwendet das Framework die Planungs-, Routing- und Statusverwaltungsfunktionen von Amazon SWF, um die Ausführung Ihres Workflows zu verwalten und ihn skalierbar, zuverlässig und überprüfbar zu machen. AWS Flow Framework basierte Workflows laufen in hohem Maße parallel ab. Die Workflows können auf mehrere Komponenten verteilt werden, die als separate Prozesse auf separaten Computern ausgeführt und unabhängig voneinander skaliert werden können. Die Anwendung kann weiter ausgeführt werden, wenn eine ihrer Komponenten ausgeführt wird, wodurch sie äußerst fehlertolerant ist.

Was ist in diesem Handbuch enthalten?

Dieses Handbuch enthält Informationen zur Installation, Einrichtung und Verwendung von Amazon SWF SWF-Anwendungen. AWS Flow Framework

Erste Schritte mit dem AWS Flow Framework für Java

Wenn Sie gerade erst mit dem AWS Flow Framework für Java beginnen, lesen Sie den Erste Schritte mit dem AWS Flow Framework für Java Abschnitt. Er führt Sie durch das Herunterladen und Installieren von AWS Flow Framework für Java, die Einrichtung Ihrer Entwicklungsumgebung und führt Sie durch ein einfaches Beispiel für die Erstellung eines Workflows.

Verständnis AWS Flow Framework für Java

Stellt grundlegende Amazon SWF und AWS Flow Framework Konzepte vor und beschreibt die grundlegende Struktur einer AWS Flow Framework Anwendung und wie Daten zwischen Teilen eines verteilten Workflows ausgetauscht werden.

AWS Flow Framework für Java-Programmierhandbuch

Dieses Kapitel enthält grundlegende Programmieranleitungen für die Entwicklung von Workflow-Anwendungen mit dem AWS Flow Framework für Java, einschließlich der Registrierung von Aktivitäten und Workflow-Typen, der Implementierung von Workflow-Clients, der Erstellung untergeordneter Workflows, der Behandlung von Fehlern und mehr.

Eine Aufgabe in AWS Flow Framework für Java verstehen

Dieses Kapitel bietet einen tieferen Einblick in die Funktionsweise von AWS Flow Framework for Java und bietet Ihnen zusätzliche Informationen über die Reihenfolge der Ausführung

asynchroner Workflows sowie eine logische schrittweise Vorgehensweise bei der Ausführung eines Standard-Workflows.

[Tipps zur Fehlerbehebung und zum Debuggen AWS Flow Framework für Java]

Dieses Kapitel enthält Informationen zu häufigen Fehlern, die Sie bei der Fehlerbehebung für Ihre Workflows unterstützen oder Ihnen zeigen, wie Sie häufige Fehler vermeiden.

[AWS Flow Framework für Java-Referenz]

Dieses Kapitel ist ein Verweis auf die Anmerkungen, Ausnahmen und Pakete, die AWS Flow Framework for Java dem SDK for Java hinzufügt.

Erste Schritte mit dem AWS Flow Framework für Java

In diesem Abschnitt wird das vorgestellt, AWS Flow Framework indem er Sie durch eine Reihe einfacher Beispielanwendungen führt, in denen das grundlegende Programmiermodell und die API vorgestellt werden. Die Beispielanwendungen basieren auf der standardmäßigen Anwendung "Hello World", die häufig für die Einführung von C und verwandter Programmiersprachen verwendet wird. Hier sehen Sie eine typische Java-Implementierung von "Hello World":

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

Im Folgenden finden Sie eine kurze Beschreibung der Beispielanwendungen: Der vollständige Quellcode ist angegeben, sodass Sie die Anwendungen selbst implementieren und ausführen können. Bevor Sie beginnen, sollten Sie zunächst Ihre Entwicklungsumgebung konfigurieren und ein AWS Flow Framework Java-Projekt erstellen, wie inEinrichtung des AWS Flow Framework für Java.

- HelloWorld Bewerbung stellt Workflow-Anwendungen vor, indem "Hello World" als standardmäßige Java-Anwendung implementiert, aber wie eine Workflow-Anwendung strukturiert wird.
- HelloWorldWorkflow Bewerbungverwendet den AWS Flow Framework for Java zur Konvertierung HelloWorld in einen Amazon SWF SWF-Workflow.
- HelloWorldWorkflowAsyncBewerbung ändert `HelloWorldWorkflow`, damit eine asynchrone Workflow-Methode verwendet wird.
- HelloWorldWorkflowDistributed Bewerbung ändert `HelloWorldWorkflowAsync`, sodass Workflow und Aktivitäts-Worker auf unterschiedlichen System ausgeführt werden.
- HelloWorldWorkflowParallelBewerbung ändert `HelloWorldWorkflow`, damit zwei Aktivitäten parallel ausgeführt werden können.

Einrichtung des AWS Flow Framework für Java

Das AWS Flow Framework für Java ist im Lieferumfang von enthalten. AWS SDK für Java Falls Sie das noch nicht eingerichtet haben AWS SDK für Java, finden Sie unter Erste Schritte im AWS SDK für Java Entwicklerhandbuch Informationen zur Installation und Konfiguration des SDK selbst.

Fügen Sie das Flow-Framework mit Maven hinzu

Die Amazon SWF SWF-Build-Tools sind Open Source. Um den Code anzusehen oder herunterzuladen oder die Tools selbst zu erstellen, besuchen Sie das Repository unter. https://github.com/aws/aws-swf-build-tools

Amazon stellt Amazon SWF SWF-Build-Tools im Maven Central Repository bereit.

Um das Flow-Framework für Maven einzurichten, fügen Sie die folgende Abhängigkeit zur `pom.xml`-Datei Ihres Projekts hinzu:

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-swf-build-tools</artifactId>
    <version>2.0.0</version>
</dependency>
```

HelloWorld Bewerbung

Um die Struktur von Amazon SWF SWF-Anwendungen vorzustellen, erstellen wir eine Java-Anwendung, die sich wie ein Workflow verhält, aber lokal in einem einzigen Prozess ausgeführt wird. Es ist keine Verbindung zu Amazon Web Services erforderlich.

> **Note**
>
> Das HelloWorldWorkflowBeispiel baut auf diesem auf und stellt eine Verbindung zu Amazon SWF her, um die Verwaltung des Workflows zu übernehmen.

Eine Workflow-Anwendung besteht aus drei Grundkomponenten:

- Ein Aktivitätsauftragnehmer unterstützt eine Reihe von Aktivitäten, die jeweils eine Methode sind, die unabhängig ausgeführt wird, um eine bestimmte Aufgabe zu erfüllen.
- Ein Workflow-Auftragnehmer orchestriert die Ausführung der Aktivitäten und verwaltet den Datenfluss. Er ist eine programmgesteuerte Umsetzung einer Workflow-Topologie. Dabei handelt es sich im Grunde um ein Flussdiagramm, in dem definiert wird, wann die verschiedenen Aktivitäten ausgeführt werden, ob sie nacheinander oder parallel ausgeführt werden usw.
- Ein Workflow-Starter startet eine Workflow-Instance, eine sogenannte Ausführung, und kann während der Ausführung mit ihr interagieren.

HelloWorld ist in drei Klassen und zwei verwandte Schnittstellen implementiert, die in den folgenden Abschnitten beschrieben werden. Bevor Sie beginnen, sollten Sie Ihre Entwicklungsumgebung einrichten und ein neues AWS Java-Projekt erstellen, wie unter beschriebenEinrichtung des AWS Flow Framework für Java. Die für die folgenden Anleitungen verwendeten Pakete heißen `helloWorld.`*`XYZ`*. Um diese Namen zu verwenden, legen Sie das `within`-Attribut in aop.xml wie folgt fest:

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

Erstellen Sie zur Implementierung HelloWorld ein neues Java-Paket in Ihrem AWS SDK-Projekt mit dem Namen `helloWorld.HelloWorld` und fügen Sie die folgenden Dateien hinzu:

- Eine Schnittstellendatei namens `GreeterActivities.java`
- Eine Klassendatei namens `GreeterActivitiesImpl.java`, die den Aktivitätenauftragnehmer implementiert
- Eine Schnittstellendatei namens `GreeterWorkflow.java`
- Eine Klassendatei namens `GreeterWorkflowImpl.java`, die den Workflow-Auftragnehmer implementiert
- Eine Klassendatei namens `GreeterMain.java`, die den Workflow-Starter implementiert

Die Details werden in den folgenden Abschnitten erläutert und enthalten den vollständigen Code der einzelnen Komponenten, den Sie in die jeweilige Datei einfügen können.

HelloWorld Implementierung der Aktivitäten

HelloWorld unterteilt die allgemeine Aufgabe, eine `"Hello World!"` Begrüßung auf der Konsole zu drucken, in drei Aufgaben, von denen jede mit einer Aktivitätsmethode ausgeführt wird. Die Aktivitätsmethoden sind in der Schnittstelle `GreeterActivities` wie folgt definiert.

```
public interface GreeterActivities {
   public String getName();
   public String getGreeting(String name);
   public void say(String what);
}
```

HelloWorld hat eine AktivitätsimplementierungGreeterActivitiesImpl, die die GreeterActivities folgenden Methoden bereitstellt:

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

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

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

Aktivitäten sind unabhängig voneinander und können häufig in unterschiedlichen Workflows verwendet werden. Beispielsweise kann jeder Workflow die Aktivität say verwenden, um eine Zeichenfolge auf der Konsole auszugeben. Workflows können auch über mehrere Aktivitätsimplementierungen verfügen, die jeweils unterschiedliche Aufgaben ausführen.

HelloWorld Workflow-Mitarbeiter

Um „Hello World!“ zu drucken auf der Konsole müssen die Aktivitätsaufgaben nacheinander in der richtigen Reihenfolge mit den richtigen Daten ausgeführt werden. Der HelloWorld Workflow-Worker orchestriert die Ausführung der Aktivitäten auf der Grundlage einer einfachen linearen Workflow-Topologie, die in der folgenden Abbildung dargestellt ist.

Die drei Aktivitäten werden nacheinander ausgeführt und die Daten werden von einer Aktivität an die nächste übergeben.

Der HelloWorld Workflow-Worker hat eine einzige Methode, den Einstiegspunkt des Workflows, der in der GreeterWorkflow Benutzeroberfläche wie folgt definiert ist:

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

Die `GreeterWorkflowImpl`-Klasse implementiert diese Schnittstelle wie folgt:

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

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

Die `greet` Methode implementiert die HelloWorld Topologie, indem sie eine Instanz von erstelltGreeterActivitiesImpl, jede Aktivitätsmethode in der richtigen Reihenfolge aufruft und die entsprechenden Daten an jede Methode weitergibt.

HelloWorld Workflow-Starter

Ein Workflow-Starter ist eine Anwendung, die eine Workflow-Instance startet und während der Ausführung mit dem Workflow kommunizieren kann. Die `GreeterMain` Klasse implementiert den HelloWorld Workflow-Starter wie folgt:

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

`GreeterMain` erstellt eine Instance von `GreeterWorkflowImpl` und ruft `greet` auf, um den Workflow-Auftragnehmer auszuführen. Führen Sie `GreeterMain` es als Java-Anwendung aus und Sie sollten „Hello World!“ sehen in der Konsolenausgabe.

HelloWorldWorkflow Bewerbung

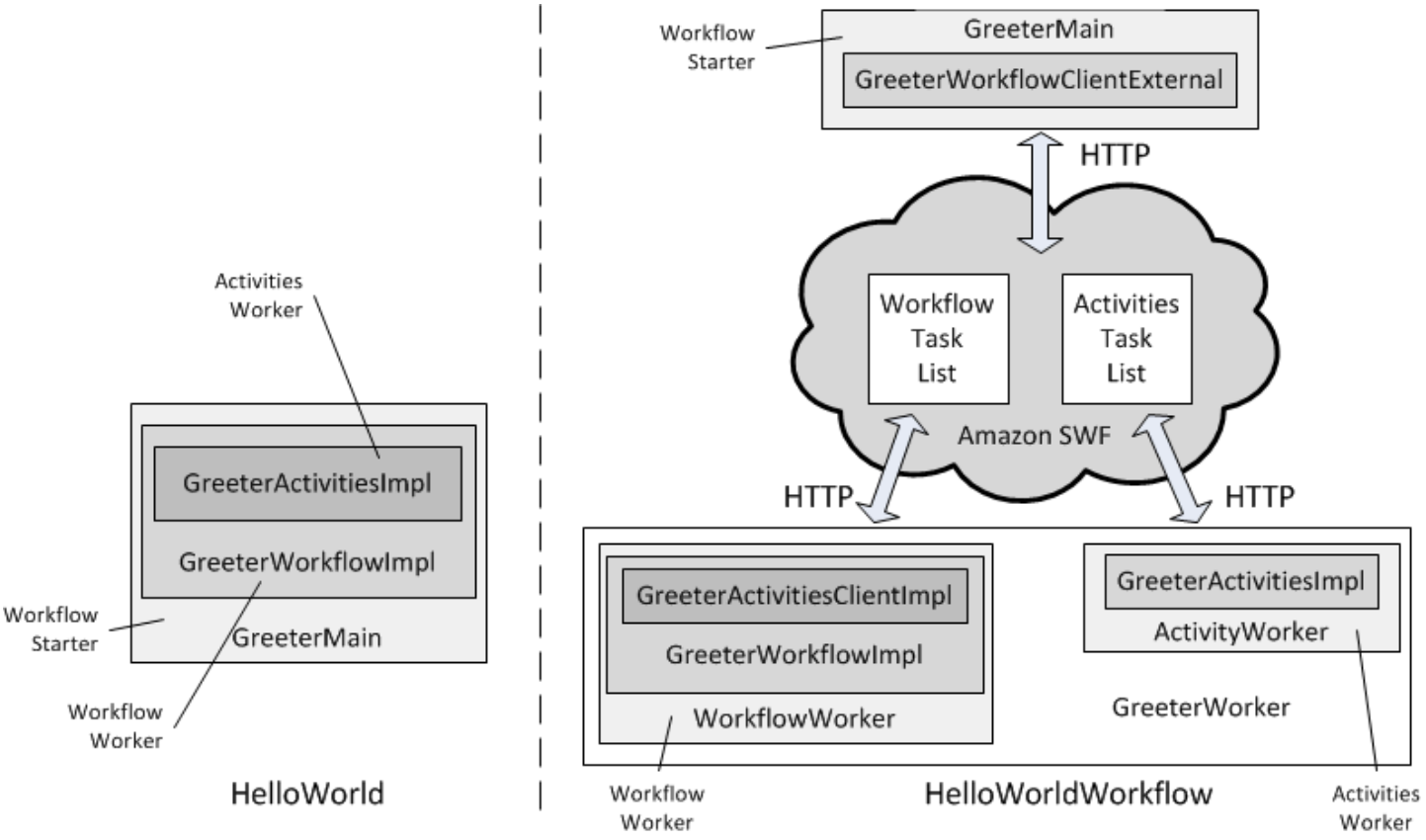
Obwohl das grundlegende HelloWorldBeispiel wie ein Workflow strukturiert ist, unterscheidet es sich in mehreren wichtigen Punkten von einem Amazon SWF SWF-Workflow:

Konventionelle Workflow-Anwendungen und Amazon SWF SWF-Workflow-Anwendungen

HelloWorld	Amazon SWF SWF-Arbeitsablauf
Wird lokal als einzelner Prozess ausgeführt.	Läuft als mehrere Prozesse, die auf mehrere Systeme verteilt werden können, darunter EC2 Amazon-In stances, private Rechenzentren, Client-Computer usw. Es muss nicht einmal das gleiche Betriebssystem verwendet werden.
Aktivitäten sind synchrone Methoden, die bis zu ihrem Abschluss für eine Blockierung sorgen.	Aktivitäten werden durch asynchrone Methoden abgebildet. Diese geben die Kontrolle sofort zurück. Sie ermöglichen es dem Workflow, während der Wartezeit auf den Abschluss der Aktivität andere Aufgaben auszuführen.
Der Workflow-Worker interagiert mit einem Aktivitäts-Worker, indem er die entsprechende Methode aufruft.	Workflow-Worker interagieren mit Activity-Workern mithilfe von HTTP-Anfragen, wobei Amazon SWF als Vermittler fungiert.
Der Workflow-Starter interagiert mit dem Workflow-Worker, indem er die entsprechende Methode aufruft.	Workflow-Starter interagieren mit Workflow-Workern mithilfe von HTTP-Anfragen, wobei Amazon SWF als Vermittler fungiert.

Sie können eine verteilte, asynchrone Workflow-Anwendung von Grund auf neu implementieren, indem Sie beispielsweise Ihren Workflow-Worker direkt über Webservice-Aufrufe mit einem Aktivitäts-Worker interagieren lassen. Allerdings müssen Sie dann den gesamten, komplizierten Code implementieren, der für die asynchrone Ausführung mehrerer Aktivitäten, den Datenfluss usw. erforderlich ist. Die SWF AWS Flow Framework für Java und Amazon kümmern sich um all diese Details, sodass Sie sich auf die Implementierung der Geschäftslogik konzentrieren können.

HelloWorldWorkflow ist eine modifizierte Version davon HelloWorld , die als Amazon SWF SWF-Workflow ausgeführt wird. Die folgende Abbildung fasst die Funktionsweise der beiden Anwendungen zusammen.



HelloWorld wird als ein einziger Prozess ausgeführt, und der Starter, der Workflow-Worker und der Aktivitäten-Worker interagieren mithilfe herkömmlicher Methodenaufrufe. Bei Starter`HelloWorldWorkflow`, Workflow Worker und Activities Worker handelt es sich um verteilte Komponenten, die über Amazon SWF mithilfe von HTTP-Anfragen interagieren. Amazon SWF verwaltet die Interaktion, indem es Listen mit Workflow- und Aktivitätsaufgaben verwaltet und an die jeweiligen Komponenten weiterleitet. In diesem Abschnitt wird beschrieben, wie das Framework für HelloWorldWorkflow funktioniert.

HelloWorldWorkflow wird mithilfe der AWS Flow Framework for Java-API implementiert, die die manchmal komplizierten Details der Interaktion mit Amazon SWF im Hintergrund verarbeitet und den Entwicklungsprozess erheblich vereinfacht. Sie können dasselbe Projekt verwenden HelloWorld, für das Sie bereits AWS Flow Framework für Java-Anwendungen konfiguriert haben. Um die Anwendung auszuführen, müssen Sie jedoch wie folgt ein Amazon SWF SWF-Konto einrichten:

- Eröffnen Sie ein AWS Konto bei Amazon Web Services, falls Sie noch keines haben.

- Weisen Sie den AWS_SECRET_KEY Umgebungsvariablen die Zugriffs-ID AWS_ACCESS_KEY_ID und die geheime ID Ihres Kontos zu. Die Schlüsselwerte selbst sollten nicht in Ihrem Code enthalten sein. Die Speicherung in Umgebungsvariablen ist ein bequemer Weg, um das Problem zu lösen.
- Eröffnen Sie ein Amazon SWF SWF-Konto bei Amazon Simple Workflow Service.
- Melden Sie sich beim Amazon SWF-Service an AWS Management Console und wählen Sie ihn aus.
- Wählen Sie oben rechts Domains verwalten und registrieren Sie eine neue Amazon SWF-Domain. Ein Domäne ist ein logischer Container für Ihre Anwendungsressourcen (z. B. Workflow- und Aktivitätstypen und Workflow-Ausführungen). Sie können jeden beliebigen Domainnamen verwenden, in den exemplarischen Vorgehensweisen wird jedoch "“ helloWorldWalkthrough verwendet.

Um das zu implementieren HelloWorldWorkflow, erstellen Sie eine Kopie von HelloWorld. HelloWorld packe es in dein Projektverzeichnis und nenne es HelloWorld. HelloWorldWorkflow. In den folgenden Abschnitten wird beschrieben, wie Sie den HelloWorld Originalcode ändern, um ihn AWS Flow Framework für Java zu verwenden und als Amazon SWF SWF-Workflow-Anwendung auszuführen.

HelloWorldWorkflow Aktivitäten Arbeiter

HelloWorld hat seine Aktivitäten Worker als eine einzige Klasse implementiert. Ein Worker AWS Flow Framework für Java-Aktivitäten besteht aus drei grundlegenden Komponenten:

- Die Aktivitätsmethoden, die die eigentlichen Aufgaben ausführen, werden in einer Schnittstelle definiert und in einer verwandten Klasse implementiert.
- Eine ActivityWorkerKlasse verwaltet die Interaktion zwischen den Aktivitätsmethoden und Amazon SWF.
- Eine Aktivitäts-Host-Anwendung, die den Aktivitäts-Worker registriert und startet und die Bereinigung übernimmt.

Dieser Abschnitt behandelt die Aktivitätsmethoden. Die beiden anderen Klassen werden später besprochen.

HelloWorldWorkflow definiert die Aktivitätsschnittstelle in `GreeterActivities` wie folgt:

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

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

@ActivityRegistrationOptions(defaultTaskScheduleToStartTimeoutSeconds = 300,
                             defaultTaskStartToCloseTimeoutSeconds = 10)
@Activities(version="1.0")

public interface GreeterActivities {
   public String getName();
   public String getGreeting(String name);
   public void say(String what);
}
```

Diese Schnittstelle war nicht unbedingt notwendig für HelloWorld, aber sie ist AWS Flow Framework für eine Java-Anwendung notwendig. Beachten Sie, dass sich die Schnittstellendefinition selbst nicht geändert hat. Sie müssen jedoch zwei AWS Flow Framework für Java-Anmerkungen @ActivityRegistrationOptions und @Aktivität für die Schnittstellendefinition anwenden. Die Anmerkungen stellen Konfigurationsinformationen bereit und weisen den Annotationsprozessor AWS Flow Framework für Java an, anhand der Schnittstellendefinition eine Clientklasse für Aktivitäten zu generieren, auf die später eingegangen wird.

`@ActivityRegistrationOptions`hat mehrere benannte Werte, die verwendet werden, um das Verhalten der Aktivitäten zu konfigurieren. HelloWorldWorkflow gibt zwei Timeouts an:

- `defaultTaskScheduleToStartTimeoutSeconds` definiert, wie lange sich die Aufgaben in der Aktivitätsaufgabenliste in der Warteschlange befinden können. Der Wert ist auf 300 Sekunden (5 Minuten) festgelegt.
- `defaultTaskStartToCloseTimeoutSeconds` definiert die maximale Zeit, die die Aktivität zur Ausführung der Aufgabe nutzen kann. Der Wert ist auf 10 Sekunden festgelegt.

Diese Timeouts stellen sicher, dass die Aktivität ihre Aufgabe in angemessener Zeit abschließt. Wird ein Timeout überschritten, generiert das Framework einen Fehler und der Workflow-Worker muss entscheiden, wie das Problem behandelt werden soll. Wie man mit solchen Fehlern umgeht, erfahren Sie unter Fehlerbehandlung.

`@Activities` hat mehrere Werte. In der Regel wird jedoch nur die Versionsnummer der Aktivität definiert. So können Sie verschiedene Generationen der Aktivitätsimplementierungen nachverfolgen. Wenn Sie eine Aktivitätsschnittstelle ändern, nachdem Sie sie bei Amazon SWF registriert haben,

einschließlich der Änderung der `@ActivityRegistrationOptions` Werte, müssen Sie eine neue Versionsnummer verwenden.

HelloWorldWorkflow implementiert die Aktivitätsmethoden wie folgt: `GreeterActivitiesImpl`

```
public class GreeterActivitiesImpl implements GreeterActivities {
   @Override
   public String getName() {
      return "World";
   }
   @Override
   public String getGreeting(String name) {
      return "Hello " + name;
   }
   @Override
   public void say(String what) {
      System.out.println(what);
   }
}
```

Beachten Sie, dass der Code mit der HelloWorld Implementierung identisch ist. Im Kern ist eine AWS Flow Framework Aktivität nur eine Methode, die Code ausführt und möglicherweise ein Ergebnis zurückgibt. Der Unterschied zwischen einer Standardanwendung und einer Amazon SWF SWF-Workflow-Anwendung besteht darin, wie der Workflow die Aktivitäten ausführt, wo die Aktivitäten ausgeführt werden und wie die Ergebnisse an den Workflow-Worker zurückgegeben werden.

HelloWorldWorkflow Workflow-Worker

Ein Amazon SWF SWF-Workflow-Worker besteht aus drei grundlegenden Komponenten.

- Eine Workflow-Implementierung. Dies ist eine Klasse, die die Workflow-bezogenen Aufgaben ausführt.
- Eine Activities-Client. Diese ist im Wesentlichen ein Proxy für die Aktivitätsklasse und wird von einer Workflow-Implementierung verwendet, um Aktivitätsmethoden asynchron auszuführen.
- Eine WorkflowWorkerKlasse, die die Interaktion zwischen dem Workflow und Amazon SWF verwaltet.

Dieser Abschnitt beschreibt die Workflow-Implementierung und den Activities-Client. Die `WorkflowWorker`-Klasse wird später besprochen.

HelloWorldWorkflow definiert die Workflow-Schnittstelle in `GreeterWorkflow` wie folgt:

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

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

Diese Schnittstelle ist auch für eine Java-Anwendung nicht unbedingt erforderlich, HelloWorld aber AWS Flow Framework für eine Java-Anwendung unerlässlich. Sie müssen zwei AWS Flow Framework für Java-Anmerkungen @Workflow und @WorkflowRegistrationOptions für die Definition der Workflow-Schnittstelle anwenden. Die Anmerkungen stellen Konfigurationsinformationen bereit und weisen den Annotationsprozessor AWS Flow Framework für Java an, auf der Grundlage der Schnittstelle eine Workflow-Client-Klasse zu generieren, wie später beschrieben wird.

`@Workflow`hat einen optionalen Parameter, DataConverter, der häufig mit seinem Standardwert verwendet wird, der angibt NullDataConverter, dass er verwendet werden JsonDataConverter sollte.

`@WorkflowRegistrationOptions` hat außerdem eine Reihe von optionalen Parametern, die zur Konfiguration des Workflow-Workers verwendet werden können. Hier legen wir `defaultExecutionStartToCloseTimeoutSeconds` — was angibt, wie lange der Workflow ausgeführt werden kann — auf 3600 Sekunden (1 Stunde) fest.

Die `GreeterWorkflow` Schnittstellendefinition unterscheidet sich HelloWorld in einem wichtigen Punkt von der Anmerkung. @Execute Workflow-Schnittstellen legen die Methoden fest, die von Anwendungen wie dem Workflow-Starter aufgerufen werden können. Sie sind auf eine Handvoll Methoden mit jeweils einer bestimmten Rolle beschränkt. Das Framework spezifiziert keinen Namen oder keine Parameterliste für Workflow-Schnittstellenmethoden. Sie verwenden eine Namens- und Parameterliste, die für Ihren Workflow geeignet ist, und fügen eine AWS Flow Framework For-Java-Anmerkung hinzu, um die Rolle der Methode zu identifizieren.

`@Execute` hat zwei Aufgaben:

- Es legt `greet` als Einstiegspunkt des Workflows fest (die Methode, die der Workflow-Starter aufruft, um den Workflow zu starten). Im Allgemeinen kann ein Einstiegspunkt einen oder mehrere

Parameter entgegennehmen. Diese ermöglichen es dem Starter, den Workflow zu initialisieren. Das aktuelle Beispiel erfordert jedoch keine Initialisierung.

- Es legt die Versionsnummer des Workflows fest, über die Sie verschiedene Generationen von Workflow-Implementierungen nachverfolgen können. Um eine Workflow-Oberfläche zu ändern, nachdem Sie sie bei Amazon SWF registriert haben, einschließlich der Änderung der Timeout-Werte, müssen Sie eine neue Versionsnummer verwenden.

Informationen zu den anderen Methoden, die in eine Workflow-Schnittstelle eingebunden werden können, finden Sie unter Workflow- und Aktivitäts-Verträge.

HelloWorldWorkflow implementiert den Workflow wie folgt: `GreeterWorkflowImpl`

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

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

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

Der Code ähnelt dem HelloWorld, weist jedoch zwei wichtige Unterschiede auf.

- `GreeterWorkflowImpl` erzeugt eine Instanz von `GreeterActivitiesClientImpl` (dem Activities-Client) statt von `GreeterActivitiesImpl`, und führt Aktivitäten durch den Aufruf von Methoden für das Client-Objekt aus.
- Der Name und Greeting-Aktivitäten geben `Promise<String>`-Objekte statt `String`-Objekte zurück.

HelloWorld ist eine Java-Standardanwendung, die lokal als ein einziger Prozess ausgeführt wird. `GreeterWorkflowImpl` Sie kann also die Workflow-Topologie implementieren, indem sie einfach eine Instanz von erstelltGreeterActivitiesImpl, die Methoden der Reihe nach aufruft und die Rückgabewerte von einer Aktivität an die nächste weitergibt. Bei einem Amazon SWF SWF-Workflow wird die Aufgabe einer Aktivität immer noch von einer Aktivitätsmethode von ausgeführtGreeterActivitiesImpl. Die Methode wird jedoch nicht notwendigerweise im selben

Prozess wie der Workflow ausgeführt. Sie wird möglicherweise nicht einmal auf demselben System ausgeführt. Der Workflow muss die Aktivität außerdem asynchron ausführen. Diese Anforderungen werfen folgende Probleme auf:

- Wie kann man eine Aktivitätsmethode ausführen, die in einem anderen Prozess oder sogar auf einem anderen System ausgeführt wird?
- Wie kann man eine Aktivitätsmethode asynchron ausführen?
- Wie kann man die Übergabe- und Rückgabewerte von Aktivitäten verwaltet? Wenn der Rückgabewert von Aktivität A beispielsweise an Aktivität B übergeben wird, müssen Sie sicherstellen, dass Aktivität B nicht ausgeführt wird, bis Aktivität A abgeschlossen ist.

Sie können mit der vertrauten Java-Flusssteuerung in Kombination mit dem Activities-Client und `Promise<T>` über den Kontrollfluss der Anwendung eine Vielzahl von Workflow-Topologien implementieren.

Activities-Client

`GreeterActivitiesClientImpl` ist im Grunde ein Proxy für `GreeterActivitiesImpl`, der es einer Workflow-Implementierung ermöglicht, die `GreeterActivitiesImpl`-Methoden asynchron auszuführen.

Die Klassen `GreeterActivitiesClient` und `GreeterActivitiesClientImpl` werden anhand der Angaben in den Annotionen Ihrer `GreeterActivities`-Klasse automatisch generiert. Sie müssen diese nicht selbst implementieren.

> **Note**
>
> Eclipse generiert die Klassen, wenn Sie Ihr Projekt speichern. Sie können den generierten Code im Unterverzeichnis `.apt_generated` Ihres Projektverzeichnisses einsehen. Um Kompilierungsfehler in Ihrer `GreeterWorkflowImpl`-Klasse zu vermeiden, empfiehlt es sich, das Verzeichnis `.apt_generated` auf der Registerkarte Order and Export (Reihenfolge und Export) des Dialogfelds Java-Buildpfad nach ganz oben zu verschieben.

Ein Workflow-Worker führt eine Aktivität aus, indem er die entsprechende Client-Methode aufruft. Die Methode arbeitet asynchron. Sie gibt sofort ein `Promise<T>`-Objekt zurück, wobei T der Rückgabetyp der Aktivität ist. Das zurückgegebene `Promise<T>`-Objekt ist im Grunde ein Platzhalter für den Wert, den die Aktivitätsmethode zurückgeben kann.

- Bei der Rückkehr aus der Activities-Client-Methode befindet sich das `Promise<T>`-Objekt zunächst im Status Unready. Dies bedeutet, dass das Objekt noch keinen gültigen Rückgabewert darstellt.
- Wenn die entsprechende Aktivitätsmethode ihre Aufgabe abschließt und die Ausführung zurückgibt, weist das Framework dem `Promise<T>`-Objekt den Rückgabewert zu und versetzt es in den Zustand Ready.

Promise <T> Type

Der Hauptzweck von `Promise<T>`-Objekten ist die Verwaltung des Datenflusses zwischen asynchronen Komponenten und der Steuerung ihrer Ausführung. Ihre Anwendung muss die Synchronisation nicht explizit verwalten oder von Mechanismen wie Timer nutzen, um sicherzustellen, dass asynchrone Komponenten nicht vorzeitig ausgeführt werden. Wenn Sie eine Activity-Client-Methode aufrufen, gibt sie die Kontrolle sofort zurück. Das Framework verschiebt die Ausführung der entsprechenden Aktivitätsmethode, bis alle übergebenen `Promise<T>`-Objekte bereit sind und gültige Daten enthalten.

Aus der Sicht von `GreeterWorkflowImpl` geben alle drei Aktivity-Client-Methoden die Kontrolle sofort zurück. Aus Sicht von `GreeterActivitiesImpl` ruft das Framework `getGreeting` erst auf, wenn `name` abgeschlossen ist. `say` wird erst aufgerufen, wenn `getGreeting` abgeschlossen ist.

Durch die Verwendung von `Promise<T>` zur Übergabe von Daten von einer Aktivität an die nächste, stellt `HelloWorldWorkflow` nicht nur sicher, dass Aktivitätsmethoden keine ungültigen Daten verwenden, sondern steuert auch, wann die Aktivitäten ausgeführt werden und definiert die Workflow-Topologie. Um den `Promise<T>`-Rückgabewert jeder Aktivität an die nächste Aktivität zu übergeben, müssen die Aktivitäten nacheinander ausgeführt werden. Dies definiert die zuvor beschriebene lineare Topologie. Mit AWS Flow Framework for Java müssen Sie keinen speziellen Modellierungscode verwenden, um selbst komplexe Topologien zu definieren, sondern nur die standardmäßige Java-Flusskontrolle und`Promise<T>`. Ein Beispiel für die Implementierung einer einfachen parallelen Topologie finden Sie unter HelloWorldWorkflowParallelAktivitäten Arbeiter.

> **Note**
>
> Wenn eine Aktivitätsmethode wie `say` keinen Wert zurückgibt, gibt die entsprechende Client-Methode ein `Promise<Void>`-Objekt zurück. Das Objekt repräsentiert keine Daten. Es hat zunächst den Status "Unready". Es ist erst dann bereit, wenn die Aktivität abgeschlossen ist. Sie können ein `Promise<Void>`-Objekt an andere Activity-Client-Methoden übergeben. So

können Sie sicherzustellen, dass diese die Ausführung bis zum Abschluss der ursprünglichen Aktivität verschieben.

`Promise<T>` ermöglicht es einer Workflow-Implementierung, die Activity-Client-Methoden und deren Rückgabewerte ähnlich wie bei synchronen Methoden zu verwenden. Sie müssen allerdings beim Zugriff auf den Wert eines `Promise<T>`-Objekts vorsichtig sein. Im Gegensatz zum Java-Typ Future<T> übernimmt das Framework und nicht die Anwendung die Synchronisation für `Promise<T>`. Wenn Sie `Promise<T>.get` aufrufen und das Objekt nicht bereit ist, löst `get` eine Ausnahme aus. Beachten Sie, dass `HelloWorldWorkflow` nie direkt auf ein `Promise<T>`-Objekt zugreift. Es übergibt die Objekte einfach von einer Aktivität zur nächsten. Wenn ein Objekt bereit ist, extrahiert das Framework den Wert und übergibt ihn als Standardtyp an die Aktivitätsmethode.

Auf `Promise<T>`-Objekte sollte nur über asynchronen Code zugegriffen werden, wobei das Framework gewährleistet, dass das Objekt bereit ist und einen gültigen Wert darstellt. `HelloWorldWorkflow` löst dieses Problem, indem `Promise<T>`-Objekte nur an Methoden des Activities-Clients übergeben werden. Sie können in Ihrer Workflow-Implementierung auf den Wert eines `Promise<T>`-Objekts zugreifen, indem Sie das Objekt an eine asynchrone Workflow-Methode übergeben, die sich wie eine Aktivität verhält. Ein Beispiel finden Sie unter HelloWorldWorkflowAsyncBewerbung.

HelloWorldWorkflow Implementierung von Arbeitsabläufen und Aktivitäten

Den Implementierungen von Workflows und Aktivitäten sind Worker-Klassen zugeordnet, ActivityWorkerund WorkflowWorker. Sie kümmern sich um die Kommunikation zwischen Amazon SWF und den Aktivitäten und Workflow-Implementierungen, indem sie die entsprechende Amazon SWF SWF-Aufgabenliste für Aufgaben abrufen, die entsprechende Methode für jede Aufgabe ausführen und den Datenfluss verwalten. Details hierzu finden Sie unter AWS Flow Framework Grundbegriffe: Anwendungsstruktur

Um die Aktivitäts- und Workflow-Implementierungen mit den entsprechenden Worker-Objekten zu verknüpfen, implementieren Sie eine oder mehrere Worker-Anwendungen. Diese haben die folgenden Aufgaben:

- Registrieren Sie Workflows oder Aktivitäten bei Amazon SWF.
- Erstellen von Worker-Objekten und Zuordnen dieser Objekte zu den Workflow- oder Aktivitäts-Worker-Implementierungen
- Weisen Sie die Worker-Objekte an, mit der Kommunikation mit Amazon SWF zu beginnen.

Wenn Sie den Workflow und die Aktivitäten als getrennte Prozesse ausführen möchten, müssen Sie separate Workflow- und Aktivitäts-Worker-Hosts implementieren. Ein Beispiel finden Sie unter HelloWorldWorkflowDistributed Bewerbung. HelloWorldWorkflowImplementiert der Einfachheit halber einen einzelnen Worker-Host, der Aktivitäten und Workflow-Worker im selben Prozess ausführt, und zwar wie folgt:

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

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

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

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

     String domain = "helloWorldWalkthrough";
     String taskListToPoll = "HelloWorldList";

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

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

`GreeterWorker`hat kein HelloWorld Gegenstück, daher müssen Sie dem Projekt eine Java-Klasse mit `GreeterWorker` dem Namen hinzufügen und den Beispielcode in diese Datei kopieren.

Der erste Schritt besteht darin, ein AmazonSimpleWorkflowClientObjekt zu erstellen und zu konfigurieren, das die zugrunde liegenden Amazon SWF-Servicemethoden aufruft. Hierzu geht `GreeterWorker` folgendermaßen vor:

1. Erstellt ein ClientConfigurationObjekt und gibt ein Socket-Timeout von 70 Sekunden an. Dieser Wert gibt an, wie lange auf die Übertragung der Daten über eine bestehende offene Verbindung gewartet wird, bevor der Socket geschlossen wird.
2. Erstellt ein AWSCredentialsBasic-Objekt zur Identifizierung des AWS Kontos und übergibt die Kontoschlüssel an den Konstruktor. Zur Vereinfachung und um diese nicht als Klartext im Code zu hinterlegen, werden die Schlüssel als Umgebungsvariablen gespeichert.
3. Erstellt ein AmazonSimpleWorkflowClientObjekt zur Darstellung des Workflows und übergibt die `ClientConfiguration` Objekte `BasicAWSCredentials` und an den Konstruktor.
4. Legt die Service-Endpunkt-URL des Client-Objekts fest. Amazon SWF ist derzeit in allen AWS Regionen verfügbar.

Der Einfachheit halber definiert `GreeterWorker` zwei String-Konstanten.

- `domain`ist der Amazon SWF-Domainname des Workflows, den Sie bei der Einrichtung Ihres Amazon SWF SWF-Kontos erstellt haben. `HelloWorldWorkflow`geht davon aus, dass Sie den Workflow in der Domäne "helloWorldWalkthrough" ausführen.
- `taskListToPoll`ist der Name der Aufgabenlisten, die Amazon SWF verwendet, um die Kommunikation zwischen den Workflow- und Aktivitätsmitarbeitern zu verwalten. Sie können den Namen auf eine beliebige beliebige Zeichenfolge setzen. HelloWorldWorkflow verwendet "HelloWorldList" sowohl für Workflow- als auch für Aktivitätsaufgabenlisten. Hinter den Kulissen werden die Namen in verschiedene Namespaces umgesetzt. Daher bleiben beide Aufgabenlisten unterscheidbar.

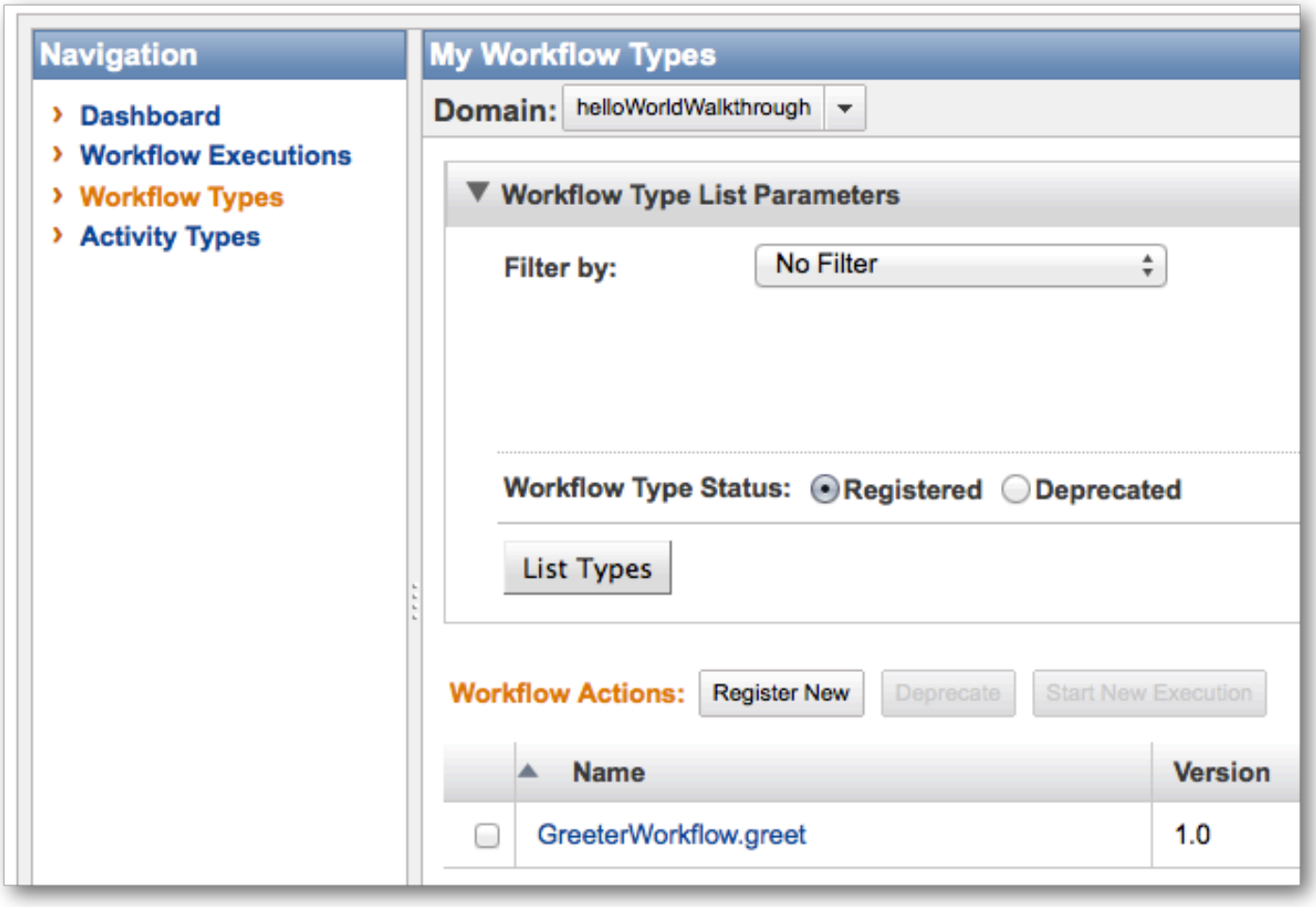
`GreeterWorker`verwendet die Zeichenkettenkonstanten und das AmazonSimpleWorkflowClientObjekt, um Worker-Objekte zu erstellen, die die Interaktion zwischen den Aktivitäten und Worker-Implementierungen und Amazon SWF verwalten. Insbesondere übernehmen die Worker-Objekte die Aufgabe, die entsprechende Aufgabenliste für Aufgaben abzufragen.

`GreeterWorker` erstellt ein `ActivityWorker`-Objekt und konfiguriert es so, dass es `GreeterActivitiesImpl` behandelt, indem es eine neue Klasseninstance hinzufügt.

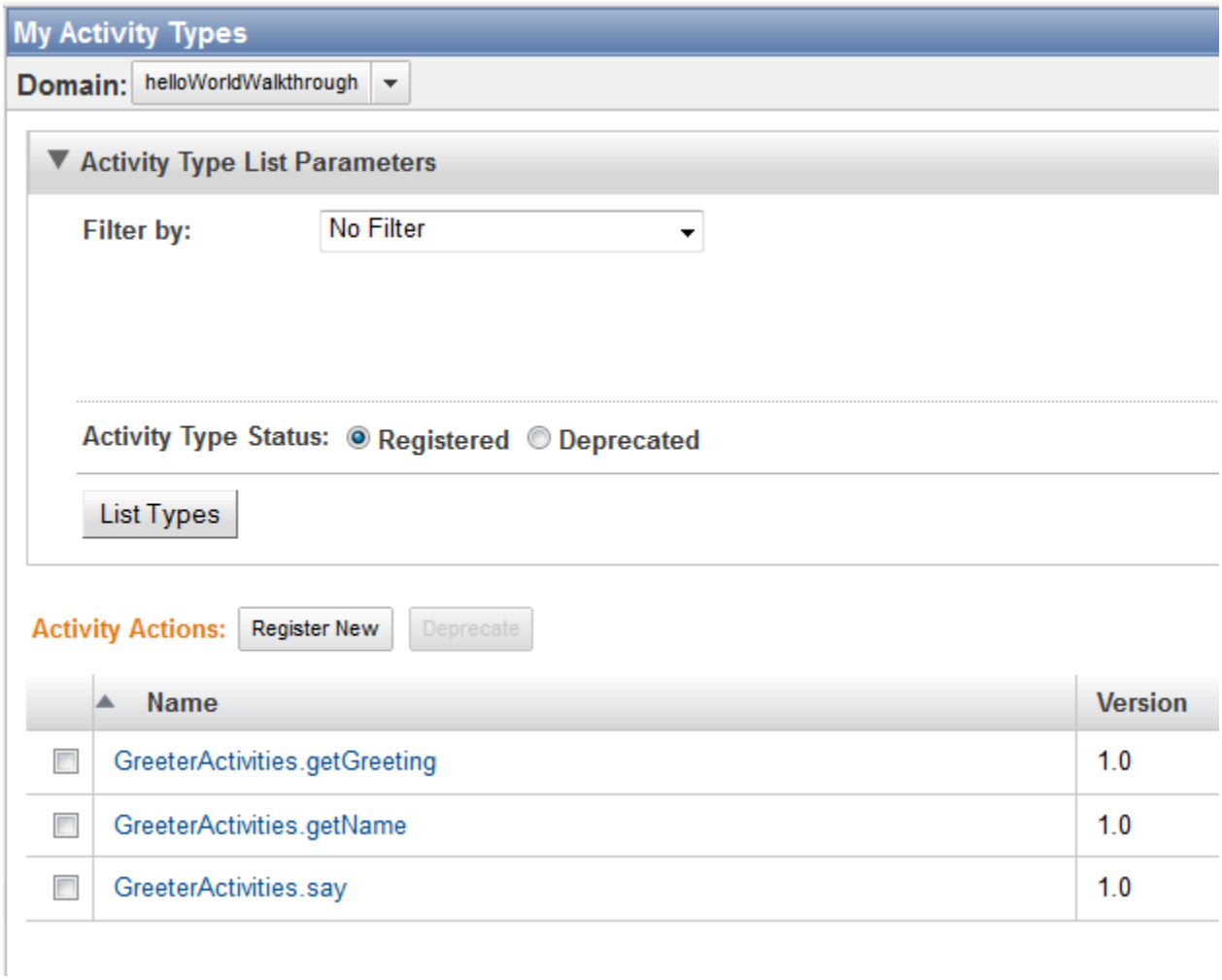
`GreeterWorker` ruft dann die `start`-Methode des `ActivityWorker`-Objekts auf, die das Objekt anweist, mit der Abfrage der angegebenen Aktivitätsaufgabenliste zu beginnen.

`GreeterWorker` erzeugt ein `WorkflowWorker`-Objekt und konfiguriert es über das Hinzufügen des Klassen-Dateinamens `GreeterWorkflowImpl.class` so, dass es `GreeterWorkflowImpl` nutzt. Es ruft dann die `WorkflowWorker`-Methode des `start`-Objekts auf, die das Objekt anweist, die angegebene Workflow-Aufgabenliste abzufragen.

Sie können `GreeterWorker` nun erfolgreich ausführen. Es registriert den Workflow und die Aktivitäten bei Amazon SWF und startet, dass die Worker-Objekte ihre jeweiligen Aufgabenlisten abfragen. Um dies zu überprüfen, starten `GreeterWorker` Sie die Amazon SWF SWF-Konsole, rufen Sie sie auf und wählen Sie eine Domain `helloWorldWalkthrough` aus der Liste der Domains aus. Wenn Sie Workflow Types (Workflow-Typen) im Bereich Navigation auswählen, sollten Sie `GreeterWorkflow.greet` sehen:



Wenn Sie Aktivity Types (Aktivitätstypen) auswählen, werden die `GreeterActivities`-Methoden angezeigt:



Wenn Sie Workflow Executions (Workflow-Ausführungen) auswählen, sehen Sie jedoch keine aktiven Ausführungen. Die Workflow- und Aktivitäts-Worker suchen zwar nach Aufgaben, aber wir haben noch keine Workflow-Ausführung gestartet.

HelloWorldWorkflow Vorspeise

Als letztes muss ein Workflow-Starter implementiert werden – eine Anwendung, die die Workflow-Ausführung startet. Der Ausführungsstatus wird von Amazon SWF gespeichert, sodass Sie dessen Verlauf und Ausführungsstatus einsehen können. HelloWorldWorkflow implementiert einen Workflow-Starter, indem die `GreeterMain` Klasse wie folgt geändert wird:

```
import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
```

```
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient;

public class GreeterMain {

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

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

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

     String domain = "helloWorldWalkthrough";

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

`GreeterMain` erzeugt ein `AmazonSimpleWorkflowClient`-Objekt mit dem gleichen Code wie `GreeterWorker`. Es stellt dann ein `GreeterWorkflowClientExternal`-Objekt, das als Proxy für den Workflow fungiert (ähnlich wie der in `GreeterWorkflowClientImpl` angelegte Activity-Client als Proxy für die Aktivitätsmethoden agiert). Anstatt ein Workflow-Client-Objekt mit `new` anzulegen, gehen Sie folgendermaßen vor:

1. Erstellen Sie ein externes Client-Factory-Objekt und übergeben Sie das `AmazonSimpleWorkflowClient` Objekt und den Amazon SWF-Domänennamen an den Konstruktor. Das Client-Factory-Objekt wird vom Annotationsprozessor des Frameworks erstellt, der den Objektnamen erstellt, indem einfach "ClientExternalFactoryImpl" an den Namen der Workflow-Schnittstelle angehängt wird.
2. Erstellen Sie ein externes Client-Objekt, indem Sie die `getClient` Methode des Factory-Objekts aufrufen, die den Objektnamen erstellt, indem "ClientExternal" an den Namen der Workflow-Schnittstelle angehängt wird. Sie können optional `getClient` eine Zeichenfolge übergeben, die

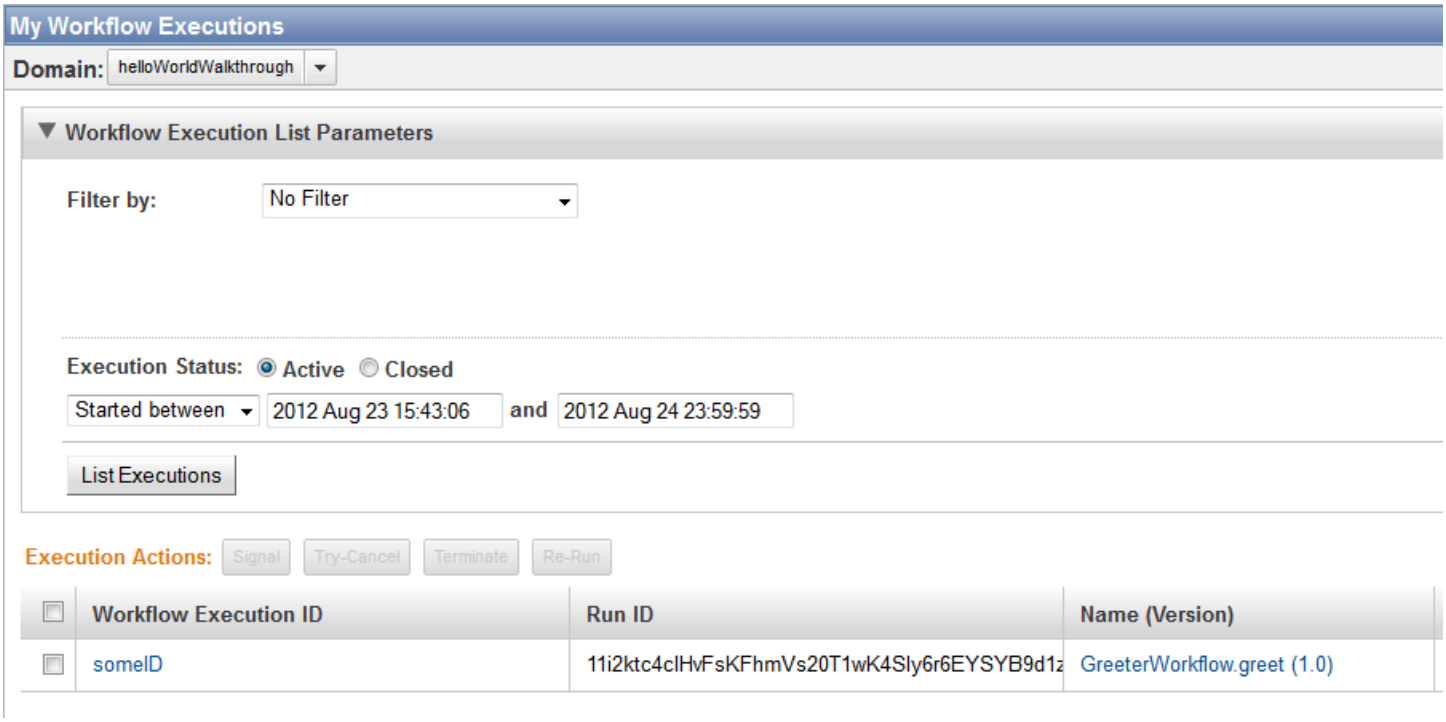
Amazon SWF verwendet, um diese Instanz des Workflows zu identifizieren. Andernfalls stellt Amazon SWF eine Workflow-Instanz mithilfe einer generierten GUID dar.

Der von der Factory zurückgegebene Client erstellt nur Workflows, die mit der Zeichenfolge benannt sind, die an die Methode getClient übergeben wurde (der von der Factory zurückgegebene Client hat bereits den Status in Amazon SWF). Um einen Workflow mit einer anderen ID auszuführen, müssen Sie zurück zur Factory wechseln und einen neuen Client mit der anderen ID anlegen.

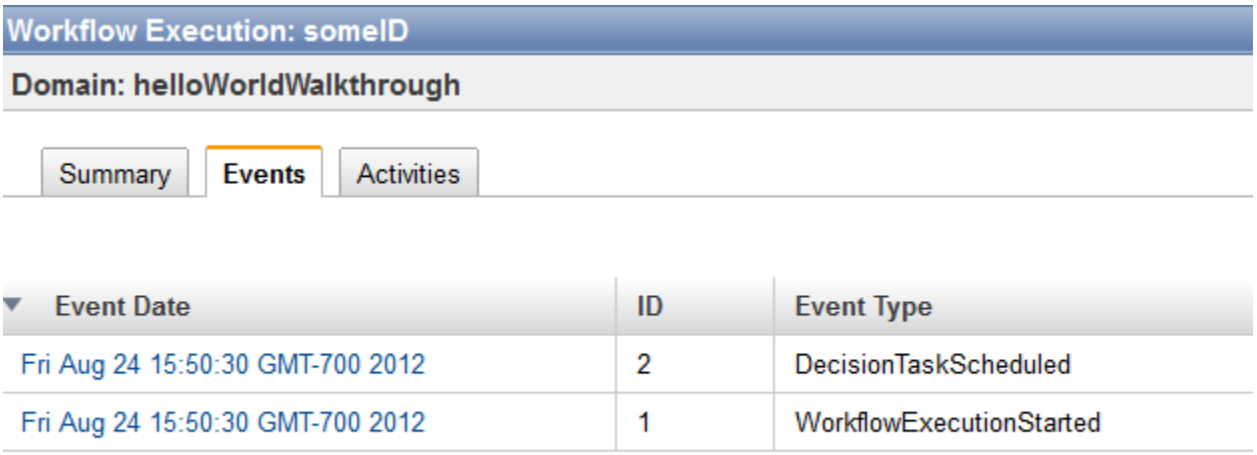
Der Workflow-Client stellt eine `greet`-Methode zur Verfügung, die `GreeterMain` aufruft, um den Workflow zu starten (da `greet()` die mit der `@Execute`-Annotation angegebene Methode war).

> **Note**
>
> Der Annotationsprozessor erzeugt außerdem ein internes Client-Factory-Objekt, das zur Erstellung von untergeordneten Workflows verwendet wird. Details hierzu finden Sie unter Untergeordnete Workflow-Ausführungen.

Beenden Sie `GreeterWorker` (falls noch ausgeführt). Starten Sie `GreeterMain`. Sie sollten jetzt SomeID in der Liste der aktiven Workflow-Ausführungen der Amazon SWF SWF-Konsole sehen:.



Wenn Sie `someID` und die Registerkarte Events (Ereignisse) auswählen, werden die Ereignisse angezeigt:

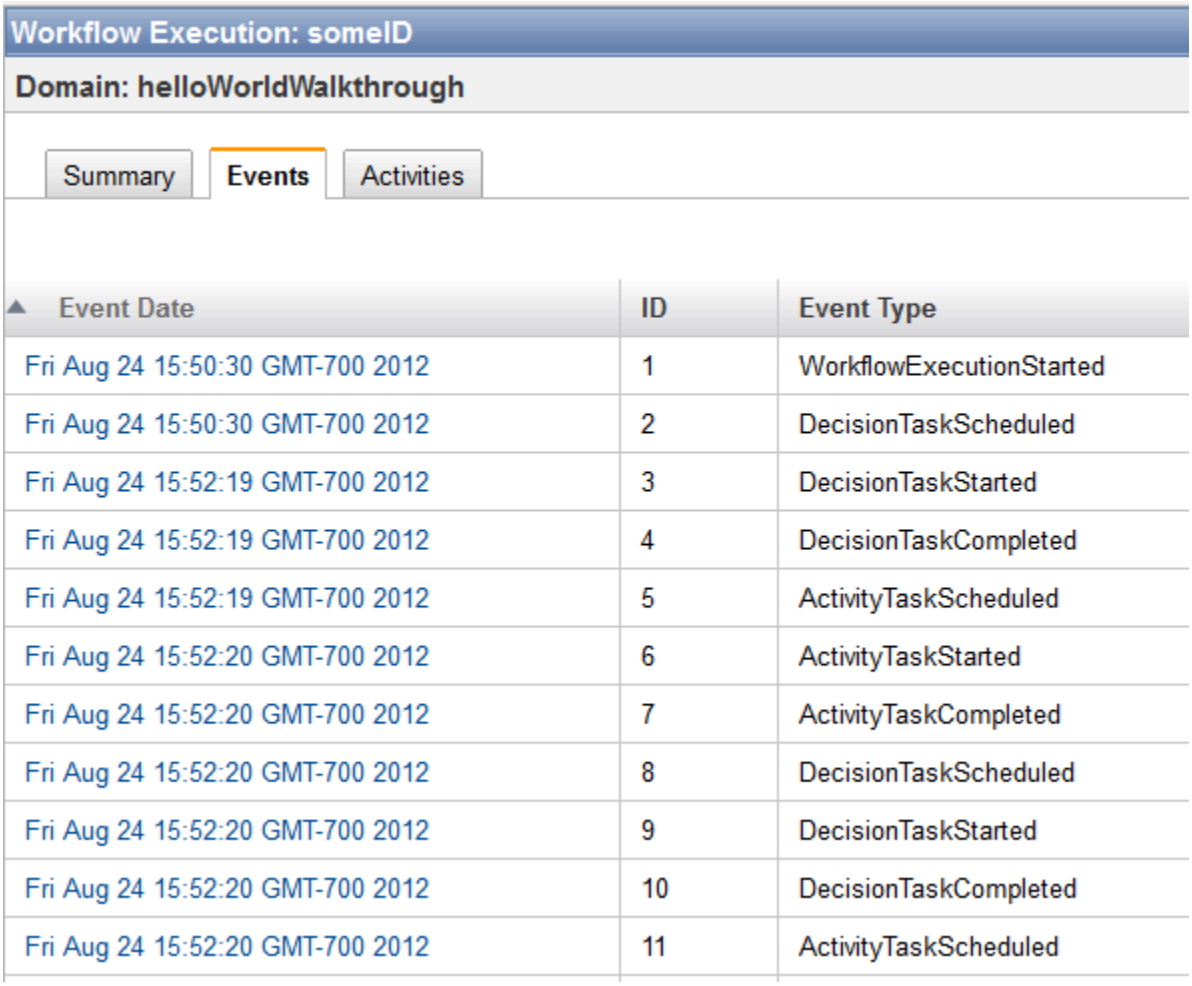


> **Note**
>
> Wenn Sie `GreeterWorker` bereits früher gestartet haben und es noch ausgeführt wird, sehen Sie eine längere Ereignisliste. Die Gründe hierfür werden gleich besprochen. Halten Sie `GreeterWorker` an und versuchen Sie erneut, `GreaterMain` zu starten.

Die Registerkarte Events (Ereignisse) zeigt nur zwei Ereignisse an:

- `WorkflowExecutionStarted` zeigt an, dass der Workflow mit der Ausführung begonnen hat.
- `DecisionTaskScheduled`gibt an, dass Amazon SWF die erste Entscheidungsaufgabe in die Warteschlange gestellt hat.

Der Grund dafür, dass der Workflow bei der ersten Entscheidungsaufgabe blockiert wird, ist, dass der Workflow auf zwei Anwendungen verteilt ist, `GreeterMain` und `GreeterWorker`. `GreeterMain` die Workflow-Ausführung gestartet haben, aber `GreeterWorker` nicht läuft, sodass die Worker die Listen nicht abfragen und Aufgaben ausführen. Sie können beide Anwendungen unabhängig voneinander ausführen. Sie benötigen jedoch beide, damit die Workflow-Ausführung über die erste Entscheidungsaufgabe hinausgeht. Wenn Sie nun `GreeterWorker` ausführen, beginnen die Workflow- und Aktivitäts-Worker mit dem Abrufen und die verschiedenen Aufgaben werden schnell abgeschlossen. Wenn Sie nun die Registerkarte `Events` prüfen, wird die erste Ereignisgruppe angezeigt.



Workflow Execution: someID

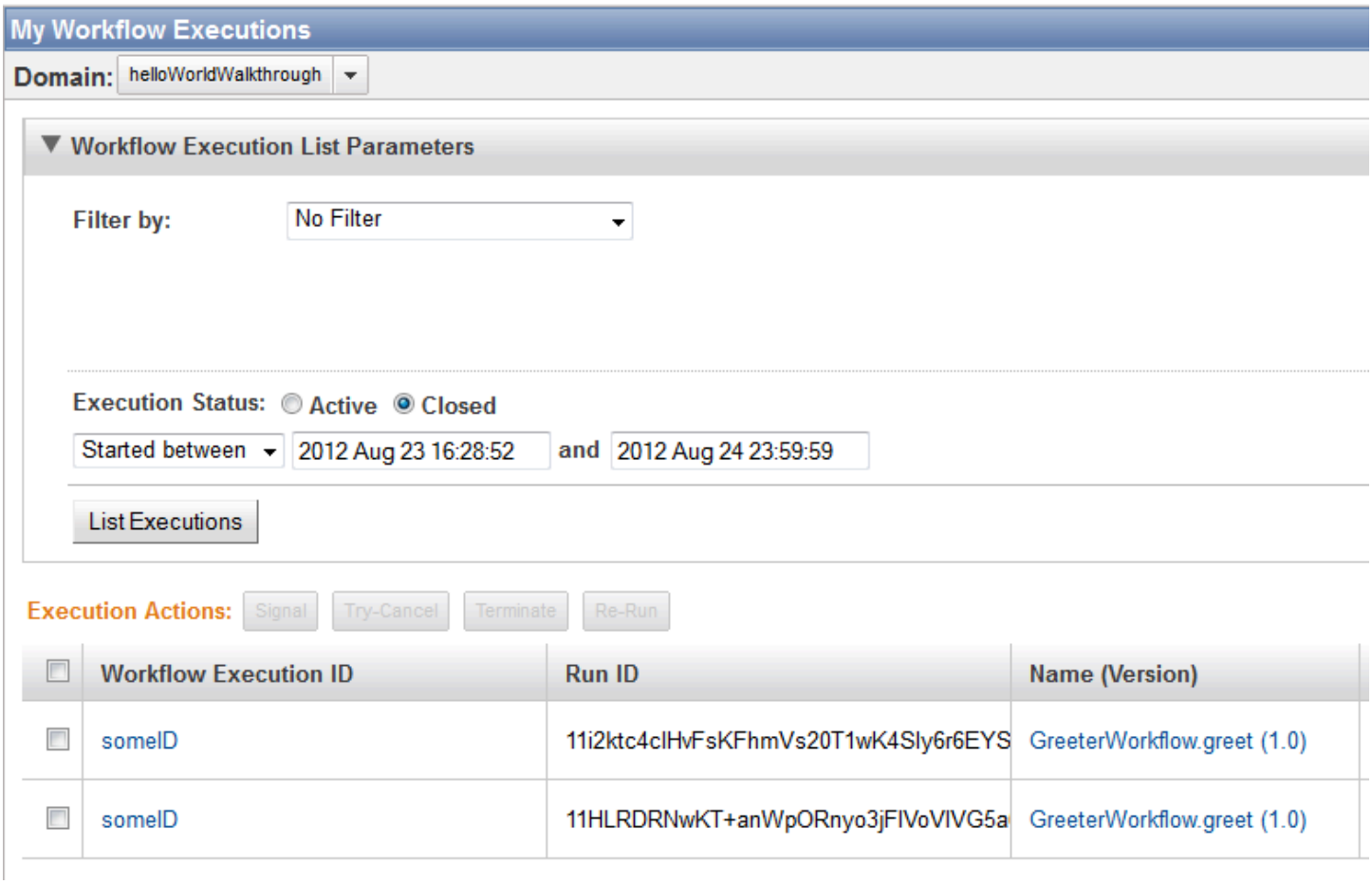
Domain: helloWorldWalkthrough

Summary | Events | Activities

Event Date	ID	Event Type
Fri Aug 24 15:50:30 GMT-700 2012	1	WorkflowExecutionStarted
Fri Aug 24 15:50:30 GMT-700 2012	2	DecisionTaskScheduled
Fri Aug 24 15:52:19 GMT-700 2012	3	DecisionTaskStarted
Fri Aug 24 15:52:19 GMT-700 2012	4	DecisionTaskCompleted
Fri Aug 24 15:52:19 GMT-700 2012	5	ActivityTaskScheduled
Fri Aug 24 15:52:20 GMT-700 2012	6	ActivityTaskStarted
Fri Aug 24 15:52:20 GMT-700 2012	7	ActivityTaskCompleted
Fri Aug 24 15:52:20 GMT-700 2012	8	DecisionTaskScheduled
Fri Aug 24 15:52:20 GMT-700 2012	9	DecisionTaskStarted
Fri Aug 24 15:52:20 GMT-700 2012	10	DecisionTaskCompleted
Fri Aug 24 15:52:20 GMT-700 2012	11	ActivityTaskScheduled

Sie können einzelne Ereignisse auswählen, um weitere Informationen zu erhalten. Wenn Sie mit der Suche fertig sind, sollte der Workflow „Hello World!“ gedruckt haben auf deine Konsole.

Nach dem Abschluss des Workflows erscheint er nicht mehr in der Liste der aktiven Ausführungen. Wenn Sie dies überprüfen möchten, wählen Sie die Schaltfläche für den Ausführungsstatus Closed (Geschlossen) und dann List Executions (Ausführungen auflisten) aus. Es werden alle abgeschlossenen Workflow-Instances in der angegebenen Domäne (`helloWorldWalkthrough`) angezeigt (die die beim Anlegen der Domäne angegebene Aufbewahrungszeit nicht überschritten haben).



Beachten Sie, dass jede Workflow-Instance einen eindeutigen Run ID-Wert hat. Sie können dieselbe Workflow-ID für verschiedene Workflow-Instanzen verwenden, jedoch jeweils nur für eine aktive Ausführung.

HelloWorldWorkflowAsyncBewerbung

Gelegentlich ist es vorteilhaft, einen Workflow bestimmte Aufgaben lokal durchführt zu lassen, statt eine Aktivität zu verwenden. Jedoch umfassen Workflow-Aufgaben häufig die Verarbeitung der Werte, die von `Promise<T>`-Objekten repräsentiert werden. Wenn Sie ein `Promise<T>`-Objekt an eine synchrone Workflow-Methode weiterleiten, wird die Methode sofort ausgeführt, aber sie kann nicht auf den Wert des `Promise<T>`-Objekts zugreifen, bevor das Objekt bereit ist. Sie könnten `Promise<T>.isReady` abfragen, bis es `true` zurückgibt, dies ist jedoch ineffizient und die Methode könnte lange blockiert sein. Eine besserer Ansatz ist das Verwenden einer asynchronen Methode.

Eine asynchrone Methode wird ähnlich wie eine Standardmethode implementiert — oft als Mitglied der Workflow-Implementierungsklasse — und wird im Kontext der Workflow-Implementierung

ausgeführt. Sie legen sie als asynchrone Methode fest, indem Sie eine `@Asynchronous`-Anmerkung anwenden, wodurch das Framework angewiesen wird, sie ähnlich wie eine Aktivität zu behandeln.

- Wenn eine Workflow-Implementierung eine asynchrone Methode aufruft, wird sie sofort zurückgegeben. Asynchrone Methoden geben in der Regel ein `Promise<T>`-Objekt zurück, das verfügbar wird, wenn die Methode abgeschlossen ist.
- Wenn Sie einer asynchronen Methode eine oder mehrere `Promise<T>`-Objekte übergeben, verschiebt sie die Ausführung, bis alle Eingabeobjekte bereit sind. Eine asynchrone Methode kann daher auf ihre `Promise<T>`-Werte der Eingabe zugreifen, ohne eine Ausnahme zu riskieren.

> **Note**
>
> Aufgrund der Art und Weise, wie die AWS Flow Framework für Java den Workflow ausführt, werden asynchrone Methoden in der Regel mehrfach ausgeführt. Sie sollten sie daher nur für schnelle Aufgaben mit geringem Overhead verwenden. Aktivitäten sollten Sie zur Durchführung zeitintensiver Aufgaben wie großen Berechnungen verwenden. Details hierzu finden Sie unter AWS Flow Framework Grundbegriffe: Verteilte Ausführung.

Dieses Thema ist eine exemplarische Vorgehensweise für eine modifizierte Version HelloWorldWorkflowAsync, HelloWorldWorkflow die eine der Aktivitäten durch eine asynchrone Methode ersetzt. Um die Anwendung zu implementieren, erstellen Sie eine Kopie von HelloWorld. HelloWorldWorkflow packe es in dein Projektverzeichnis und nenne es HelloWorld. HelloWorldWorkflowAsync.

> **Note**
>
> Dieses Thema baut auf den Konzepten und Dateien auf, die in den Themen HelloWorld Bewerbung und HelloWorldWorkflow Bewerbung vorgestellt werden. Machen Sie sich mit den Dateien und vorgestellten Konzepten finden Sie in diesen Themen, bevor Sie fortfahren.

In den folgenden Abschnitten wird beschrieben, wie der ursprüngliche HelloWorldWorkflow Code geändert wird, um eine asynchrone Methode zu verwenden.

HelloWorldWorkflowAsync Implementierung der Aktivitäten

HelloWorldWorkflowAsync implementiert seine Worker-Schnittstelle für Aktivitäten wie folgt: `GreeterActivities`

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

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

Diese Schnittstelle ähnelt der von verwendeten HelloWorldWorkflow, mit den folgenden Ausnahmen:

- Sie lässt die `getGreeting`-Aktivität weg. Diese Aufgabe wird jetzt von einer asynchronen Methode verarbeitet.
- Die Versionsnummer wird auf 2.0. Nachdem Sie eine Aktivitätsschnittstelle bei Amazon SWF registriert haben, können Sie sie nur ändern, wenn Sie die Versionsnummer ändern.

Die übrigen Implementierungen der Aktivitätsmethoden sind identisch mit. HelloWorldWorkflow Löschen Sie einfach `getGreeting` aus `GreeterActivitiesImpl`.

HelloWorldWorkflowAsync Workflow-Implementierung

HelloWorldWorkflowAsync definiert die Workflow-Schnittstelle wie folgt:

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

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 3600)
public interface GreeterWorkflow {
```

```
   @Execute(version = "2.0")
   public void greet();
}
```

Die Schnittstelle ist bis auf HelloWorldWorkflow eine neue Versionsnummer identisch mit. Wenn Sie einen registrierten Workflow ändern möchten, müssen Sie wie bei Aktivitäten seine Version ändern.

HelloWorldWorkflowAsync implementiert den Workflow wie folgt:

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

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

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

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

HelloWorldWorkflowAsync ersetzt die `getGreeting` Aktivität durch eine `getGreeting` asynchrone Methode, aber die `greet` Methode funktioniert fast genauso:

1. Führen Sie die `getName`-Aktivität aus, die sofort ein `Promise<String>`-Objekt, `name`, zurückgibt, das den Namen repräsentiert.
2. Rufen Sie die asynchrone Methode `getGreeting` auf und übergeben Sie ihr das `name`-Objekt. `getGreeting` gibt umgehend ein `Promise<String>`-Objekt, `greeting`, zurück, das die Begrüßung repräsentiert.
3. Führen Sie die `say`-Aktivität aus und übergeben Sie ihr das `greeting`-Objekt.
4. Wenn `getName` abgeschlossen wird, ist `name` einsatzbereit und `getGreeting` verwendet seinen Wert zur Erstellung der Begrüßung.

5. Wenn `getGreeting` abgeschlossen wird, ist `greeting` einsatzbereit und `say` gibt die Zeichenfolge in der Konsole aus.

Der Unterschied liegt darin, dass Greet (Gruß) nicht den Aktivitäten-Client aufruft, um eine `getGreeting`-Aktivität auszuführen, sondern die asynchrone `getGreeting`-Methode. Das Endergebnis ist dasselbe, aber die `getGreeting`-Methode funktioniert etwas anders als die `getGreeting`-Aktivität.

- Der Workflow-Worker verwendet Aufrufsemantiken der Standardfunktion für die Ausführung von `getGreeting`. Die asynchrone Ausführung der Aktivität wird jedoch von Amazon SWF vermittelt.
- `getGreeting` wird im Prozess der Workflow-Implementierung ausgeführt.
- `getGreeting` gibt ein `Promise<String>`-Objekt anstelle eines `String`-Objekts zurück. Um den Zeichenfolgewert abzurufen, der sich im Besitz von `Promise` befindet, rufen Sie seine `get()`-Methode auf. Da die Aktivität jedoch asynchron ausgeführt wird, ist ihr Rückgabewert möglicherweise nicht sofort bereit. Es `get()` wird eine Ausnahme ausgelöst, bis der Rückgabewert der asynchronen Methode verfügbar ist.

  Weitere Informationen zur Funktionsweise von `Promise` finden Sie unter AWS Flow Framework Grundbegriffe: Data Exchange zwischen Aktivitäten und Workflows.

`getGreeting` erstellt einen Rückgabewert, indem die Begrüßungszeichenfolge an die statische `Promise.asPromise`-Methode übergeben wird. Diese Methode erstellt ein `Promise<T>`-Objekt des entsprechenden Typs, legt den Wert fest und versetzt es in den betriebsbereiten Zustand.

HelloWorldWorkflowAsyncArbeitsablauf und Aktivitäten: Host und Starter

HelloWorldWorkflowAsync implementiert `GreeterWorker` als Hostklasse für die Workflow- und Aktivitätsimplementierungen. Sie ist mit der HelloWorldWorkflow Implementierung identisch, mit Ausnahme des `taskListToPoll` Namens, der auf "`HelloWorldAsyncList`" gesetzt ist.

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

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

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

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

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

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

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

HelloWorldWorkflowAsync implementiert den Workflow-Starter in`GreeterMain`; er ist identisch mit der HelloWorldWorkflow Implementierung.

Um den Workflow auszuführen, führen Sie `GreeterWorker` und aus`GreeterMain`, genau wie bei HelloWorldWorkflow.

HelloWorldWorkflowDistributed Bewerbung

Mit HelloWorldWorkflow und HelloWorldWorkflowAsync vermittelt Amazon SWF die Interaktion zwischen den Implementierungen des Workflows und der Aktivitäten, sie werden jedoch lokal als ein einziger Prozess ausgeführt. `GreeterMain`befindet sich in einem separaten Prozess, läuft aber immer noch auf demselben System.

Ein wesentliches Merkmal von Amazon SWF ist die Unterstützung verteilter Anwendungen. Sie könnten beispielsweise den Workflow-Worker auf einer EC2 Amazon-Instance, den Workflow-

Starter auf einem Rechenzentrumscomputer und die Aktivitäten auf einem Client-Desktop-Computer ausführen. Sie können sogar unterschiedliche Aktivitäten auf unterschiedlichen Systemen ausführen.

Die HelloWorldWorkflowDistributed Anwendung erstreckt sich HelloWorldWorkflowAsync auf die Verteilung der Anwendung auf zwei Systeme und drei Prozesse.

- Der Workflow und der Workflow-Starter werden als getrennte Prozesse auf einem System ausgeführt.
- Die Aktivitäten werden auf einem getrennten System ausgeführt.

Um die Anwendung zu implementieren, erstellen Sie eine Kopie von HelloWorld. HelloWorldWorkflowAsync packe es in dein Projektverzeichnis und nenne es HelloWorld. HelloWorldWorkflowDistributed. In den folgenden Abschnitten wird beschrieben, wie Sie den HelloWorldWorkflowAsync Originalcode ändern, um die Anwendung auf zwei Systeme und drei Prozesse zu verteilen.

Sie müssen den Workflow oder das Implementieren der Aktivitäten nicht ändern, um sie auf getrennten Systemen auszuführen, auch nicht die Versionsnummern. Sie müssen `GreeterMain` auch nicht ändern. Sie müssen lediglich den Aktivitäten- und Workflow-Host ändern.

Dabei HelloWorldWorkflowAsync dient eine einzige Anwendung als Host für den Workflow und die Aktivität. Um den Workflow und das Implementieren der Aktivitäten auf getrennten Systemen auszuführen, müssen Sie getrennte Anwendungen implementieren. GreeterWorker Aus dem Projekt löschen und zwei neue Klassendateien hinzufügen, GreeterWorkflowWorker und GreeterActivitiesWorker.

HelloWorldWorkflowDistributed implementiert seinen Aktivitäten-Host in GreeterActivitiesWorker wie folgt:

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

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

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

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

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

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

HelloWorldWorkflowDistributed implementiert seinen Workflow-Host wie folgt: `GreeterWorkflowWorker`

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

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

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

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

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

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

Beachten Sie, dass `GreeterActivitiesWorker` nur `GreeterWorker` ohne den `WorkflowWorker`-Code ist und `GreeterWorkflowWorker` nur `GreeterWorker` ohne den `ActivityWorker`-Code ist.

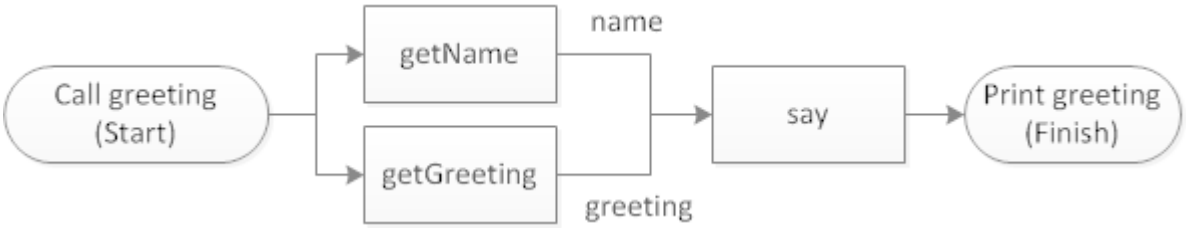
So führen Sie den Workflow aus:

1. Erstellen Sie eine ausführbare JAR-Datei mit `GreeterActivitiesWorker` als Eingangspunkt.
2. Kopieren Sie die JAR-Datei aus Schritt 1 in ein anderes System, das jedes von Java unterstütztes Betriebssystem ausführen kann.
3. Stellen Sie sicher, dass AWS Anmeldeinformationen mit Zugriff auf dieselbe Amazon SWF-Domain auf dem anderen System verfügbar sind.
4. Führen Sie die JAR-Datei aus.
5. Verwenden Sie auf Ihrem Entwicklungssystem Eclipse zum Ausführen von `GreeterWorkflowWorker` und `GreeterMain`.

Abgesehen von der Tatsache, dass die Aktivitäten auf einem anderen System als dem Workflow-Worker und dem Workflow-Starter ausgeführt werden, funktioniert der Workflow genauso wie HelloWorldAsync. Allerdings, weil das `println` Aufrufen „Hello World!“ ausgibt Wenn sich die `say` Aktivität auf der Konsole befindet, erscheint die Ausgabe auf dem System, auf dem der Activities Worker ausgeführt wird.

HelloWorldWorkflowParallelBewerbung

In den Vorgängerversionen von Hello World! wird eine lineare Workflow-Topologie verwendet. Amazon SWF ist jedoch nicht auf lineare Topologien beschränkt. Die HelloWorldWorkflowParallel Anwendung ist eine modifizierte Version davon HelloWorldWorkflow , die eine parallel Topologie verwendet, wie in der folgenden Abbildung dargestellt.



Mit HelloWorldWorkflowParallel, `getName` und parallel `getGreeting` laufen und jeweils einen Teil der Begrüßung zurückgeben. `say`führt dann die beiden Zeichenketten zu einer Begrüßung zusammen und druckt sie auf der Konsole aus.

Um die Anwendung zu implementieren, erstellen Sie eine Kopie von HelloWorld. HelloWorldWorkflow packe es in dein Projektverzeichnis und nenne es HelloWorld. HelloWorldWorkflowParallel. In den folgenden Abschnitten wird beschrieben, wie Sie den HelloWorldWorkflow Originalcode so ändern, dass er `getGreeting` parallel ausgeführt `getName` wird.

HelloWorldWorkflowParallelAktivitäten Arbeiter

Die HelloWorldWorkflowParallel Aktivitätsschnittstelle ist in implementiert`GreeterActivities`, wie im folgenden Beispiel gezeigt.

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

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

Die Schnittstelle ist ähnlich wie HelloWorldWorkflow, mit den folgenden Ausnahmen:

- `getGreeting` übernimmt keine Eingabe. Sie gibt nur eine Begrüßungszeichenfolge zurück.
- `say` übernimmt zwei Eingabezeichenfolgen, die Begrüßung und den Namen.
- Die Schnittstelle hat eine neue Versionsnummer. Diese ist bei jeder Änderung an der registrierten Schnittstelle erforderlich.

HelloWorldWorkflowParallel implementiert die Aktivitäten wie folgt: `GreeterActivitiesImpl`

```
public class GreeterActivitiesImpl implements GreeterActivities {

   @Override
   public String getName() {
      return "World!";
   }

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

   @Override
   public void say(String greeting, String name) {
      System.out.println(greeting + name);
   }
}
```

`getName` und `getGreeting` geben nun einfach die Hälfte der Begrüßungszeichenkette zurück. `say` verkettet die beiden Teile, um die vollständige Zeichenfolge zu erzeugen, und gibt sie auf der Konsole aus.

HelloWorldWorkflowParallelWorkflow-Mitarbeiter

Die HelloWorldWorkflowParallel Workflow-Schnittstelle ist wie folgt implementiert: `GreeterWorkflow`

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

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 3600)
public interface GreeterWorkflow {

   @Execute(version = "5.0")
   public void greet();
}
```

Die Klasse ist identisch mit der HelloWorldWorkflow Version, mit der Ausnahme, dass die Versionsnummer so geändert wurde, dass sie dem Activities Worker entspricht.

Der Workflow wird in `GreeterWorkflowImpl` wie folgt implementiert:

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

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

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

Auf den ersten Blick sieht diese Implementierung sehr ähnlich aus wie die drei Aktivitäten HelloWorldWorkflow, die die Client-Methoden nacheinander ausführen. die Aktivitäten jedoch nicht.

- HelloWorldWorkflow übergeben `name` an`getGreeting`. Da `name` ein `Promise<T>`-Objekt ist, verschiebt `getGreeting` die Ausführung der Aktivität, bis `getName` abgeschlossen ist. Daher werden die beiden Aktivitäten nacheinander ausgeführt.
- HelloWorldWorkflowParallel übergibt keine Eingabe `getName` oder`getGreeting`. Keine der Methoden verschiebt die Ausführung und die zugehörigen Aktivitätsmethoden werden sofort parallel ausgeführt.

Die Aktivität `say` übernimmt sowohl `greeting` als auch `name` als Eingabeparameter. Da es sich dabei um `Promise<T>`-Objekte handelt, verschiebt `say` die Ausführung, bis beide Aktivitäten abgeschlossen sind, erstellt dann die Begrüßung und gibt sie aus.

Beachten Sie, dass HelloWorldWorkflowParallel kein spezieller Modellierungscode verwendet wird, um die Workflow-Topologie zu definieren. Dies geschieht implizit, indem es die standardmäßige Java-Ablaufsteuerung verwendet und die Eigenschaften von `Promise<T>` Objekten ausnutzt. AWS Flow Framework für Java-Anwendungen können selbst komplexe Topologien einfach durch die Verwendung von `Promise<T>` Objekten in Verbindung mit herkömmlichen Java-Kontrollflusskonstrukten implementiert werden.

HelloWorldWorkflowParallel Arbeitsablauf und Aktivitäten: Host und Starter

HelloWorldWorkflowParallel implementiert `GreeterWorker` als Hostklasse für die Workflow- und Aktivitätsimplementierungen. Sie ist mit der HelloWorldWorkflow Implementierung identisch, mit Ausnahme des `taskListToPoll` Namens, der auf "HelloWorldParallelList" gesetzt ist.

`HelloWorldWorkflowParallel`implementiert den `GreeterMain` Workflow-Starter in und ist mit der HelloWorldWorkflow Implementierung identisch.

Führen Sie zur Ausführung des Workflows `GreeterWorker` und `GreeterMain` genau wie bei `HelloWorldWorkflow` aus.

Verständnis AWS Flow Framework für Java

The AWS Flow Framework for Java arbeitet mit Amazon SWF zusammen, um die Erstellung skalierbarer und fehlertoleranter Anwendungen für asynchrone Aufgaben zu vereinfachen, die lange dauern, remote oder beides ausführen können. Das „Hallo Welt!“ In den Beispielen Was ist das AWS Flow Framework für Java? wurden die Grundlagen der Verwendung von AWS Flow Framework zur Implementierung grundlegender Workflow-Anwendungen vorgestellt. Dieser Abschnitt enthält grundlegende Informationen zur Funktionsweise von AWS Flow Framework Anwendungen. Der erste Abschnitt fasst die grundlegende Struktur einer AWS Flow Framework Anwendung zusammen, und die übrigen Abschnitte enthalten weitere Einzelheiten zur Funktionsweise von AWS Flow Framework Anwendungen.

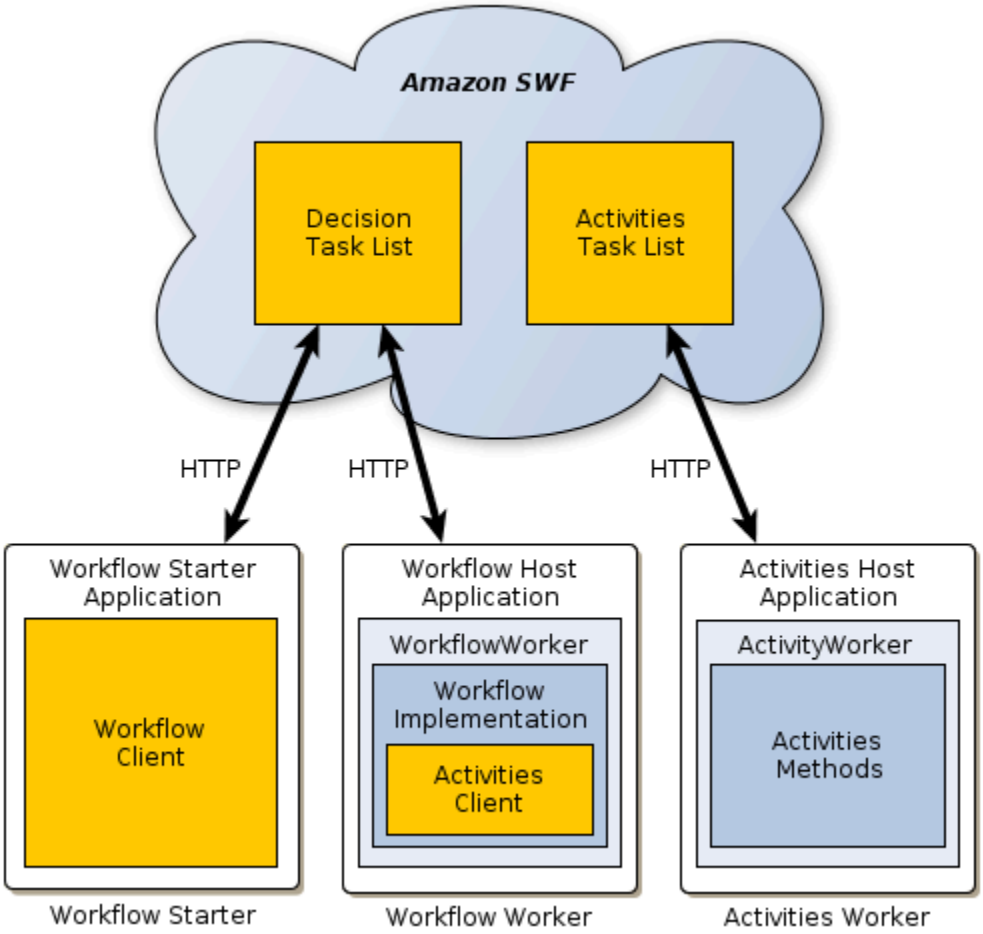
Themen

- AWS Flow Framework Grundbegriffe: Anwendungsstruktur
- AWS Flow Framework Grundkonzepte: Zuverlässige Ausführung
- AWS Flow Framework Grundbegriffe: Verteilte Ausführung
- AWS Flow Framework Grundbegriffe: Aufgabenlisten und Aufgabenausführung
- AWS Flow Framework Grundkonzepte: Skalierbare Anwendungen
- AWS Flow Framework Grundbegriffe: Data Exchange zwischen Aktivitäten und Workflows
- AWS Flow Framework Grundbegriffe: Data Exchange zwischen Anwendungen und Workflow-Ausführungen
- Amazon SWF-Timeout-Typen

AWS Flow Framework Grundbegriffe: Anwendungsstruktur

Konzeptionell besteht eine AWS Flow Framework Anwendung aus drei grundlegenden Komponenten: Workflow-Startern, Workflow-Workern und Activity-Workern. Eine oder mehrere Hostanwendungen sind dafür verantwortlich, die Worker (Workflow und Aktivität) bei Amazon SWF zu registrieren, die Worker zu starten und die Bereinigung durchzuführen. Die Worker setzen die Mechaniken der Workflow-Ausführung um und können auf verschiedenen Hosts implementiert werden.

Dieses Diagramm stellt eine grundlegende AWS Flow Framework Anwendung dar:



> **Note**
>
> Die Implementierung dieser Komponenten in drei getrennten Anwendungen ist konzeptionell praktisch. Sie können jedoch Anwendungen erstellen, um diese Funktionalität auf verschiedene Weise zu implementieren. Es ist zum Beispiel möglich, eine einzelne Host-Anwendung für die Aktivitäts- und Workflow-Worker oder getrennte Aktivitäts- und Workflow-Hosts zu verwenden. Sie können auch mehrere Aktivitäts-Worker jeweils eine unterschiedliche Reihe von Aktivitäten auf getrennten Hosts ausführen lassen und Ähnliches.

Die drei AWS Flow Framework Komponenten interagieren indirekt, indem sie HTTP-Anfragen an Amazon SWF senden, das die Anfragen verwaltet. Amazon SWF macht Folgendes:

- Er verwaltet eine oder mehrere Entscheidungsaufgabenlisten, mit denen die nächsten Schritte festgelegt werden, die ein Workflow-Worker ausführen soll.

- Er verwaltet eine oder mehrere Aktivitätsaufgabenlisten, mit denen die nächsten Aufgaben festgelegt werden, die ein Aktivitäts-Worker ausführen soll.
- Verwaltet einen detaillierten step-by-step Verlauf der Ausführung des Workflows.

Mit dem AWS Flow Framework muss sich Ihr Anwendungscode nicht direkt mit vielen der in der Abbildung gezeigten Details befassen, z. B. dem Senden von HTTP-Anfragen an Amazon SWF. Sie rufen einfach AWS Flow Framework Methoden auf und das Framework kümmert sich im Hintergrund um die Details.

Rolle des Aktivitäts-Workers

Der Aktivitäts-Worker führt die verschiedenen Aufgaben durch, die der Workflow bewerkstelligen muss. Er besteht aus Folgendem:

- Der Aktivitätsimplementierung. Diese enthält eine Reihe von Aktivitätsmethoden, die bestimmte Aufgaben für den Workflow ausführen.
- Ein ActivityWorkerObjekt, das lange HTTP-Abfrageanfragen verwendet, um Amazon SWF nach auszuführenden Aktivitätsaufgaben abzufragen. Wenn eine Aufgabe benötigt wird, beantwortet Amazon SWF die Anfrage, indem es die für die Ausführung der Aufgabe erforderlichen Informationen sendet. Das ActivityWorkerObjekt ruft dann die entsprechende Aktivitätsmethode auf und gibt die Ergebnisse an Amazon SWF zurück.

Rolle des Workflow-Workers

Der Workflow-Workers orchestriert die Ausführung der verschiedenen Aktivitäten, verwaltet den Datenfluss und verarbeitet fehlgeschlagene Aktivitäten. Er besteht aus Folgendem:

- Der Workflow-Implementierung. Diese enthält die Logik zur Aktivitätsorchestrierung, verarbeitet fehlgeschlagene Aktivitäten und so weiter.
- Einem Aktivitäts-Client. Dieser fungiert als Proxy für den Aktivitäts-Worker und ermöglicht dem Workflow-Worker, eine asynchrone Ausführung von Aktivitäten zu planen.
- Ein WorkflowWorkerObjekt, das lange HTTP-Abfrageanfragen verwendet, um Amazon SWF nach Entscheidungsaufgaben abzufragen. Wenn die Workflow-Aufgabenliste Aufgaben enthält, beantwortet Amazon SWF die Anfrage, indem es die Informationen zurücksendet, die für die Ausführung der Aufgabe erforderlich sind. Das Framework führt dann den Workflow zur Ausführung der Aufgabe aus und gibt die Ergebnisse an Amazon SWF zurück.

Rolle des Workflow-Starters

Der Workflow-Starter startet eine Workflow-Instance, auch Workflow-Ausführung genannt. Er kann während der Ausführung mit einer Instance interagieren, um zusätzliche Daten an den Workflow-Worker zu übergeben oder den aktuellen Workflow-Status abzufragen.

Der Workflow-Starter startet die Workflow-Ausführung mithilfe eines Workflow-Clients. Er interagiert mit dem Workflow nach Bedarf während der Workflow-Ausführung und führt die Bereinigung durch. Der Workflow-Starter könnte eine lokal ausgeführte Anwendung, eine Webanwendung AWS CLI oder sogar die sein. AWS Management Console

So interagiert Amazon SWF mit Ihrer Anwendung

Amazon SWF vermittelt die Interaktion zwischen den Workflow-Komponenten und führt einen detaillierten Workflow-Verlauf. Amazon SWF initiiert keine Kommunikation mit den Komponenten. Es wartet auf HTTP-Anfragen von den Komponenten und verwaltet die Anfragen nach Bedarf. Zum Beispiel:

- Wenn die Anfrage von einem Mitarbeiter stammt, der nach verfügbaren Aufgaben fragt, antwortet Amazon SWF dem Mitarbeiter direkt, ob eine Aufgabe verfügbar ist. Weitere Informationen zur Funktionsweise von Abfragen finden Sie unter Abfragen von Aufgaben im Amazon Simple Workflow Service – Entwicklerhandbuch.
- Handelt es sich bei der Anfrage um eine Benachrichtigung eines Aktivitätsarbeiters, dass eine Aufgabe abgeschlossen ist, zeichnet Amazon SWF die Informationen im Ausführungsverlauf auf und fügt der Liste der Entscheidungsaufgaben eine Aufgabe hinzu, um den Workflow-Mitarbeiter darüber zu informieren, dass die Aufgabe abgeschlossen ist, sodass er mit dem nächsten Schritt fortfahren kann.
- Wenn die Anforderung vom Workflow-Worker zur Ausführung einer Aktivität stammt, zeichnet Amazon SWF die Informationen im Ausführungsverlauf auf und fügt der Aufgabenliste der Aktivitäten eine Aufgabe hinzu, um einen Aktivitätsarbeiter anzuweisen, die entsprechende Aktivitätsmethode auszuführen.

Dieser Ansatz ermöglicht es Mitarbeitern, auf jedem System mit Internetverbindung zu arbeiten, einschließlich EC2 Amazon-Instances, Unternehmensrechenzentren, Client-Computern usw. Es muss nicht einmal dasselbe Betriebssystem ausgeführt werden. Da die HTTP-Anforderungen von den Workern stammen, sind keine extern sichtbaren Ports erforderlich. Worker können sogar hinter einer Firewall ausgeführt werden.

Weitere Informationen

Eine ausführlichere Erläuterung der Funktionsweise von Amazon SWF finden Sie im Amazon Simple Workflow Service Developer Guide.

AWS Flow Framework Grundkonzepte: Zuverlässige Ausführung

Asynchron verteilte Anwendungen müssen mit Zuverlässigkeitsproblemen umgehen, die bei herkömmlichen Anwendungen nicht auftreten, einschließlich:

- So stellen Sie eine zuverlässige Kommunikation zwischen asynchron verteilten Komponenten bereit, z. B. lang andauernde Komponenten auf Remote-Systemen.
- So stellen Sie sicher, dass Ergebnisse nicht verloren gehen, wenn eine Komponente fehlschlägt oder getrennt wird, besonders bei lang andauernden Anwendungen.
- So handhaben Sie fehlgeschlagene verteilte Komponenten.

Anwendungen können sich auf die SWF AWS Flow Framework und Amazon SWF verlassen, um diese Probleme zu lösen. Wir werden untersuchen, wie Amazon SWF Mechanismen bereitstellt, die sicherstellen, dass Ihre Workflows zuverlässig und vorhersehbar funktionieren, auch wenn sie lange dauern und von asynchronen Aufgaben abhängen, die rechnerisch und mit menschlicher Interaktion ausgeführt werden.

Bereitstellen von zuverlässiger Kommunikation

AWS Flow Framework ermöglicht eine zuverlässige Kommunikation zwischen einem Workflow-Worker und seinen Activity-Workern, indem Amazon SWF verwendet wird, um Aufgaben an Mitarbeiter mit verteilten Aktivitäten zu verteilen und die Ergebnisse an den Workflow-Worker zurückzugeben. Amazon SWF verwendet die folgenden Methoden, um eine zuverlässige Kommunikation zwischen einem Mitarbeiter und seinen Aktivitäten sicherzustellen:

- Amazon SWF speichert geplante Aktivitäten und Workflow-Aufgaben dauerhaft und garantiert, dass sie höchstens einmal ausgeführt werden.
- Amazon SWF garantiert, dass eine Aktivitätsaufgabe entweder erfolgreich abgeschlossen wird und ein gültiges Ergebnis zurückgibt, oder dass der Workflow-Worker darüber informiert wird, dass die Aufgabe fehlgeschlagen ist.
- Amazon SWF speichert dauerhaft das Ergebnis jeder abgeschlossenen Aktivität oder, bei fehlgeschlagenen Aktivitäten, relevante Fehlerinformationen.

AWS Flow Framework Anschließend bestimmt der anhand der Aktivitätsergebnisse von Amazon SWF, wie mit der Ausführung des Workflows fortgefahren werden soll.

Sicherstellen, dass Ergebnisse nicht verloren gegangen sind

Beibehalten des Workflow-Verlaufs

Eine Aktivität, die eine Data Mining-Operation für ein Petabyte an Daten durchführt, kann Stunden dauern und eine Aktivität, die einen menschlichen Worker anweist, eine komplexe Aufgabe durchzuführen, kann Tage oder sogar Wochen dauern!

Um solchen Szenarien Rechnung zu tragen, kann die Fertigstellung von AWS Flow Framework Workflows und Aktivitäten beliebig lange dauern: bis zu einem Jahr für die Ausführung eines Workflows. Die zuverlässige Ausführung von lange dauernden Prozessen erfordert einen Mechanismus für die dauerhafte Speicherung des Workflow-Ausführungsverlaufs auf fortschreitender Basis.

Das AWS Flow Framework handhabt dies, abhängig von Amazon SWF, das einen Laufverlauf jeder Workflow-Instanz verwaltet. Der Workflow-Verlauf stellt einen vollständigen und autoritativen Datensatz des Workflow-Fortschritts bereit, einschließlich aller Workflow- und Aktivitätsaufgaben, die geplant und abgeschlossen wurden, und den Informationen, die durch abgeschlossene oder fehlgeschlagene Aktivitäten zurückgegeben wurden.

AWS Flow Framework Anwendungen müssen normalerweise nicht direkt mit dem Workflow-Verlauf interagieren, können aber bei Bedarf darauf zugreifen. Für die meisten Zwecke können Anwendungen einfach das Framework mit dem Workflow-Verlauf im Hintergrund interagieren lassen. Eine vollständige Erläuterung des Workflow-Verlaufs finden Sie unter Workflow-Verlauf im Amazon Simple Workflow Service Developer Guide.

Zustandslose Ausführung

Der Ausführungsverlauf ermöglicht Workflow-Workern zustandslos zu sein. Wenn Sie über mehrere Instances eines Workflow- oder Aktivitäts-Worker verfügen, kann jeder Worker jede Aufgabe durchführen. Der Mitarbeiter erhält alle Statusinformationen, die er zur Ausführung der Aufgabe benötigt, von Amazon SWF.

Dieser Ansatz macht die Workflows zuverlässiger. Wenn zum Beispiel ein Aktivitäts-Worker fehlschlägt, müssen Sie den Workflow nicht neu starten. Starten Sie den Worker einfach neu und er beginnt damit, die Aufgabenliste abzufragen und eine beliebige Aufgabe auf der Liste zu

verarbeiten, unabhängig davon, wann der Fehler aufgetreten ist. Sie können Ihren gesamten Workflow fehlertolerant machen, indem Sie zwei oder mehr Workflow- und Aktivitäts-Worker verwenden, eventuell auf getrennten Systemen. Wenn dann einer der Worker fehlschlägt, fahren die anderen mit der Verarbeitung geplanter Aufgaben ohne jegliche Unterbrechung im Workflow-Fortschritt fort.

Verarbeitung fehlgeschlagener verteilter Komponenten

Aktivitäten schlagen häufig aus temporären Gründen fehl, z. B. eine kurzzeitige Verbindungstrennung, daher ist eine allgemeine Strategie für die Handhabung von fehlgeschlagenen Aktivitäten, die Aktivität zu wiederholen. Statt den Wiederholungsprozess zu behandeln, indem komplexe Strategien der Nachrichtenübergabe implementiert werden, können sich Anwendungen auf den AWS Flow Framework verlassen. Er bietet mehrere Mechanismen zum Wiederholen fehlgeschlagener Aktivitäten und stellt einen integrierten Ausnahmebehandlungsmechanismus bereit, der mit asynchronen, verteilten Ausführungen von Aufgaben in einem Workflow funktioniert.

AWS Flow Framework Grundbegriffe: Verteilte Ausführung

Eine Workflow-Instanz ist im Grunde ein virtueller Ausführungsthread, der Aktivitäten und Orchestrierungslogik umfassen kann, die auf mehreren Remotecomputern ausgeführt werden. Amazon SWF und die AWS Flow Framework Funktion als Betriebssystem, das Workflow-Instanzen auf einer virtuellen CPU wie folgt verwaltet:

- Den Ausführungsstatus der jeweiligen Instance verwalten
- Zwischen den Instances wechseln
- Fortsetzen der Ausführung einer Instance an der Stelle, an der sie herausgeschaltet wurde

Workflow-Replay

Da Aktivitäten langwierig sein können, ist eine Blockierung durch den Workflow bis zu seinem Abschluss unerwünscht. Stattdessen AWS Flow Framework verwaltet der die Workflow-Ausführung mithilfe eines Wiedergabemechanismus, der sich auf den von Amazon SWF verwalteten Workflow-Verlauf stützt, um den Workflow in Episoden auszuführen.

Jeder Abschnitt wiederholt die Workflow-Logik so, dass jede Aktivität nur einmal ausgeführt wird. Daher ist sichergestellt, dass Aktivitäten und asynchrone Methoden erst ausgeführt werden, wenn ihre Promise-Objekte bereit sind.

Der Workflow-Starter startet den ersten Replay-Abschnitt, sobald er die Workflow-Ausführung startet. Das Framework ruft die Einstiegspunktmethode des Workflows auf. Dann geht es folgendermaßen vor:

1. Es führt alle Workflow-Aufgaben aus, die nicht vom Abschluss einer Aktivität abhängen, einschließlich des Aufrufs aller Aktivitäts-Client-Methoden.
2. Gibt Amazon SWF eine Liste von Aktivitäten und Aufgaben, deren Ausführung geplant werden soll. Für den ersten Abschnitt besteht diese Liste nur aus den Aktivitäten, die nicht von einem Promise-Objekt abhängig sind und sofort ausgeführt werden können.
3. Benachrichtigt Amazon SWF, dass die Episode abgeschlossen ist.

Amazon SWF speichert die Aktivitätsaufgaben im Workflow-Verlauf und plant ihre Ausführung, indem sie in die Aktivitätsaufgabenliste aufgenommen werden. Die Aktivitäts-Worker rufen die Aufgabenliste ab und führen die Aufgaben aus.

Wenn ein Activity Worker eine Aufgabe abschließt, gibt er das Ergebnis an Amazon SWF zurück. Amazon SWF zeichnet es im Workflow-Ausführungsverlauf auf und plant eine neue Workflow-Aufgabe für den Workflow-Worker, indem es sie in die Workflow-Aufgabenliste aufnimmt. Der Workflow-Worker fragt die Aufgabenliste ab. Wenn er die Aufgabe erhält, führt er den nächsten Replay-Abschnitt wie folgt aus:

1. Das Framework führt die Einstiegspunktmethode des Workflows aus. Dann geht es folgendermaßen vor:
   - Es führt alle Workflow-Aufgaben aus, die nicht vom Abschluss einer Aktivität abhängen, einschließlich des Aufrufs aller Aktivitäts-Client-Methoden. Das Framework überprüft jedoch den Ausführungsverlauf und plant keine doppelten Aktivitätsaufgaben.
   - Es prüft den Verlauf, um zu ermitteln, welche Aktivitätsaufgaben abgeschlossen wurden. Dann führt es alle asynchronen Workflow-Methoden aus, die von diesen Aktivitäten abhängen.
2. Wenn alle Workflow-Aufgaben, die ausgeführt werden können, abgeschlossen sind, meldet das Framework zurück an Amazon SWF:
   - Es gibt Amazon SWF eine Liste aller Aktivitäten, deren `Promise<T>` Eingabeobjekte seit der letzten Episode fertig geworden sind und deren Ausführung geplant werden kann.
   - Wenn die Episode keine zusätzlichen Aktivitätsaufgaben generiert hat, es aber immer noch nicht abgeschlossene Aktivitäten gibt, benachrichtigt das Framework Amazon SWF, dass die Episode abgeschlossen ist. Es wartet dann auf eine andere Aktivität, um den nächsten Replay-Abschnitt zu starten.

- Wenn die Episode keine zusätzlichen Aktivitätsaufgaben generiert hat und alle Aktivitäten abgeschlossen wurden, benachrichtigt das Framework Amazon SWF, dass die Workflow-Ausführung abgeschlossen ist.

Beispiele zum Replay-Verhalten finden Sie unter AWS Flow Framework für Java Replay Behavior.

Replay und asynchrone Workflow-Methoden

Asynchrone Workflow-Methoden werden oft ähnlich wie Aktivitäten verwendet, denn die Methode verzögert die Ausführung, bis alle übergebenen `Promise<T>`-Objekte bereit sind. Der Replay-Mechanismus behandelt asynchrone Methoden jedoch anders als dies bei Aktivitäten der Fall ist.

- Das Replay garantiert nicht, dass eine asynchrone Methode nur einmal ausgeführt wird. Es verzögert die Ausführung einer asynchronen Methode nur so lange, bis die ihr übergeben Promise-Objekte bereit sind. Dann führt er sie für alle folgenden Abschnitte aus.
- Wenn eine asynchrone Methode abgeschlossen ist, startet sie keinen neuen Abschnitt.

Ein Beispiel für das Replay eines asynchronen Workflows finden Sie in AWS Flow Framework für Java Replay Behavior.

Replay und die Workflow-Implementierung

In den meisten Fällen müssen Sie sich nicht um die Einzelheiten des Replay-Mechanismus kümmern. Er arbeitet im Grunde hinter den Kulissen. Das Replay hat jedoch zwei wichtige Auswirkungen auf Ihre Workflow-Implementierung.

- Verwenden Sie keine Workflow-Methoden, um langlaufende Aufgaben auszuführen, da das Replay die entsprechende Aufgabe mehrfach wiederholt. Auch asynchrone Workflow-Methoden werden typischerweise mehr als einmal ausgeführt. Verwenden Sie stattdessen für langlaufende Aufgaben Aktivitäten. Dann führt das Replay die Aktivitäten nur einmal aus.
- Ihre Workflow-Logik muss vollständig deterministisch sein. Jeder Abschnitt muss dem gleichen Steuerungsfluss folgen. Beispielsweise sollte der Steuerungsfluss nicht von der aktuellen Zeit abhängen. Eine detaillierte Beschreibung des Replays und der deterministischen Anforderungen finden Sie unter Nichtdeterminismus.

AWS Flow Framework Grundbegriffe: Aufgabenlisten und Aufgabenausführung

Amazon SWF verwaltet Workflow- und Aktivitätsaufgaben, indem es sie in benannten Listen veröffentlicht. Amazon SWF verwaltet mindestens zwei Aufgabenlisten, eine für Workflow-Worker und eine für Activity Worker.

> **Note**
>
> Sie können beliebig viele Aufgabenlisten angeben, wobei jeder Liste unterschiedliche Worker zugeordnet sind. Die Anzahl der Aufgabenlisten ist unbegrenzt. In der Regel geben Sie die Aufgabenliste eines Workers in der Worker-Host-Anwendung an, sobald Sie das Worker-Objekt erstellen.

Der folgende Auszug aus der `HelloWorldWorkflow`-Host-Anwendung legt einen neuen Aktivitäts-Worker an und ordnet ihn der Aktivitätsaufgabenliste `HelloWorldList` zu.

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

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

Standardmäßig plant Amazon SWF die Aufgaben des Mitarbeiters auf der `HelloWorldList` Liste. Dann fragt der Worker diese Liste nach Aufgaben ab. Sie können einer Aufgabenliste einen beliebigen Namen zuweisen. Sie können sogar den gleichen Namen für Workflow- und Aktivitätslisten verwenden. Intern platziert Amazon SWF die Namen der Arbeitsablauf- und Aktivitätsaufgabenlisten in unterschiedlichen Namespaces, sodass die beiden Listen unterschiedlich sind.

Wenn Sie keine Aufgabenliste angeben, AWS Flow Framework gibt die eine Standardliste an, wenn der Worker den Typ bei Amazon SWF registriert. Weitere Informationen finden Sie unter Registrierung von Workflow- und Aktivitätstypen.

Manchmal ist es sinnvoll, einen bestimmten Worker oder eine bestimmte Gruppe von Workern bestimmte Aufgaben ausführen zu lassen. Beispielsweise kann ein Bildverarbeitungs-Workflow eine Aktivität verwenden, um ein Bild herunterzuladen und eine andere Aktivität, um das Bild zu bearbeiten. Es ist effizienter, beide Aufgaben auf dem gleichen System auszuführen und den Overhead der Übertragung großer Dateien über das Netzwerk zu vermeiden.

Um solche Szenarien zu unterstützen, können Sie beim Aufruf einer Activity-Client-Methode explizit eine Aufgabenliste angeben, indem Sie einen Overload mit einem `schedulingOptions`-Parameter verwenden. Sie geben die Aufgabenliste an, indem Sie der Methode ein entsprechend konfiguriertes `ActivitySchedulingOptions` Objekt übergeben.

Nehmen wir beispielsweise an, die `say`-Aktivität der `HelloWorldWorkflow`-Anwendung wird von einem anderen Aktivitäts-Worker als `getName` und `getGreeting` gehostet. Das folgende Beispiel zeigt, wie Sie sicherstellen können, dass `say` dieselbe Aufgabenliste wie `getName` und `getGreeting` verwendet (auch wenn sie ursprünglich verschiedenen Listen zugeordnet waren).

```
public class GreeterWorkflowImpl implements GreeterWorkflow {
  private GreeterActivitiesClient operations1 = new GreeterActivitiesClientImpl1(); //
getGreeting and getName
  private GreeterActivitiesClient operations2 = new GreeterActivitiesClientImpl2(); //
say
  @Override
  public void greet() {
    Promise<String> name = operations1.getName();
    Promise<String> greeting = operations1.getGreeting(name);
    runSay(greeting);
  }
  @Asynchronous
  private void runSay(Promise<String> greeting){
    String taskList = operations1.getSchedulingOptions().getTaskList();
    ActivitySchedulingOptions schedulingOptions = new ActivitySchedulingOptions();
    schedulingOptions.setTaskList(taskList);
    operations2.say(greeting, schedulingOptions);
  }
}
```

Die asynchrone `runSay`-Methode ruft die `getGreeting`-Aufgabenliste aus seinem Client-Objekt ab. Dann erstellt und konfiguriert es ein `ActivitySchedulingOptions`-Objekt. Dieses stellt sicher, dass `say` dieselbe Aufgabenliste wie `getGreeting` abfragt.

> **Note**
>
> Wenn Sie einen `schedulingOptions`-Parameter an eine Activity-Client-Methode übergeben, überschreibt dieser den ursprünglichen Aufgabenplan nur für diese Aktivitätsausführung. Wenn Sie die Client-Methode für Aktivitäten erneut aufrufen, ohne eine Aufgabenliste anzugeben, weist Amazon SWF die Aufgabe der ursprünglichen Liste zu, und der Activity Worker fragt diese Liste ab.

AWS Flow Framework Grundkonzepte: Skalierbare Anwendungen

Amazon SWF verfügt über zwei Hauptfunktionen, die es einfach machen, eine Workflow-Anwendung so zu skalieren, dass sie die aktuelle Last bewältigen kann:

- Ein vollständiger Ausführungsverlauf des Workflows ermöglicht die Implementierung einer zustandslosen Anwendung.
- Eine lose an die Aufgabenausführung gekoppelte Aufgabenplanung vereinfacht das Skalieren der Anwendung den aktuellen Anforderungen entsprechend.

Amazon SWF plant Aufgaben, indem es sie in dynamisch zugewiesenen Aufgabenlisten veröffentlicht und nicht direkt mit den Workflow- und Aktivitätsmitarbeitern kommuniziert. Worker fragen ihre jeweiligen Listen für Aufgaben stattdessen über HTTP-Anforderungen ab. Dieser Ansatz verbindet die Aufgabenplanung lose mit der Aufgabenausführung und ermöglicht es den Mitarbeitern, auf jedem geeigneten System zu arbeiten, einschließlich EC2 Amazon-Instances, Unternehmensrechenzentren, Client-Computern usw. Da die HTTP-Anfragen von den Workern stammen, sind keine extern sichtbaren Ports erforderlich, sodass die Mitarbeiter sogar hinter einer Firewall laufen können.

Der Langabfragemechanismus, mit dem Worker Aufgaben abfragen, verhindert eine Überlastung der Worker. Selbst wenn Spitzen bei den geplanten Aufgaben auftreten, rufen Worker Aufgaben nach ihrem eigenen Rhythmus ab. Da Worker jedoch zustandslos sind, können Sie eine Anwendung bei zunehmender Last dynamisch skalieren, indem Sie zusätzliche Worker-Instances starten. Selbst wenn diese auf verschiedenen Systemen ausgeführt werden, ruft jede Instance dieselbe Aufgabenliste ab und der erste verfügbare Worker führt die Aufgabe aus. Dabei spielt es keine Rolle,

wo sich der Worker befindet oder wann er gestartet wird. Bei abnehmender Last können Sie die Anzahl der Worker wieder entsprechend reduzieren.

AWS Flow Framework Grundbegriffe: Data Exchange zwischen Aktivitäten und Workflows

Wenn Sie eine asynchrone Aktivitäts-Client-Methode aufrufen, gibt sie sofort ein Promise-Objekt (auch als Future-Objekt bekannt) zurück, das den Rückgabewert der Aktivitätsmethode darstellt. Das Promise-Objekt weist zunächst einen nicht bereiten Zustand auf und der Rückgabewert ist undefiniert. Nachdem die Aktivitätsmethode ihre Aufgabe abgeschlossen hat und zurückgibt, marshallt das Framework den Rückgabewert über das Netzwerk zum Workflow-Worker, der dem Promise-Objekt einen Wert zuweist und das Objekt in einen betriebsbereiten Zustand versetzt.

Selbst wenn eine Aktivitätsmethode keinen Rückgabewert hat, können Sie das Promise-Objekt dennoch für das Verwalten der Workflow-Ausführung verwenden. Wenn Sie ein zurückgegebenes Promise-Objekt an eine Aktivitäts-Client-Methode oder eine asynchrone Workflow-Methode übergeben, schiebt es die Ausführung auf, bis das Objekt bereit ist.

Wenn Sie ein oder mehrere Promise-Objekte an eine Aktivitäts-Client-Methode übergeben, fügt das Framework die Aufgabe in die Warteschlange ein, schiebt sie aber auf, bis alle Objekte bereit sind. Es extrahiert dann die Daten aus jedem Promise-Objekt und marshallt sie über das Internet zu dem Aktivitäts-Worker, der sie dann an die Aktivitätsmethode als Standardtyp übergibt.

> **Note**
>
> Wenn Sie große Mengen an Daten zwischen Workflow- und Aktivitäts-Workern übermitteln müssen, besteht der bevorzugte Ansatz darin, die Daten an einem passenden Speicherort zu speichern und nur die Abrufinformationen zu übergeben. Sie können die Daten beispielsweise in einem Amazon S3 S3-Bucket speichern und die zugehörige URL übergeben.

Die Promise <T> Type

Der `Promise<T>`-Typ ist in mancherlei Hinsicht mit dem Java-Typ `Future<T>` vergleichbar. Beide Typen stellen Werte dar, die von asynchronen Methoden zurückgegeben werden und ursprünglich undefiniert sind. Sie können auf den Wert eines Objekts zugreifen, indem Sie seine `get`-Methode aufrufen. Darüber hinaus verhalten sich die beiden Typen auf eher unterschiedliche Art und Weise.

- `Future<T>` ist ein Synchronisierungskonstrukt, das einer Anwendung ermöglicht, auf die Beendigung einer asynchronen Methode zu warten. Wenn Sie `get` aufrufen und das Objekt nicht bereit ist, blockiert es, bis das Objekt bereit ist.
- Mit `Promise<T>` wird die Synchronisierung vom Framework verarbeitet. Wenn Sie `get` aufrufen und das Objekt nicht bereit ist, löst `get` eine Ausnahme aus.

Der Hauptzweck von `Promise<T>` besteht darin, den Datenfluss von einer Aktivität zu einer anderen zu verwalten. Es stellt sicher, dass eine Aktivität erst dann ausgeführt wird, wenn die Eingabedaten gültig sind. In vielen Fällen müssen Workflow-Worker nicht direkt auf `Promise<T>`-Objekte zugreifen. Sie übergeben die Objekte einfach von einer Aktivität an eine andere und lassen das Framework und die Aktivitäts-Worker die Details handhaben. Um auf den Wert eines `Promise<T>`-Objekts in einem Workflow-Worker zuzugreifen, müssen Sie sicher sein, dass das Objekt bereit ist, bevor Sie seine `get`-Methode aufrufen.

- Der bevorzugte Ansatz besteht darin, das `Promise<T>`-Objekt an eine asynchrone Workflow-Methode zu übergeben und die Werte dort zu bearbeiten. Eine asynchrone Methode schiebt die Ausführung auf, bis all seine `Promise<T>`-Eingabeobjekte bereit sind, was garantiert, dass Sie sicher auf ihre Werte zugreifen können.
- `Promise<T>` macht eine `isReady`-Methode verfügbar, die `true` zurückgibt, wenn das Objekt bereit ist. Die Verwendung von `isReady` zum Abfragen eines `Promise<T>`-Objekts wird nicht empfohlen, `isReady` ist jedoch unter bestimmten Umständen hilfreich.

Der AWS Flow Framework für Java enthält auch einen `Settable<T>` Typ, der von diesem abgeleitet ist `Promise<T>` und ein ähnliches Verhalten aufweist. Der Unterschied besteht darin, dass das Framework normalerweise den Wert eines `Promise<T>` Objekts festlegt und der Workflow-Worker für die Festlegung des Werts von a verantwortlich ist`Settable<T>`.

Es gibt einige Situationen, in denen ein Workflow-Worker ein `Promise<T>`-Objekt erstellen und seinen Wert festlegen muss. So muss etwa eine asynchrone Methode, die ein `Promise<T>`-Objekt zurückgibt, einen Rückgabewert erstellen.

- Um ein Objekt zu erstellen, das einen typisierten Wert darstellt, rufen Sie die statische `Promise.asPromise`-Methode auf, die ein `Promise<T>`-Objekt des entsprechenden Typs erstellt, seinen Wert festlegt und es in den betriebsbereiten Zustand versetzt.
- Zum Erstellen eines `Promise<Void>`-Objekts rufen Sie die statische `Promise.Void`-Methode auf.

> **Note**
>
> `Promise<T>` kann einen beliebigen gültigen Typ darstellen. Wenn die Daten jedoch über das Internet gemarshallt werden müssen, muss der Typ mit dem Datenkonverter kompatibel sein. Details finden Sie im nächsten Abschnitt.

Datenkonverter und Marshaling

Der AWS Flow Framework leitet Daten mithilfe eines Datenkonverters über das Internet weiter. Standardmäßig verwendet das Framework einen Datenkonverter, der auf dem Jackson JSON-Prozessor basiert. Dieser Konverter weist jedoch einige Einschränkungen auf. Er kann beispielsweise Zuordnungen, die keine Zeichenfolgen als Schlüssel verwenden, nicht marshallen. Wenn der Standardkonverter für Ihre Anwendung nicht ausreichend ist, können Sie einen benutzerdefinierten Datenkonverter implementieren. Details hierzu finden Sie unter DataConverters.

AWS Flow Framework Grundbegriffe: Data Exchange zwischen Anwendungen und Workflow-Ausführungen

Eine Workflow-Eintrittspunktmethode kann über einen oder mehrere Parameter verfügen, die es dem Workflow-Starter ermöglichen, erste Daten an den Workflow zu übergeben. Sie kann außerdem dazu dienen, dem Workflow während der Ausführung zusätzliche Daten zur Verfügung zu stellen. Wenn beispielsweise ein Kunde seine Versandadresse ändert, können Sie den Workflow zur Bestellverarbeitung benachrichtigen, der dann die entsprechenden Änderungen vornimmt.

Amazon SWF ermöglicht es Workflows, eine Signalmethode zu implementieren, die es Anwendungen wie dem Workflow-Starter ermöglicht, jederzeit Daten an den Workflow zu übergeben. Eine Signal-Methode kann jeden beliebigen Namen und beliebige Parameter haben. Sie legen sie als Signal-Methode fest, indem Sie sie in Ihre Workflow-Schnittstellendefinition einbeziehen und eine `@Signal`-Annotation auf die Methodendeklaration anwenden.

Das folgende Beispiel zeigt eine Workflow-Schnittstelle zur Bestellverarbeitung, die die Signal-Methode `changeOrder` deklariert, durch die der Workflow-Starter die Originalbestellung ändern kann, nachdem der Workflow gestartet wurde.

```
@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 300)
```

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

Die Annotationsverarbeitung des Frameworks erzeugt eine Workflow-Client-Methode mit demselben Namen wie die Signal-Methode und der Workflow-Starter ruft die Client-Methode auf, um Daten an den Workflow zu übergeben. Ein Beispiel finden Sie unter AWS Flow Framework Rezepte

Amazon SWF-Timeout-Typen

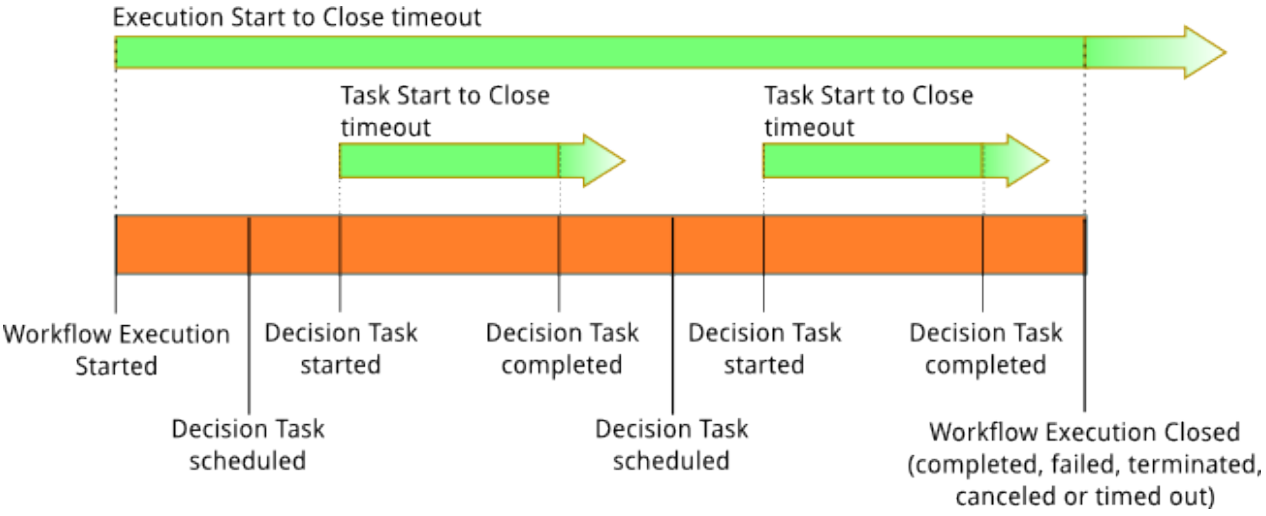
Um sicherzustellen, dass Workflow-Ausführungen korrekt ausgeführt werden, können Sie mit Amazon SWF verschiedene Arten von Timeouts festlegen. Einige Zeitüberschreitungen legen fest, wie lange der Workflow insgesamt ausgeführt werden kann. Andere Zeitüberschreitungen legen fest, wie lange es dauern darf, bis Aktivitätsaufgaben einem Worker zugewiesen werden, und wie lange die Ausführung einer Aufgabe ab der Planung dauern darf. Alle Timeouts in der Amazon SWF SWF-API sind in Sekunden angegeben. Amazon SWF unterstützt die Zeichenfolge auch `NONE` als Timeout-Wert, was bedeutet, dass es kein Timeout gibt.

Für Zeitüberschreitungen im Zusammenhang mit Entscheidungs- und Aktivitätsaufgaben fügt Amazon SWF dem Workflow-Ausführungsverlauf ein Ereignis hinzu. Die Attribute des Ereignisses geben Auskunft darüber, welche Art von Timeout eingetreten ist und welche Entscheidungs- oder Aktivitätsaufgabe betroffen war. Amazon SWF plant auch eine Entscheidungsaufgabe. Wenn der Entscheider die neue Entscheidungsaufgabe erhält, sieht er das Timeout-Ereignis in der Historie und ergreift die entsprechende Aktion, indem er die RespondDecisionTaskCompletedAktion aufruft.

Eine Aufgabe gilt vom Zeitpunkt der Planung bis zum Schließen der Aufgabe als offen. Für Aufgaben, die gerade von einem Worker verarbeitet werden, wird daher der Status "offen" gesendet. Eine Aufgabe ist geschlossen, wenn ein Worker sie als abgeschlossen, abgebrochen oder fehlgeschlagen meldet. Eine Aufgabe kann auch von Amazon SWF aufgrund eines Timeouts geschlossen werden.

Zeitüberschreitungen in Workflow- und Entscheidungsaufgaben

Die folgende Abbildung zeigt, wie Zeitüberschreitungen für Workflow- und Entscheidungsaufgaben sich auf die Lebensdauer eines Workflows auswirken:



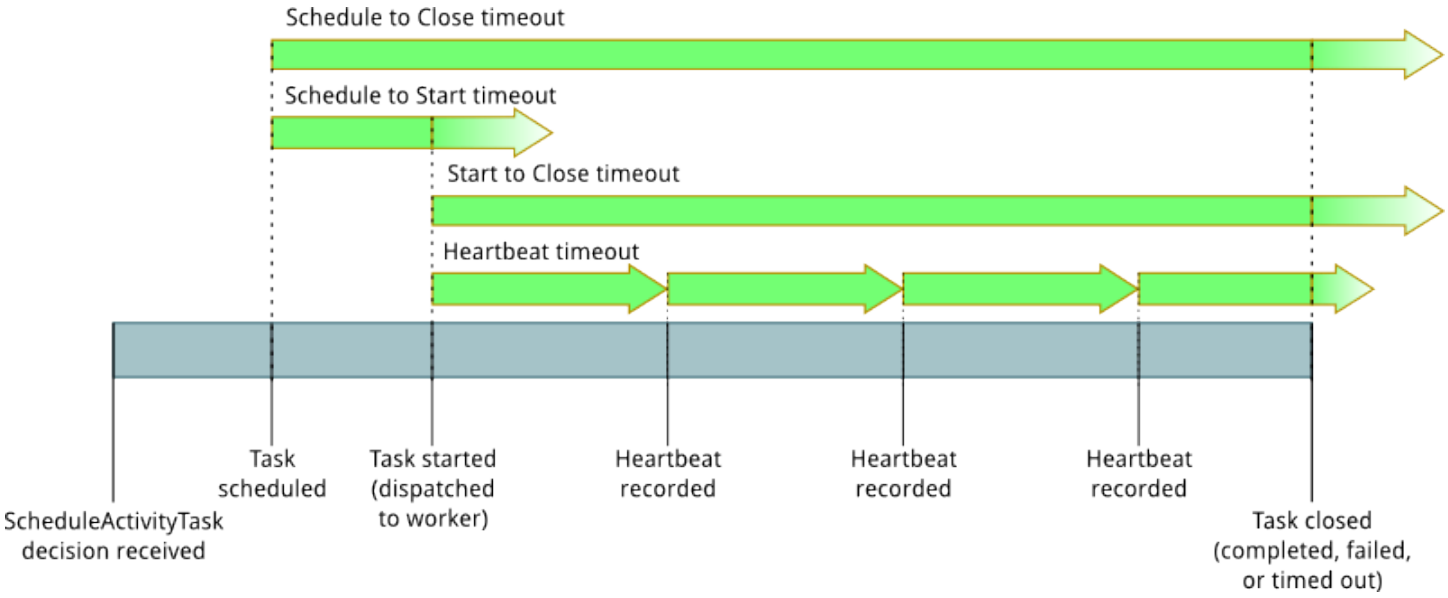
Für Workflow- und Entscheidungsaufgaben gibt es zwei relevante Zeitüberschreitungstypen:

- Workflow-Start to Close (**`timeoutType: START_TO_CLOSE`**) — Dieses Timeout gibt die maximale Zeit an, die bis zum Abschluss einer Workflow-Ausführung in Anspruch nehmen kann. Bei der Registrierung eines Workflows wird ein Standardwert festgelegt, der jedoch beim Starten des Workflows durch andere Werte überschrieben werden kann. Wenn dieses Timeout überschritten wird, schließt Amazon SWF die Workflow-Ausführung und fügt dem Workflow-Ausführungsverlauf ein Ereignis des Typs WorkflowExecutionTimedOuthinzu. Neben dem `timeoutType` legen die Ereignisattribute auch die `childPolicy` fest, die sich auf die Workflow-Ausführung auswirkt. Die untergeordnete Richtlinie legt fest, wie mit untergeordneten Workflow-Ausführungen verfahren wird, wenn bei der übergeordneten Workflow-Ausführung eine Zeitüberschreitung auftritt oder sie anderweitig beendet wird. Wenn in der `childPolicy` beispielsweise TERMINATE festgelegt ist, werden die untergeordneten Workflow-Ausführungen beendet. Nachdem bei einer Workflow-Ausführung eine Zeitüberschreitung aufgetreten ist, können Sie als einzige Aktionen dafür noch Sichtbarkeitsaufrufe ausführen.
- Entscheidungsaufgabe von Anfang bis Ende (**`timeoutType: START_TO_CLOSE`**) — Dieses Timeout gibt die maximale Zeit an, die der entsprechende Entscheider benötigen kann, um eine Entscheidungsaufgabe abzuschließen. Sie wird während der Registrierung des Workflow-Typs festgelegt. Wenn dieses Timeout überschritten wird, wird die Aufgabe im Workflow-Ausführungsverlauf als Timeout markiert, und Amazon SWF fügt dem Workflow-Verlauf ein Ereignis des Typs DecisionTaskTimedOuthinzu. Zu den Ereignisattributen gehören die IDs Ereignisse, die dem Zeitpunkt entsprechen, zu dem diese Entscheidungsaufgabe geplant (`scheduledEventId`) und wann sie gestartet wurde (`startedEventId`). Amazon SWF fügt nicht nur das Ereignis hinzu, sondern plant auch eine neue Entscheidungsaufgabe, um den Entscheider darüber zu informieren, dass bei dieser Entscheidungsaufgabe das Timeout überschritten wurde.

Nach einer Zeitüberschreitung schlagen Versuche, die abgelaufene Entscheidungsaufgabe mit `RespondDecisionTaskCompleted` abzuschließen, fehl.

Zeitüberschreitungen in Aktivitätsaufgaben

Die folgende Abbildung zeigt, wie Zeitüberschreitungen sich auf die Lebensdauer einer Aktivitätsaufgabe auswirken:



Für Aktivitätsaufgaben gibt es vier relevante Zeitüberschreitungstypen:

- Aktivitätsaufgabe von Anfang bis Ende (**`timeoutType: START_TO_CLOSE`**) — Dieses Timeout gibt die maximale Zeit an, die ein Mitarbeiter für die Bearbeitung einer Aufgabe benötigen kann, nachdem der Mitarbeiter die Aufgabe erhalten hat. Versuche, eine Aktivitätsaufgabe mit RespondActivityTaskCanceled,, und zu schließen RespondActivityTaskCompleted, schlagen RespondActivityTaskFailedfehl.
- Activity Task Heartbeat (**`timeoutType: HEARTBEAT`**) — Dieses Timeout gibt die maximale Zeit an, die eine Aufgabe ausgeführt werden kann, bevor ihr Fortschritt durch die Aktion angezeigt wird. `RecordActivityTaskHeartbeat`
- Zeitplan für den Start der Aktivitätsaufgabe (**`timeoutType: SCHEDULE_TO_START`**) — Dieses Timeout gibt an, wie lange Amazon SWF wartet, bis das Zeitlimit für die Aktivitätsaufgabe überschritten wird, wenn keine Mitarbeiter für die Ausführung der Aufgabe verfügbar sind. Nach der Zeitüberschreitung wird die abgelaufene Aufgabe keinem anderen Worker zugewiesen.
- Zeitplan für das Schließen der Aktivitätsaufgabe (**`timeoutType: SCHEDULE_TO_CLOSE`**) — Dieser Timeout gibt an, wie lange die Aufgabe von der geplanten bis zur Fertigstellung dauern

kann. Es hat sich bewährt, dass dieser Wert nicht größer als die Summe aus Task-Timeout und schedule-to-start Task-Timeout sein sollte. start-to-close

> **Note**
>
> Jeder Zeitüberschreitungstyp verfügt übe einen Standardwert, in der Regel `NONE` (unendlich). Die Höchstdauer für die Ausführung einer Aktivität ist jedoch auf ein Jahr beschränkt.

Die Standardwerte für diese Zeitüberschreitungen werden während der Registrierung des Aktivitätstyps festgelegt, können jedoch beim Planen der Aktivitätsaufgabe überschrieben werden. Wenn einer dieser Timeouts eintritt, fügt Amazon SWF dem Workflow-Verlauf ein Ereignis des Typs ActivityTaskTimedOuthinzu. Das Wertattribut `timeoutType` dieses Ereignisses gibt an, welche dieser Zeitüberschreitungen aufgetreten ist. Der Wert von `timeoutType` für jede Zeitüberschreitung ist in Klammern angegeben. Zu den Ereignisattributen gehören auch die IDs Ereignisse, die dem Zeitpunkt entsprechen, zu dem die Aktivitätsaufgabe geplant (`scheduledEventId`) und wann sie gestartet wurde (`startedEventId`). Zusätzlich zum Hinzufügen des Ereignisses plant Amazon SWF auch eine neue Entscheidungsaufgabe, um den Entscheider darüber zu informieren, dass das Timeout eingetreten ist.

Eine Aufgabe in AWS Flow Framework für Java verstehen

Themen

- Aufgabe
- Reihenfolge der Ausführung
- Workflow-Ausführung
- Nichtdeterminismus

Aufgabe

Das zugrunde liegende Primitiv, das AWS Flow Framework for Java verwendet, um die Ausführung von asynchronem Code zu verwalten, ist die `Task` Klasse. Ein Objekt vom Typ `Task` stellt Arbeit dar, die asynchron durchgeführt werden muss. Wenn Sie eine asynchrone Methode aufrufen, erzeugt das Framework eine `Task`, um den Code in dieser Methode auszuführen, und setzt ihn in eine Liste für eine Ausführung zu einem späteren Zeitpunkt. Ebenso wird beim Aufruf einer `Activity` eine `Task` dafür erstellt. Der Methodenaufruf wird danach zurückgegeben, in der Regel mit einem `Promise<T>` als zukünftiges Ergebnis des Aufrufs.

Die `Task`-Klasse ist öffentlich und kann direkt verwendet werden. Wir können beispielsweise das Beispiel „Hello World“ so neu schreiben, dass es eine `Task` anstatt einer asynchronen Methode verwendet.

```
@Override
public void startHelloWorld(){
       final Promise<String> greeting = client.getName();
        new Task(greeting) {
        @Override
        protected void doExecute() throws Throwable {
         client.printGreeting("Hello " + greeting.get() +"!");
        }
    };
}
```

Das Framework ruft die `doExecute()`-Methode auf, wenn alle `Promises`, die an den Konstruktor der `Task` übergeben wurden, einsatzbereit sind. Weitere Informationen zur `Task` Klasse finden Sie in der AWS SDK für Java Dokumentation.

Das Framework umfasst auch eine Klasse mit der Bezeichnung `Functor`, die eine `Task` darstellt, die auch ein `Promise<T>` ist. Das `Functor`-Objekt ist einsatzbereit, wenn die `Task` abgeschlossen wird. Im folgenden Beispiel wird ein `Functor` erstellt, um die Begrüßungsnachricht abzurufen:

```
Promise<String> greeting = new Functor<String>() {
    @Override
    protected Promise<String> doExecute() throws Throwable {
        return client.getGreeting();
    }
};
client.printGreeting(greeting);
```

Reihenfolge der Ausführung

Aufgaben werden nur für die Ausführung berechtigt, wenn alle `Promise<T>`-typisierten Parameter, die an die entsprechende asynchrone Methode oder Aktivität übergeben wurden, einsatzbereit sind. Eine zur Ausführung bereite `Task` wird logisch eine eine einsatzbereite Warteschlange verschoben. Mit anderen Worten wird sie für die Ausführung geplant. Die Worker-Klasse führt die Aufgabe aus, indem sie den Code aufruft, den Sie in den Hauptteil der asynchronen Methode geschrieben haben, oder indem sie im Fall einer Aktivitätsmethode eine Aktivitätsaufgabe in Amazon Simple Workflow Service (AWS) plant.

Wenn Aufgaben ausgeführt werden und Ergebnisse erzielen, sorgen sie dafür, dass andere Aufgaben einsatzbereit werden, und die Ausführung des Programms schreitet weiter voran. Die Art und Weise, wie das Framework Aufgaben ausführt, ist wichtig, um die Reihenfolge zu verstehen, in der Ihr asynchroner Code ausgeführt wird. Code, der sequenziell in Ihrem Programm erscheint, wird möglicherweise nicht tatsächlich in dieser Reihenfolge ausgeführt.

```
Promise<String> name = getUserName();
printHelloName(name);
printHelloWorld();
System.out.println("Hello, Amazon!");

@Asynchronous
private Promise<String> getUserName(){
 return Promise.asPromise("Bob");
}
@Asynchronous
private void printHelloName(Promise<String> name){
```

```
 System.out.println("Hello, " + name.get() + "!");
}
@Asynchronous
private void printHelloWorld(){
 System.out.println("Hello, World!");
}
```

Der Code in der Auflistung oben wird wie folgt gedruckt:

```
Hello, Amazon!
Hello, World!
Hello, Bob
```

Dies ist möglicherweise nicht das, was Sie erwarten. Dies kann aber einfach erklärt werden, indem Sie durchdenken, wie die Aufgaben für die asynchronen Methoden ausgeführt wurden:

1. Der Aufruf von `getUserName` erzeugt eine `Task`. Nennen wir sie `Task1`. Weil `getUserName` es keine Parameter akzeptiert, `Task1` wird es sofort in die Bereitschaftswarteschlange gestellt.
2. Anschließend erzeugt der Aufruf von `printHelloName` eine `Task`, die auf das Ergebnis von `getUserName` warten muss. Nennen wir sie `Task2`. Weil der erforderliche Wert noch nicht bereit ist, `Task2` wird er auf die Warteliste gesetzt.
3. Dann wird eine Aufgabe für `printHelloWorld` erstellt und zur einsatzbereiten Warteschlange hinzugefügt. Nennen wir sie `Task3`.
4. `println`In der Erklärung wird dann „Hello, Amazon!“ gedruckt zur Konsole.
5. An diesem Punkt befinden sich `Task1` und `Task3` in der einsatzbereiten Warteschlange und `Task2` befindet sich in der Warteliste.
6. Der Worker führt `Task1` aus. Durch das Ergebnis wird `Task2` vorbereitet. `Task2` wird der Bereitschaftswarteschlange hinter `Task3` hinzugefügt.
7. `Task3` und `Task2` werden dann in dieser Reihenfolge ausgeführt.

Die Ausführung der Aktivitäten folgt dem gleichen Muster. Wenn Sie eine Methode auf dem Aktivitätsclient aufrufen, wird eine erstellt, `Task` die bei der Ausführung eine Aktivität in Amazon SWF plant.

Das Framework nutzt Funktionen, wie Code-Generierung und dynamische Proxys, um die Logik für die Konvertierung von Methodenaufrufen in Aktivitätsaufrufe und asynchrone Aufgaben in Ihrem Programm einzufügen.

Workflow-Ausführung

Die Ausführung der Workflow-Implementierung wird auch von der Worker-Klasse verwaltet. Wenn Sie eine Methode auf dem Workflow-Client aufrufen, ruft sie Amazon SWF auf, um eine Workflow-Instanz zu erstellen. Die Aufgaben in Amazon SWF sollten nicht mit den Aufgaben im Framework verwechselt werden. Eine Aufgabe in Amazon SWF ist entweder eine Aktivitätsaufgabe oder eine Entscheidungsaufgabe. Die Ausführung einer Aktivitätsaufgabe ist einfach. Die Activity Worker-Klasse empfängt Aktivitätsaufgaben von Amazon SWF, ruft die entsprechende Aktivitätsmethode in Ihrer Implementierung auf und gibt das Ergebnis an Amazon SWF zurück.

Die Ausführung der Entscheidungsaufgaben ist komplexer. Der Workflow-Worker erhält Entscheidungsaufgaben von Amazon SWF. Eine Entscheidungsaufgabe ist effektiv eine Anfrage, die die Workflow-Logik fragt, was als Nächstes zu tun ist. Die erste Entscheidungsaufgabe wird für eine Workflow-Instance generiert, wenn sie über den Workflow-Client gestartet wird. Beim Empfang dieser Entscheidungsaufgabe beginnt das Framework mit der Ausführung des Codes in der Workflow-Methode, die mit `@Execute` versehen ist. Diese Methode führt die Koordinationslogik aus, die Aktivitäten plant. Wenn sich der Status der Workflow-Instanz ändert, z. B. wenn eine Aktivität abgeschlossen ist, werden weitere Entscheidungsaufgaben geplant. An diesem Punkt kann die Workflow-Logik entscheiden, eine Aktion basierend auf dem Ergebnis der Aktivität auszuführen, zum Beispiel kann sie entscheiden, eine andere Aktivität zu planen.

Das Framework blendet alle diese Details vom Entwickler aus, indem Entscheidungsaufgaben nahtlos in die Workflow-Logik übertragen werden. Aus der Sicht eines Entwicklers sieht der Code einfach wie ein reguläres Programm aus. Unter dem Deckmantel ordnet das Framework es Aufrufen von Amazon SWF und Entscheidungsaufgaben zu und verwendet dabei den von Amazon SWF verwalteten Verlauf. Wenn eine Entscheidungsaufgabe eintrifft, gibt das Framework die Programmausführung erneut wieder, wobei die Ergebnisse der bisher abgeschlossenen Aktivitäten eingefügt werden. Asynchrone Methoden und Aktivitäten, die auf diese Ergebnisse gewartet haben, werden entsperrt und die Programmausführung geht weiter.

Die Ausführung des Beispiel-Bildverarbeitungs-Workflows und des entsprechenden Verlaufs werden in der folgenden Tabelle gezeigt.

Ausführung des Thumbnail-Workflows

Workflow-Programmausführung	Von Amazon SWF verwalteter Verlauf
Anfängliche Ausführung	

Workflow-Programmausführung	Von Amazon SWF verwalteter Verlauf
1. Bereitstellungsschleife 2. getImageUrls 3. downloadImage 4. createThumbnail (Aufgabe in Warteschlange) 5. uploadImage (Aufgabe in Warteschlange) 6. <nächster Durchlauf der Schleife>	1. Workflow-Instance gestartet, id="1" 2. downloadImage geplant
Erneut abspielen	
1. Bereitstellungsschleife 2. getImageUrls 3. downloadImage image path="foo" 4. createThumbnail 5. uploadImage (Aufgabe in Warteschlange) 6. <nächster Durchlauf der Schleife>	1. Workflow-Instance gestartet, id="1" 2. downloadImage geplant 3. downloadImage abgeschlossen, return="foo" 4. createThumbnail geplant
Erneut abspielen	
1. Bereitstellungsschleife 2. getImageUrls 3. downloadImage image path="foo" 4. createThumbnail thumbnail path="bar" 5. uploadImage 6. <nächster Durchlauf der Schleife>	1. Workflow-Instance gestartet, id="1" 2. downloadImage geplant 3. downloadImage abgeschlossen, return="foo" 4. createThumbnail geplant 5. createThumbnail abgeschlossen, return="bar" 6. uploadImage geplant
Erneut abspielen	

Workflow-Programmausführung	Von Amazon SWF verwalteter Verlauf
1. Bereitstellungsschleife 2. getImageUrls 3. downloadImage image path="foo" 4. createThumbnail thumbnail path="bar" 5. uploadImage 6. <nächster Durchlauf der Schleife>	1. Workflow-Instance gestartet, id="1" 2. downloadImage geplant 3. downloadImage abgeschlossen, return="foo" 4. createThumbnail geplant 5. createThumbnail abgeschlossen, return="bar" 6. uploadImage geplant 7. uploadImage abgeschlossen ...

Wenn ein Aufruf von `processImage` erfolgt, erstellt das Framework eine neue Workflow-Instanz in Amazon SWF. Dies ist ein dauerhafter Datensatz der gestarteten Workflow-Instance. Das Programm wird bis zum Aufruf der `downloadImage` Aktivität ausgeführt, wodurch Amazon SWF aufgefordert wird, eine Aktivität zu planen. Der Workflow wird weiter ausgeführt und erstellt Aufgaben für nachfolgende Aktivitäten. Sie können jedoch erst ausgeführt werden, wenn die `downloadImage` Aktivität abgeschlossen ist. Somit endet diese Episode der Wiederholung. Amazon SWF sendet die Aufgabe für die `downloadImage` Aktivität zur Ausführung. Sobald sie abgeschlossen ist, wird zusammen mit dem Ergebnis ein Eintrag in der Historie erstellt. Der Workflow ist jetzt bereit, fortzufahren, und eine Entscheidungsaufgabe wird von Amazon SWF generiert. Das Framework empfängt die Entscheidungsaufgabe und wiederholt den Workflow, wobei das Ergebnis des heruntergeladenen Abbilds wie im Verlauf aufgezeichnet eingefügt wird. Dadurch wird die Blockierung der Aufgabe für `createThumbnail` aufgehoben, und die Ausführung des Programms wird weiter fortgesetzt, indem die `createThumbnail` Aktivitätsaufgabe in Amazon SWF geplant wird. Derselbe Prozess wird für `uploadImage` wiederholt. Die Ausführung des Programms geht auf diese Weise weiter, bis der Workflow alle Abbilder verarbeitet hat und es keine ausstehenden Aufgaben gibt. Da kein Ausführungsstatus lokal gespeichert ist, kann jede Entscheidungsaufgabe möglicherweise auf einem anderen Computer ausgeführt werden. Dadurch können Sie ganz einfach Programme schreiben, die fehlertolerant und problemlos skalierbar sind.

Nichtdeterminismus

Da das Framework auf der Wiedergabe basiert, ist es wichtig, dass der Orchestrierungscode (der gesamte Workflow-Code mit Ausnahme von Aktivitätsimplementierungen) deterministisch ist. Beispielsweise sollte der Steuerungsfluss in Ihrem Programm nicht von einer zufälligen Zahl oder der aktuellen Zeit abhängen. Da sich diese Dinge zwischen Aufrufen ändern, folgt die Wiedergabe möglicherweise nicht demselben Pfad durch die Orchestrierungslogik. Dies führt zu unerwarteten Ergebnissen oder Fehlern. Das Framework bietet einen `WorkflowClock`, den Sie verwenden können, um die aktuelle Zeit auf deterministische Weise abzurufen. Weitere Informationen finden Sie im Abschnitt zu Ausführungskontext.

> **Note**
>
> Eine falsche Spring-Verdrahtung der Workflow-Implementierungsobjekte kann auch zu Nichtdeterminismus führen. Workflow-Implementierungs-Beans sowie Beans, von denen sie abhängig sind, müssen sich im Workflow-Umfang (`WorkflowScope`) befinden. Beispielsweise führt das Vertraten einer Workflow-Implementierungs-Bean mit einer Bean, die den Zustand behält und sich im globalen Kontext befindet, zu unerwartetem Verhalten. Weitere Details finden Sie im Abschnitt Spring-Integration.

AWS Flow Framework für Java-Programmierhandbuch

Dieser Abschnitt enthält Einzelheiten zur Verwendung der Funktionen von AWS Flow Framework for Java zur Implementierung von Workflow-Anwendungen.

Themen

- Implementierung von Workflow-Anwendungen mit dem AWS Flow Framework
- Workflow- und Aktivitäts-Verträge
- Registrierung von Workflow- und Aktivitätstypen
- Aktivitäts- und Workflow-Clients
- Workflow-Implementierung
- Implementierung von Aktivitäten
- AWS Lambda Aufgaben umsetzen
- Ausführen von Programmen, die mit dem AWS Flow Framework für Java geschrieben wurden
- Ausführungskontext
- Untergeordnete Workflow-Ausführungen
- Fortlaufende Workflows
- Aufgabenpriorität in Amazon SWF festlegen
- DataConverters
- Datenübergabe an asynchrone Methoden
- Prüfbarkeit und Dependency Injection
- Fehlerbehandlung
- Wiederholen fehlgeschlagener Aktivitäten
- Daemon-Aufgaben
- AWS Flow Framework für Java Replay Behavior

Implementierung von Workflow-Anwendungen mit dem AWS Flow Framework

Die typischen Schritte bei der Entwicklung eines Workflows mit dem AWS Flow Framework sind:

1. Definieren Sie Aktivitäts- und Workflow-Verträge. Analysieren Sie die Anforderungen Ihrer Anwendung und bestimmen Sie die erforderlichen Aktivitäten sowie die Workflow-Topologie. Die Aktivitäten betreffen die erforderlichen Verarbeitungsaufgaben, während die Workflow-Topologie die grundlegende Struktur und die Geschäftslogik des Workflows definiert.

   Eine Medien verarbeitende Anwendung muss z. B. eine Datei herunterladen, verarbeiten und die verarbeitete Datei in einen Amazon Simple Storage Service (S3)-Bucket herunterladen. Dieser Prozess lässt sich in vier Aktivitätsaufgaben gliedern:

   1. Die Datei von einem Server herunterladen
   2. Die Datei verarbeiten (z. B. durch Transcodieren in ein anderes Medienformat)
   3. Die Datei in den S3-Bucket hochladen
   4. Eine Bereinigung durch Löschen der lokalen Dateien durchführen

   Dieser Workflow verfügt über eine Eintrittspunktmethode und implementiert eine einfache lineare Topologie, die die Aktivitäten nacheinander ausführt, ähnlich wie HelloWorldWorkflow Bewerbung.

2. Implementieren Sie Aktivitäts- und Workflow-Schnittstellen. Die Workflow- und Aktivitätsverträge werden durch Java-Schnittstellen definiert, durch die ihre Aufrufkonventionen von SWF prognostizierbar werden und Sie Flexibilität beim Implementieren Ihrer Workflow-Logik und Aktivitätsaufgaben erhalten. Die verschiedenen Teile Ihres Programms können als Consumer der Daten des jeweils anderen agieren, müssen jedoch die Implementierungsdetails der anderen Teile nicht alle kennen.

   Sie können z. B. eine `FileProcessingWorkflow`-Schnittstelle definieren und verschiedene Workflow-Implementierungen für Videocodierung, Komprimierung, Thumbnails usw. bereitstellen. Jeder dieser Workflows kann über verschiedene Kontrollabläufe verfügen und unterschiedliche Aktivitätsmethoden aufrufen, ohne dass Ihr Workflow-Starter davon Kenntnis haben muss. Mit Schnittstellen können Sie Ihre Workflows ganz einfach testen, indem Sie Pseudoimplementierungen verwenden, die später durch funktionierenden Code ersetzt werden können.

3. Generieren Sie Aktivitäts- und Workflow-Clients. AWS Flow Framework Dadurch müssen Sie die Einzelheiten der Verwaltung der asynchronen Ausführung, des Sendens von HTTP-Anfragen, des Marshallings von Daten usw. nicht mehr implementieren. Stattdessen führt der Workflow-Starter durch Aufrufen einer Methode auf dem Workflow-Client eine Workflow-Instance aus und

die Workflow-Implementierung führt Aktivitäten durch Aufrufen von Methoden auf dem Aktivitäts-Client aus. Das Framework verarbeitet die Details dieser Interaktionen im Hintergrund.

Wenn Sie Eclipse verwenden und Ihr Projekt wie in konfiguriert haben, verwendet der AWS Flow Framework Annotationsprozessor die SchnittstellendefinitionenEinrichtung des AWS Flow Framework für Java, um automatisch Workflow- und Aktivitätsclients zu generieren, die dieselben Methoden wie die entsprechende Schnittstelle bereitstellen.

4. Implementieren Sie Aktivitäts- und Workflow-Hostanwendungen. Ihre Workflow- und Aktivitätsimplementierungen müssen in Hostanwendungen eingebettet sein, die Amazon SWF nach Aufgaben abfragen, alle Daten zusammenführen und die entsprechenden Implementierungsmethoden aufrufen. AWS Flow Framework für Java beinhaltet WorkflowWorkerund ActivityWorkerKlassen, die die Implementierung von Hostanwendungen unkompliziert und einfach machen.
5. Testen Sie Ihren Arbeitsablauf. AWS Flow Framework for Java bietet eine JUnit Integration, mit der Sie Ihre Workflows inline und lokal testen können.
6. Stellen Sie die Worker bereit. Sie können Ihre Mitarbeiter nach Bedarf einsetzen — Sie können sie beispielsweise auf EC2 Amazon-Instances oder auf Computern in Ihrem Rechenzentrum einsetzen. Nach der Bereitstellung und dem Start beginnen die Worker, Amazon SWF nach Aufgaben abzufragen und diese nach Bedarf zu bearbeiten.
7. Starten Sie die Ausführungen. Eine Anwendung startet eine Workflow-Instance, indem der Workflow-Client zum Abrufen des Eintrittspunkts des Workflows verwendet wird. Sie können Workflows auch mithilfe der Amazon SWF SWF-Konsole starten. Unabhängig davon, wie Sie eine Workflow-Instance starten, können Sie die Amazon SWF SWF-Konsole verwenden, um die laufende Workflow-Instanz zu überwachen und den Workflow-Verlauf auf laufende, abgeschlossene und fehlgeschlagene Instances zu untersuchen.

Das AWS SDK für Javabeinhaltet eine Reihe von AWS Flow Framework Java-Beispielen, die Sie durchsuchen und ausführen können, indem Sie den Anweisungen in der Datei readme.html im Stammverzeichnis folgen. Es gibt auch eine Reihe von Rezepten — einfache Anwendungen —, die zeigen, wie man mit einer Vielzahl von spezifischen Programmierproblemen umgeht. Diese finden Sie unter AWS Flow Framework Rezepte.

Workflow- und Aktivitäts-Verträge

Java-Schnittstellen werden zum Deklarieren der Signaturen der Workflows und Aktivitäten verwendet. Die Schnittstelle bildet den Vertrag zwischen der Implementierung des Workflows (oder der

Aktivität) und dem Client dieses Workflows (oder der Aktivität). Ein Workflow-Typ `MyWorkflow` wird beispielsweise mithilfe einer Schnittstelle definiert, die mit der `@Workflow`-Anmerkung versehen ist:

```
@Workflow
@WorkflowRegistrationOptions(
   defaultExecutionStartToCloseTimeoutSeconds = 60,
   defaultTaskStartToCloseTimeoutSeconds = 10)
public interface MyWorkflow
{
    @Execute(version = "1.0")
    void startMyWF(int a, String b);

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

    @GetState
    MyWorkflowState getState();
}
```

Der Vertrag hat keine implementierungsspezifischen Einstellungen. Diese Nutzung der implementierungsneutralen Verträge ermöglicht es, dass Client von der Implementierung entkoppelt werden, und bietet dadurch die Flexibilität, die Implementierungsdetails zu ändern, ohne den Client zu brechen. Umgekehrt können Sie den Client auch ändern, ohne zu nötigen, dass Änderungen am Workflow oder der Aktivität verbraucht werden. Beispielsweise kann der Client so geändert werden, dass er eine Aktivität asynchron unter Verwendung von Promises (`Promise<T>`) aufruft, ohne eine Änderung an der Aktivitätsimplementierung zu erfordern. In ähnlicher Weise kann die Implementierung der Aktivität so geändert werden, dass sie asynchron abgeschlossen wird, z. B. durch eine Person, die eine E-Mail sendet, ohne dass die Clients der Aktivität geändert werden müssen.

Im Beispiel oben enthält die Workflow-Instance `MyWorkflow` eine Methode, `startMyWF`, für das Starten einer neuen Ausführung. Diese Methode wird mit der `@Execute`-Anmerkung versehen und muss einen Rückgabetyp von `void` oder `Promise<>` haben. In einer gegebenen Workflow-Schnittstelle kann maximal eine Methode mit dieser Anmerkung versehen werden. Diese Methode ist der Eintrittspunkt der Workflow-Logik und das Framework ruft diese Methode auf, um die Workflow-Logik auszuführen, wenn eine Entscheidungsaufgabe empfangen wird.

Die Workflow-Schnittstelle definiert auch die Signale, die an den Workflow gesendet werden können. Die Signalmethode wird aufgerufen, wenn ein Signal mit einem passenden Namen von der

Workflow-Ausführung empfangen wird. Beispielsweise deklariert die `MyWorkflow`-Schnittstelle eine Signalmethode, `signal1`, mit der Anmerkung `@Signal` versehen.

Die `@Signal`-Anmerkung ist auf Signalmethoden erforderlich. Der Rückgabetyp einer Signalmethode muss `void` sein. Eine Workflow-Schnittstelle kann null oder mehrere Signalmethoden in ihr definiert haben. Sie können eine Workflow-Schnittstelle ohne eine `@Execute`-Methode deklarieren und einige `@Signal`-Methoden zum Generieren von Clients deklarieren, die ihre Ausführung nicht starten, aber Signal an laufende Ausführungen senden können.

Methoden, die mit den Anmerkungen `@Execute` und `@Signal` versehen sind, können eine beliebige Anzahl an Parametern jeden Typ haben, abgesehen von `Promise<T>` oder seinen Derivaten. Dadurch können Sie stark typisierte Eingaben beim Start und während der Ausführung an eine Workflow-Ausführung übergeben. Der Rückgabetyp der `@Execute`-Methode muss `void` oder `Promise<>` sein.

Zudem können Sie auch eine Methode in der Workflow-Schnittstelle deklarieren, um den aktuellen Zustand einer Workflow-Ausführung zu melden, zum Beispiel die `getState`-Methode im vorherigen Beispiel. Dieser Zustand ist nicht der gesamte Anwendungszustand des Workflows. Die vorgesehene Nutzung dieser Funktion ist, die Speicherung von bis zu 32 KB an Daten zuzulassen, um den aktuellen Status der Ausführung anzugeben. In einem Bestellvorgangs-Workflow können Sie auch eine Zeichenfolge speichern, die angibt, dass die Bestellung eingegangen, verarbeitet oder storniert wurde. Diese Methode wird jedes Mal, wenn eine Entscheidungsaufgabe abgeschlossen wurde, vom Framework aufgerufen, um den aktuellen Zustand zu erhalten. Der Status wird in Amazon Simple Workflow Service (Amazon SWF) gespeichert und kann mit dem generierten externen Client abgerufen werden. Auf diese Weise können Sie den aktuellen Zustand einer Workflow-Ausführung prüfen. Mit `@GetState` versehene Methoden dürfen keine Argumente akzeptieren und dürfen nicht den Rückgabetyp `void` haben. Sie können jeden Typ, der ihren Anforderungen entspricht, von dieser Methode zurückgeben. Im Beispiel oben wird ein Objekt `MyWorkflowState` (siehe Definition unten) von der Methode zurückgegeben, die verwendet wird, um einen Zeichenfolgenzustand und einen numerischen Prozentabschluss zu speichern. Diese Methode soll einen schreibgeschützten Zugriff auf das Workflow-Implementierungsobjekt durchführen und wird synchron aufgerufen, wodurch die Verwendung von asynchronen Operationen, wie das Aufrufen von Methoden mit der Anmerkung `@Asynchronous` nicht mehr zulässig ist. In einer Workflow-Schnittstelle kann maximal eine Methode mit dieser Anmerkung `@GetState` versehen werden.

```
public class MyWorkflowState {
   public String status;
   public int percentComplete;
```

```
}
```

Gleichermaßen wird eine Reihe von Aktivitäten mit einer Schnittstelle mit der Anmerkung `@Activities` definiert. Jede Methode in der Schnittstelle entspricht einer Aktivität — zum Beispiel:

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

    void activity2(int a);
}
```

Über die Schnittstelle können Sie eine Reihe von verwandten Aktivitäten gruppieren. Sie können eine beliebige Anzahl von Aktivitäten innerhalb einer Aktivitäten-Schnittstelle definieren und Sie können so viele Aktivitäten-Schnittstellen definieren wie Sie möchten. Ähnlich wie die Methoden `@Execute` und `@Signal` können Aktivitätsmethoden eine beliebige Anzahl an Argumenten jeden Typs akzeptieren, abgesehen von `Promise<T>` oder seinen Derivaten. Der Rückgabetyp einer Aktivität darf nicht `Promise<T>` oder seine Derivate sein.

Registrierung von Workflow- und Aktivitätstypen

Amazon SWF erfordert, dass Aktivitäts- und Workflowtypen registriert werden, bevor sie verwendet werden können. Das Framework registriert die Workflows und Aktivitäten automatisch in den Implementierungen, die Sie dem Worker hinzufügen. Das Framework sucht nach Typen, die Workflows und Aktivitäten implementieren, und registriert sie bei Amazon SWF. Das Framework verwendet standardmäßig die Schnittstellendefinitionen, um Registrierungsoptionen für Workflow- und Aktivitätstypen abzuleiten. Alle Workflow-Schnittstellen müssen entweder über die `@WorkflowRegistrationOptions`-Annotation oder die `@SkipRegistration`-Annotation verfügen. Der Workflow-Worker registriert alle Workflow-Typen, mit denen er konfiguriert ist, die über die `@WorkflowRegistrationOptions`-Annotation verfügen. Gleichermaßen muss jede Aktivitätsmethode mit entweder der `@ActivityRegistrationOptions`-Annotation oder der `@SkipRegistration`-Annotation versehen sein oder es muss eine dieser Annotationen in der

`@Activities`-Schnittstelle vorhanden sein. Der Aktivitäts-Worker registriert alle Aktivitäts-Typen, mit denen er konfiguriert ist, für die eine `@ActivityRegistrationOptions`-Annotation gilt. Die Registrierung wird beim Starten einer der Worker automatisch durchgeführt. Workflow- und Aktivitätsarten, die über `@SkipRegistration`-Annotation verfügen, werden nicht registriert. `@ActivityRegistrationOptions`und `@SkipRegistration`-Annotationen besitzen eine Übersteuersemantik und die spezifischste wird auf einen Aktivitätstyp angewendet.

Beachten Sie, dass Amazon SWF es Ihnen nicht erlaubt, den Typ erneut zu registrieren oder zu ändern, nachdem er registriert wurde. Das Framework wird versuchen, alle Typen zu registrieren, aber wenn der Typ bereits registriert ist, wird er nicht erneut registriert und es wird kein Fehler gemeldet.

Wenn Sie registrierte Einstellungen ändern möchten, müssen Sie eine neue Version des Typs registrieren. Sie können registrierte Einstellungen auch beim Starten einer neuen Ausführung oder beim Aufrufen einer Aktivität, die die generierten Clients verwendet, überschreiben.

Die Registrierung erfordert einen Typnamen und andere Registrierungsoptionen. Die Standardimplementierungen bestimmt diese wie folgt:

Workflow-Typname und Version

Das Framework bestimmt den Namen des Workflow-Typs über die Workflow-Schnittstelle. Die Form des Standard-Workflow-Typnamens lautet {*`prefix`*} {*`name`*}. {*`prefix`*} ist auf den Namen der `@Workflow` Schnittstelle gefolgt von einem '.' gesetzt und {*`name`*} ist auf den Namen der `@Execute` Methode gesetzt. Der Standardname des Workflow-Typs im vorhergehenden Beispiel lautet `MyWorkflow.startMyWF`. Sie können den Standardnamen mithilfe des Namenparameters der `@Execute`-Methode überschreiben. Der Standardname des Workflow-Typs im Beispiel lautet `startMyWF`. Der Name darf keine leere Zeichenfolge sein. Beachten Sie, dass beim Überscheiben des Namens mit `@Execute` das Framework diesem nicht automatisch ein Präfix voranstellt. Es steht Ihnen frei, Ihr eigenes Namensschema zu verwenden.

Die Workflow-Version wird mit dem `version`-Parameter der `@Execute`-Annotation angegeben. Es gibt keinen Standard für `version` und es muss explizit angegeben werden. `version`ist eine formfreie Zeichenfolge und es steht Ihnen frei, Ihr eigenes Versioning-Schema zu verwenden.

Signalname

Der Name des Signals kann mit dem Namenparameter der `@Signal`-Annotation angegeben werden. Wenn nicht angegeben, gilt standardmäßig der Name der Signalmethode.

Aktivitätstypname und Version

Das Framework bestimmt den Namen des Aktivitätstypen über die Aktivitätenschnittstelle. Die Form des Standardnamens für den Aktivitätstyp ist {*prefix*} {*name*}. {*prefix*} ist auf den Namen der `@Activities` Schnittstelle gefolgt von einem '.' gesetzt und {*name*} ist auf den Methodennamen gesetzt. Die Standardeinstellung {*prefix*} kann in der `@Activities` Anmerkung auf der Aktivitätsschnittstelle außer Kraft gesetzt werden. Sie können den Namen des Aktivitätstyps auch mit der `@Activity`-Annotation in der Aktivitätsmethode angeben. Beachten Sie, dass beim Überscheiben des Namens mit `@Activity` das Framework diesem nicht automatisch ein Präfix voranstellt. Es steht Ihnen frei, Ihr eigenes Namensschema zu verwenden.

Die Aktivitätsversion wird mit dem Versionsparameter der `@Activities`-Annotation angegeben. Diese Version wird als Standard für alle Aktivitäten verwendet, die in der Schnittstelle definiert sind, und kann pro Aktivität mit der `@Activity`-Annotation überschrieben werden.

Standardaufgabenliste

Die Standardaufgabenliste kann mit den Annotationen `@WorkflowRegistrationOptions` und `@ActivityRegistrationOptions` und durch Festlegen des `defaultTaskList`-Parameters konfiguriert werden. Standardmäßig ist der Wert eingestell `USE_WORKER_TASK_LIST`. Dies ist ein spezieller Wert, der das Framework anweist, die Aufgabenliste zu verwenden, die in dem Worker-Objekt konfiguriert ist, das für die Registrierung des Aktivitäts- oder Workflow-Typs verwendet wird. Sie können eine Standardaufgabenliste auch nicht registrieren, indem Sie sie mit diesen Annotationen auf `NO_DEFAULT_TASK_LIST` festlegen. Dies kann in Fällen verwendet werden, bei denen Sie festlegen möchten, dass die Aufgabenliste zur Laufzeit angegeben werden soll. Wenn keine Standardaufgabenliste registriert wurde, müssen Sie die Aufgabenliste beim Starten des Workflows oder beim Aufrufen der Aktivitätsmethode mit den Parametern `StartWorkflowOptions` und `ActivitySchedulingOptions` für die jeweilige Methodenüberladung des generierten Clients angeben.

Weitere Registrierungsoptionen

Alle Registrierungsoptionen für Workflows und Aktivitätstypen, die von der Amazon SWF SWF-API zugelassen werden, können über das Framework angegeben werden.

Eine vollständige Liste der Workflow-Registrierungsoptionen finden Sie im Folgenden:

- @Workflow
- @Execute

- @WorkflowRegistrationOptions
- @Signal

Eine vollständige Liste der Aktivitäts-Registrierungsoptionen finden Sie im Folgenden:

- @Aktivität
- @Aktivität
- @ActivityRegistrationOptions

Wenn Sie die vollständige Kontrolle über die Registrierung von Typen haben möchten, lesen Sie Worker-Erweiterbarkeit.

Aktivitäts- und Workflow-Clients

Workflow- und Aktivitäts-Clients werden von dem Framework generiert, das auf den Schnittstellen `@Workflow` und `@Activities` basiert. Es werden separate Client-Schnittstellen generiert, die Methoden und Einstellungen enthalten, die nur innerhalb des Clients Sinn ergeben. Wenn Sie mit Eclipse entwickeln, erledigt dies das Amazon SWF Eclipse-Plug-In jedes Mal, wenn Sie die Datei mit der entsprechenden Schnittstelle speichern. Der generierte Code wird im generierten Quellverzeichnis Ihres Projekts im selben Paket platziert wie die Schnittstelle.

> **Note**
>
> Beachten Sie, dass der von Eclipse verwendete Standardname für das Verzeichnis .apt_generated lautet. Eclipse zeigt keine Verzeichnisse, deren Namen mit beginnen. Wenn Sie im Package Explorer. Wenn Sie die generierten Dateien im Project Explorer anzeigen möchten, verwenden Sie einen anderen Verzeichnisnamen. Klicken Sie in Eclipse mit der rechten Maustaste auf das Paket im Package Explorer, wählen Sie dann Properties (Eigenschaften), Java Compiler, Annotation processing (Annotationen verarbeiten) und ändern Sie die Einstellung Generate source directory (Quellverzeichnis generieren).

Workflow-Clients

Die generierten Artefakte für den Workflow enthalten drei clientseitige Schnittstellen und die Klassen, die sie implementieren. Die generierten Clients umfassen Folgendes:

- Einen asynchronen Client, der aus einer Workflow-Implementierung heraus verbraucht werden soll, die asynchrone Methoden bereitstellt, die den Workflow ausführen und Signale senden.
- Einen externen Client, der verwendet werden kann, um Ausführungen zu starten, Signale zu senden und den Workflow-Status von außerhalb einer Workflow-Implementierung zu empfangen.
- Einen Self-Client, der verwendet werden kann, um einen kontinuierlichen Workflow zu generieren.

So lauten z.B. die generierten Client-Schnittstellen für die Musterschnittstelle `MyWorkflow` wie folgt:

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

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

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

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

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

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

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

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

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

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

    MyWorkflowState getState();
}

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

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

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

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

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

    void startMyWF(
       Promise<Integer> a,
       Promise<String> b,
```

```
        StartWorkflowOptions optionsOverride,
        Promise<?>... waitFor);
```

Die Schnittstellen haben eine Methodenüberladungen, die jeweils der in der Schnittstelle deklarierten Methode `@Workflow` entsprechen.

Der externe Client spiegelt die Methoden der Schnittstelle `@Workflow` mit einer zusätzlichen Methodenüberladung `@Execute`, die `StartWorkflowOptions` verwendet. Sie können mit dieser Überladung zusätzliche Optionen weiterleiten, wenn Sie eine neue Workflow-Ausführung starten. Diese Optionen ermöglichen Ihnen die Standardaufgabenliste, die Timeout-Einstellungen und die zugehörigen Tags mit der Workflow-Ausführung zu überschreiben.

Auf der anderen Seite verfügt der asynchrone Client über Methoden, die einen asynchronen Aufruf der Methode `@Execute` ermöglichen. Die folgenden Methodenüberladungen werden in der Client-Schnittstelle für die Methode `@Execute` in der Workflow-Schnittstelle generiert:

1. Eine Überladung, welche die ursprünglichen Argumente unverändert übernimmt. Der Rückgabetyp dieser Überladung ist `Promise<Void>`, wenn die ursprüngliche Methode `void` zurückgegeben hat; andernfalls ist es `Promise<>`, wie in der ursprünglichen Methode deklariert. Zum Beispiel:

   Ursprüngliche Methode:

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

   Generierte Methode:

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

   Diese Überladung sollte verwendet werden, wenn alle Argumente des Workflows verfügbar sind und nicht auf diese gewartet werden muss.
2. Eine Überladung, die die ursprünglichen Argumente unverändert sowie zusätzliche variable Argumente des Typs `Promise<?>` übernimmt. Der Rückgabetyp dieser Überladung ist `Promise<Void>`, wenn die ursprüngliche Methode `void` zurückgegeben hat; andernfalls ist es `Promise<>`, wie in der ursprünglichen Methode deklariert. Zum Beispiel:

   Ursprüngliche Methode:

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

Generierte Methode:

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

Diese Überladung sollte verwendet werden, wenn alle Argumente des Workflows verfügbar sind und nicht auf diese gewartet werden muss, wenn Sie jedoch darauf warten möchten, dass ein anderes Promise betriebsbereit wird. Das variable Argument kann verwendet werden, um Objekte wie `Promise<?>` weiterzuleiten, die nicht als Argumente deklariert waren, wenn Sie noch mit der Ausführung des Aufrufs warten möchten.

3. Eine Überladung, die die ursprünglichen Argumente unverändert übernimmt sowie ein zusätzliches Argument des Typs `StartWorkflowOptions` und ein zusätzliches Argument des Typs `Promise<?>`. Der Rückgabetyp dieser Überladung ist `Promise<Void>`, wenn die ursprüngliche Methode `void` zurückgegeben hat; andernfalls ist es `Promise<>`, wie in der ursprünglichen Methode deklariert. Zum Beispiel:

   Ursprüngliche Methode:

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

   Generierte Methode:

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

   Diese Überladung sollte verwendet werden, wenn alle Argumente des Workflows verfügbar sind und nicht auf diese gewartet werden muss, wenn Sie Standardeinstellungen überschreiben und die Workflow-Ausführung starten möchten oder wenn Sie darauf warten möchten, dass ein anderes Promise betriebsbereit wird. Das variable Argument kann verwendet werden, um Objekte wie `Promise<?>` weiterzuleiten, die nicht als Argumente deklariert waren, wenn Sie noch mit der Ausführung des Aufrufs warten möchten.

4. Eine Überladung mit jedem Argument in der ursprünglichen Methode, die durch einen `Promise<>`-Wrapper ersetzt wurde. Der Rückgabetyp dieser Überladung ist `Promise<Void>`,

wenn die ursprüngliche Methode `void` zurückgegeben hat; andernfalls ist es `Promise<>`, wie in der ursprünglichen Methode deklariert. Zum Beispiel:

Ursprüngliche Methode:

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

Generierte Methode:

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

Diese Überladung sollte verwendet werden, wenn die Argumente, die an die Workflow-Ausführung weitergeleitet werden sollen, asynchron evaluiert werden müssen. Ein Aufruf dieser Methodenüberladung wird so lange nicht ausgeführt, bis alle Argumente, die an diese weitergeleitet wurden, betriebsbereit sind.

Wenn einige der Argumente bereits betriebsbereit sind, dann konvertieren Sie diese in ein `Promise`, das durch die Methode `Promise.asPromise(`*`value`*`)` bereits im betriebsbereiten Status ist. Zum Beispiel:

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

5. Eine Überladung mit jedem Argument in der ursprünglichen Methode wird durch den Wrapper `Promise<>` ersetzt. Die Überladung hat auch unterschiedliche variable Argumente des Typs `Promise<?>`. Der Rückgabetyp dieser Überladung ist `Promise<Void>`, wenn die ursprüngliche Methode `void` zurückgegeben hat; andernfalls ist es `Promise<>`, wie in der ursprünglichen Methode deklariert. Zum Beispiel:

   Ursprüngliche Methode:

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

   Generierte Methode:

   ```
   Promise<Void> startMyWF(
   ```

```
   Promise<Integer> a,
   Promise<String> b,
   Promise<?>...waitFor);
```

Diese Überladung sollte verwendet werden, wenn die Argumente, die an die Workflow-Ausführung weitergeleitet wurden, asynchron evaluiert werden und wenn Sie darauf warten möchten, dass ein anderes Promise ebenfalls betriebsbereit wird. Ein Aufruf dieser Methodenüberladung wird so lange nicht ausgeführt, bis alle Argumente, die an diese weitergeleitet wurden, betriebsbereit sind.

6. Eine Überladung mit jedem Argument in der ursprünglichen Methode, die durch einen `Promise<?>`-Wrapper ersetzt wurde. Die Überladung enthält auch ein zusätzliches Argument des Typs `StartWorkflowOptions` und variable Argumente des Typs `Promise<?>`. Der Rückgabetyp dieser Überladung ist `Promise<Void>`, wenn die ursprüngliche Methode `void` zurückgegeben hat; andernfalls ist es `Promise<>`, wie in der ursprünglichen Methode deklariert. Zum Beispiel:

   Ursprüngliche Methode:

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

   Generierte Methode:

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

   Verwenden Sie diese Überladung, wenn die Argumente, die an die Workflow-Ausführung weitergeleitet werden sollen, asynchron evaluiert werden und wenn Sie die Standardeinstellungen überschreiben möchten, die zum Starten der Workflow-Ausführung verwendet werden. Ein Aufruf dieser Methodenüberladung wird so lange nicht ausgeführt, bis alle Argumente, die an diese weitergeleitet wurden, betriebsbereit sind.

Außerdem wird für jedes Signal in der Workflow-Oberfläche eine Methode generiert — zum Beispiel:

Ursprüngliche Methode:

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

Generierte Methode:

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

Der asynchrone Client enthält keine Methode, welche der Methode entspricht, die in der ursprünglichen Schnittstelle mit `@GetState` annotiert wurde. Da das Abrufen des Status einen Webservice-Aufruf erfordert, ist er nicht für die Verwendung innerhalb eines Workflows geeignet. Daher kann er nur über einen externen Client zur Verfügung gestellt werden.

Der Self-Client soll in einem Workflow verwendet werden, um eine neue Ausführung nach Beendigung der aktuellen Ausführung zu starten. Die Methoden auf diesem Client sind den Methoden auf dem asynchronen Client ähnlich, aber die Rückgabe ist `void`. Der Client enthält keine Methoden, welche den Methode entsprechen, die mit `@Signal` und `@GetState` annotiert wurden. Weitere Informationen hierzu finden Sie unter Fortlaufende Workflows.

Die generierten Clients sind von den Basisschnittstellen `WorkflowClient` und `WorkflowClientExternal` abgeleitet, die Methoden bereitstellen, die Sie zum Abbrechen oder Beenden der Workflow-Ausführung verwenden können. Weitere Informationen zu diesen Schnittstellen finden Sie in der AWS SDK für Java -Dokumentation.

Die generierten Clients ermöglichen die Interaktion mit den Workflow-Ausführungen in einer stark typisierten Form. Eine Instance eines generierten Client, wird, wenn sie einmal erstellt wurde, mit einer spezifischen Workflow-Ausführung verknüpft und kann nur für diese Ausführung verwendet werden. Außerdem stellt das Framework dynamische Clients bereit, die nicht für einen Workflow-Typ oder eine Ausführung typisch sind. Die generierten Clients basieren verdeckt auf diesem Client. Sie können diese Clients auch direkt verwenden. Sehen Sie sich den Abschnitt zu Dynamische Clients an.

Das Framework generiert auch Fabriken zum Erstellen stark typisierter Clients. Die generierten Client-Fabriken für die Musterschnittstelle `MyWorkflow` sind:

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

//Factory for clients to be used outside the scope of a workflow
public interface MyWorkflowClientExternalFactory
{
```

```
   GenericWorkflowClientExternal getGenericClient();
   void setGenericClient(GenericWorkflowClientExternal genericClient);
   DataConverter getDataConverter();
   void setDataConverter(DataConverter dataConverter);
   StartWorkflowOptions getStartWorkflowOptions();
   void setStartWorkflowOptions(StartWorkflowOptions startWorkflowOptions);
   MyWorkflowClientExternal getClient();
   MyWorkflowClientExternal getClient(String workflowId);
   MyWorkflowClientExternal getClient(WorkflowExecution workflowExecution);
   MyWorkflowClientExternal getClient(
      WorkflowExecution workflowExecution,
      GenericWorkflowClientExternal genericClient,
      DataConverter dataConverter,
      StartWorkflowOptions options);
}
```

Die Basisschnittstelle für `WorkflowClientFactory` ist:

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

Sie sollten diese Fabriken zum Erstellen von Client-Instances verwenden. Mit der Fabrik können Sie den generischen Client (der generische Client sollte für die Bereitstellung der benutzerdefinierten Client-Implementierung verwendet werden) und den `DataConverter`, der vom Client zum Daten-Marshalling verwendet wird, sowie die Optionen, die verwendet werden, um die Workflow-Ausführung zu starten, konfigurieren. Weitere Informationen finden Sie unter den Abschnitten DataConverters und Untergeordnete Workflow-Ausführungen. Das `StartWorkflowOptions` enthält Einstellungen, mit denen Sie die bei der Registrierung angegebenen Standardwerte (z. B. Timeouts)

außer Kraft setzen können. Weitere Informationen zur Klasse finden Sie in der Dokumentation. `StartWorkflowOptions` AWS SDK für Java

Einen externen Client, der verwendet werden kann, um Workflow-Ausführungen außerhalb eines Workflows zu starten, während der asynchrone Client verwendet werden kann, um eine Workflow-Ausführung vom Code innerhalb eines Workflows zu starten. Zum Starten einer Ausführung verwenden Sie einfach den generierten Client, um die Methode aufzurufen, welche der Methode entspricht, die `@Execute` in der Workflow-Schnittstelle annotiert ist.

Das Framework generiert auch Implementierungsklassen für die Client-Schnittstellen. Diese Clients erstellen Anfragen und senden sie an Amazon SWF, um die entsprechende Aktion auszuführen. Die Client-Version der `@Execute` Methode startet entweder eine neue Workflow-Ausführung oder erstellt eine untergeordnete Workflow-Ausführung mit Amazon SWF APIs. In ähnlicher Weise verwendet die Client-Version der `@Signal` Methode Amazon SWF APIs , um ein Signal zu senden.

> **Note**
>
> Der externe Workflow-Client muss mit dem Amazon SWF-Client und der Domain konfiguriert sein. Sie können entweder den Client Factory-Konstruktor verwenden, der diese als Parameter verwendet, oder eine generische Client-Implementierung übergeben, die bereits mit dem Amazon SWF-Client und der Domain konfiguriert ist.
> Das Framework durchläuft die Typenhierarchie der Workflow-Schnittstelle und generiert auch Client-Schnittstellen für übergeordnete Workflow-Schnittstellen und leitet sich aus diesen ab.

Aktivitäts-Clients

Ähnlich wie ein Workflow-Client, wird für jede Schnittstelle ein Client generiert, der mit `@Activities` annotiert ist. Die generierten Artefakte umfassen eine clientseitige Schnittstelle und eine Client-Klasse. Die generierte Schnittstelle für die oben genannte Musterschnittstelle `@Activities` (`MyActivities`) lautet wie folgt:

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

```
                          Promise<?>... waitFor);
  Promise<Void> activity2(int a,
                          ActivitySchedulingOptions optionsOverride,
                          Promise<?>... waitFor);
  Promise<Void> activity2(Promise<Integer> a);
  Promise<Void> activity2(Promise<Integer> a,
                          Promise<?>... waitFor);
  Promise<Void> activity2(Promise<Integer> a,
                          ActivitySchedulingOptions optionsOverride,
                          Promise<?>... waitFor);
}
```

Die Schnittstelle enthält eine Menge von Methodenüberladungen, die jeweils der Aktivitätsmethode in der Schnittstelle `@Activities` entsprechen. Die Überladungen dienen der Bequemlichkeit und ermöglichen den asynchronen Aufruf von Aktivitäten. Für jede Aktivitätsmethode in der Schnittstelle `@Activities` werden die folgenden Methodenüberladungen in der Client-Schnittstelle generiert:

1. Eine Überladung, welche die ursprünglichen Argumente unverändert übernimmt. Der Rückgabetyp dieser Überladung ist `Promise<`*T*`>`, wobei *T* der Rückgabetyp der ursprünglichen Methode ist. Zum Beispiel:

   Ursprüngliche Methode:

   ```
   void activity2(int foo);
   ```

   Generierte Methode:

   ```
   Promise<Void> activity2(int foo);
   ```

   Diese Überladung sollte verwendet werden, wenn alle Argumente des Workflows verfügbar sind und nicht auf diese gewartet werden muss.
2. Eine Überladung, welche die ursprünglichen Argumente unverändert, ein Argument der Art `ActivitySchedulingOptions` und ein zusätzliches variables Argument des Typs `Promise<?>` übernimmt. Der Rückgabetyp dieser Überladung ist `Promise<`*T*`>`, wobei *T* der Rückgabetyp der ursprünglichen Methode ist. Zum Beispiel:

   Ursprüngliche Methode:

   ```
   void activity2(int foo);
   ```

Generierte Methode:

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

Diese Überladung sollte verwendet werden, wenn alle Argumente des Workflows verfügbar sind und nicht auf diese gewartet werden muss, wenn Sie Standardeinstellungen überschreiben oder darauf warten, dass ein zusätzliches `Promises` betriebsbereit ist. Die variablen Argumente können verwendet werden, um zusätzliche Objekte wie `Promise<?>` weiterzuleiten, die nicht als Argumente deklariert waren, wenn Sie noch mit der Ausführung des Aufrufs warten möchten.

3. Eine Überladung mit jedem Argument in der ursprünglichen Methode, die durch einen `Promise<>`-Wrapper ersetzt wurde. Der Rückgabetyp dieser Überladung ist `Promise<`*T*`>`, wobei *T* der Rückgabetyp der ursprünglichen Methode ist. Zum Beispiel:

   Ursprüngliche Methode:

   ```
   void activity2(int foo);
   ```

   Generierte Methode:

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

   Diese Überladung sollte verwendet werden, wenn die Argumente, die an die Aktivität weitergeleitet werden sollen, asynchron evaluiert werden müssen. Ein Aufruf dieser Methodenüberladung wird so lange nicht ausgeführt, bis alle Argumente, die an diese weitergeleitet wurden, betriebsbereit sind.

4. Eine Überladung mit jedem Argument in der ursprünglichen Methode, die durch einen `Promise<>`-Wrapper ersetzt wurde. Die Überladung enthält auch ein zusätzliches Argument des Typs `ActivitySchedulingOptions` und variable Argumente des Typs `Promise<?>`. Der Rückgabetyp dieser Überladung ist `Promise<`*T*`>`, wobei *T* der Rückgabetyp der ursprünglichen Methode ist. Zum Beispiel:

   Ursprüngliche Methode:

```
void activity2(int foo);
```

Generierte Methode:

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

Diese Überladung sollte verwendet werden, wenn die Argumente, die an die Aktivität weitergeleitet werden sollen, asynchron evaluiert werden und wenn Sie Standardeinstellungen, die mit dem Typ registriert sind, überschreiben oder darauf warten möchten, dass zusätzliche `Promises` betriebsbereit sind. Ein Aufruf dieser Methodenüberladung wird so lange nicht ausgeführt, bis alle Argumente, die an diese weitergeleitet wurden, betriebsbereit sind. Die generierte Client-Klasse implementiert diese Schnittstelle. Die Implementierung jeder Schnittstellenmethode erstellt und sendet eine Anfrage an Amazon SWF, um mithilfe von Amazon SWF eine Aktivitätsaufgabe des entsprechenden Typs zu planen APIs.

5. Eine Überladung, die die ursprünglichen Argumente unverändert sowie zusätzliche variable Argumente des Typs `Promise<?>` übernimmt. Der Rückgabetyp dieser Überladung ist `Promise<`*T*`>`, wobei *T* der Rückgabetyp der ursprünglichen Methode ist. Zum Beispiel:

   Ursprüngliche Methode:

   ```
   void activity2(int foo);
   ```

   Generierte Methode:

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

   Diese Überladung sollte verwendet werden, wenn alle Argumente der Aktivität verfügbar sind und nicht auf diese gewartet werden muss, wenn Sie jedoch darauf warten möchten, dass andere `Promise`-Objekte betriebsbereit werden.

6. Eine Überladung, bei der jedes Argument in der ursprünglichen Methode durch einen `Promise`-Wrapper ersetzt wird und zusätzliche variable Argumente des Typs `Promise<?>` Der

Rückgabetyp dieser Überladung ist `Promise<T>`, wobei *T* der Rückgabetyp der ursprünglichen Methode ist. Zum Beispiel:

Ursprüngliche Methode:

```
void activity2(int foo);
```

Generierte Methode:

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

Diese Überladung sollte verwendet werden, wenn auf alle Argumente der Aktivität asynchron gewartet wird und wenn Sie darauf warten möchten, dass andere `Promises` betriebsbereit werden. Ein Aufruf dieser Methodenüberladung wird asynchron ausgeführt, wenn alle weitergeleiteten `Promise`-Objekte,betriebsbereit sind.

Der generierte Aktivitäts-Client verfügt auch über eine geschützte Methode, die jeder Aktivitätsmethode entspricht und den Namen {*activity method name*}`Impl()`, hat den alle Aktivitäts-Überladungen aufrufen. Sie können diese Methode überschreiben, um eine Demo-Clientimplementierung fertigzustellen. Diese Methode nimmt als Argumente an: alle Argumente der ursprünglichen Methode in `Promise<>`-Wrappers, `ActivitySchedulingOptions` und variable Argumente des Typs `Promise<?>`. Zum Beispiel:

Ursprüngliche Methode:

```
void activity2(int foo);
```

Generierte Methode:

```
Promise<Void> activity2Impl(
   Promise<Integer> foo,
   ActivitySchedulingOptions optionsOverride,
   Promise<?>...waitFor);
```

Planungsoptionen

Der generierte Aktivitäts-Client ermöglicht Ihnen, `ActivitySchedulingOptions` als Argument weiterzuleiten. Die `ActivitySchedulingOptions` Struktur enthält Einstellungen, die die Konfiguration der Aktivitätsaufgabe bestimmen, die das Framework in Amazon SWF plant. Diese Einstellungen überschreiben die Standardeinstellungen, die als Registrierungsoptionen festgelegt sind. Um Planungsoptionen dynamisch festzulegen, legen Sie ein `ActivitySchedulingOptions`-Objekt nach Ihren Wünschen an und übergeben es an die Aktivitätsmethode. Im folgenden Beispiel haben wir eine Aufgabe festgelegt, die für die Aktivitätsaufgabe verwendet werden soll. Sie überschreibt die registrierte Aufgabenliste für diesen Aktivitätsaufruf.

```
public class OrderProcessingWorkflowImpl implements OrderProcessingWorkflow {

    OrderProcessingActivitiesClient activitiesClient
            = new OrderProcessingActivitiesClientImpl();

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

        activitiesClient.shipOrder(order,
                                   schedulingOptions,
                                   paymentProcessed);
    }
}
```

Dynamische Clients

Zusätzlich zu den generierten Clients bietet das Framework auch Allzweck-Clients — `DynamicWorkflowClient` und `DynamicActivityClient` —, die Sie verwenden können, um Workflow-Ausführungen dynamisch zu starten, Signale zu senden, Aktivitäten zu planen usw. So möchten Sie z. B. eine Aktivität planen, deren Typ beim Design nicht bekannt war. Sie

können `DynamicActivityClient` zur Planung einer solchen Aktivitätsaufgabe verwenden. Ebenso können Sie eine untergeordnete Workflow-Ausführung dynamisch planen, indem Sie `DynamicWorkflowClient` verwenden. Im folgenden Beispiel schlägt der Workflow die Aktivität von einer Datenbank aus nach und verwendet zur Planung den Client für dynamische Aktivität:

```
//Workflow entrypoint
@Override
public void start() {
   MyActivitiesClient client = new MyActivitiesClientImpl();
  Promise<ActivityType> activityType
      = client.lookUpActivityFromDB();
  Promise<String> input = client.getInput(activityType);
  scheduleDynamicActivity(activityType,
                          input);
}
@Asynchronous
void scheduleDynamicActivity(Promise<ActivityType> type,
                             Promise<String> input){
  Promise<?>[] args = new Promise<?>[1];
   args[0] = input;
   DynamicActivitiesClient activityClient
       = new DynamicActivitiesClientImpl();
   activityClient.scheduleActivity(type.get(),
                                   args,
                                   null,
                                   Void.class);
}
```

Weitere Einzelheiten finden Sie in der AWS SDK für Java Dokumentation.

Signalisieren und Abbrechen von Workflow-Ausführungen

Der generierte Workflow-Client verfügt über Methoden, die jedem Signal entsprechen, das an den Workflow gesendet werden kann. Sie können diese aus einem Workflow heraus verwenden, um Signale an andere Workflow-Ausführungen zu senden. Dadurch wird ein typisierter Mechanismus zum Senden von Signalen bereitgestellt. Manchmal müssen Sie den Signalnamen jedoch möglicherweise dynamisch bestimmen, z. B. wenn der Signalname in einer Nachricht empfangen wird. Sie können den dynamischen Workflow-Client verwenden, um dynamisch Signale an eine beliebige Workflow-Ausführung zu senden. Auf ähnliche Weise können Sie den Client verwenden, um einen Abbruch einer anderen Workflow-Ausführung anzufordern.

Im folgenden Beispiel schlägt der Workflow die Ausführung zum Senden eines Signals von einer Datenbank aus nach und sendet das Signal dynamisch, wobei er den dynamischen Workflow-Client verwendet.

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

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

Workflow-Implementierung

Um einen Workflow zu implementieren, schreiben Sie eine Klasse, die die gewünschte `@Workflow`-Schnittstelle implementiert. Das Beispiel für die Workflow-Schnittstelle (`MyWorkflow`) kann wie folgt implementiert werden:

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

```
  @Override
  public void signal1(int a, int b, String c){
    //Process signal
     client.activity2(a + b);
  }
}
```

Die `@Execute`-Methode in dieser Klasse ist der Eintrittspunkt der Workflow-Logik. Da das Framework Replay verwendet, um den Objektstatus zu rekonstruieren, wenn eine Entscheidungsaufgabe verarbeitet werden soll, wird für jede Entscheidungsaufgabe ein neues Objekt erstellt.

Die Verwendung von `Promise<`*T*`>` als Parameter ist in der `@Execute`-Methode innerhalb einer `@Workflow`-Schnittstelle nicht erlaubt. Der Grund hierfür ist, dass das Ausführen eines asynchronen Aufrufs allein eine Entscheidung des Aufrufers ist. Die Workflow-Implementierung selbst hängt nicht davon ab, ob der Aufruf synchron oder asynchron erfolgt. Daher hat die generierte Client-Schnittstelle Überlastungen, die `Promise<`*T*`>`-Parameter akzeptieren, sodass diese Methoden asynchron aufgerufen werden können.

Der Rückgabetyp einer `@Execute`-Methode kann entweder `void` oder `Promise<`*T*`>` sein. Beachten Sie, dass ein Rückgabetyp des entsprechenden externen Clients `void` und nicht `Promise<>` ist. Da der externe Client nicht für die Verwendung im asynchronen Code vorgesehen ist, gibt der externe Client keine Objekte zurück. `Promise` Um extern festgelegte Ergebnisse von Workflow-Ausführungen zu erhalten, können Sie den Workflow so entwerfen, dass der Status in einem externen Datenspeicher durch eine Aktivität aktualisiert wird. Die Sichtbarkeit von Amazon SWF APIs kann auch verwendet werden, um das Ergebnis eines Workflows zu Diagnosezwecken abzurufen. Es wird nicht empfohlen, die Sichtbarkeit APIs zum Abrufen von Ergebnissen von Workflow-Ausführungen als allgemeine Praxis zu verwenden, da diese API-Aufrufe von Amazon SWF gedrosselt werden können. Um die Sichtbarkeit zu gewährleisten, APIs müssen Sie die Workflow-Ausführung anhand einer Struktur identifizieren. `WorkflowExecution` Diese Struktur können Sie vom generierten Workflow-Client durch Aufrufen der `getWorkflowExecution`-Methode abrufen. Diese Methode gibt die `WorkflowExecution`-Struktur zurück, die der Workflow-Ausführung entspricht, an die der Client gebunden ist. Weitere Informationen zur Sichtbarkeit finden Sie in der Amazon Simple Workflow Service API-Referenz APIs.

Beim Aufrufen von Aktivitäten aus Ihrer Workflow-Implementierung sollten Sie den generierten Aktivitäten-Client verwenden. Zum Senden von Signalen verwenden Sie entsprechend die generierten Workflow-Clients.

Entscheidungskontext

Das Framework stellt bei jeder Ausführung von Workflow-Code durch das Framework einen Umgebungskontext zur Verfügung. Dieser Kontext bietet kontextspezifische Funktionalität, auf die Sie in Ihrer Workflow-Implementierung zugreifen können, z. B. Erstellen eines Timers. Weitere Informationen finden Sie im Abschnitt Ausführungskontext.

Offenlegen des Ausführungsstatus

Amazon SWF ermöglicht es Ihnen, dem Workflow-Verlauf einen benutzerdefinierten Status hinzuzufügen. Der letzte Status, der von der Workflow-Ausführung gemeldet wurde, wird Ihnen durch Visibility-Aufrufe an den Amazon SWF-Service und in der Amazon SWF SWF-Konsole zurückgegeben. In einem Auftragsverarbeitungs-Workflow können Sie z. B. den Bestellungsstatus in verschiedenen Phasen melden, z. B. Bestellung erhalten, Bestellung versendet usw. In der AWS Flow Framework Version für Java wird dies durch eine Methode auf Ihrer Workflow-Oberfläche erreicht, die mit der `@GetState` Anmerkung versehen ist. Wenn der Entscheider die Verarbeitung einer Entscheidungsaufgabe abgeschlossen hat, wird diese Methode aufgerufen, um den aktuellen Status von der Workflow-Implementierung abzurufen. Neben Sichtbarkeitsaufrufen kann der Status auch mit dem generierten, externen Client abgerufen werden (der die Sichtbarkeits-API-Aufrufe intern verwendet).

Das folgende Beispiel zeigt, wie Sie den Ausführungskontext festlegen.

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

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

    @GetState
    String getState();
}

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

```
}

public class PeriodicWorkflowImpl implements PeriodicWorkflow {

    private DecisionContextProvider contextProvider
              = new DecisionContextProviderImpl();

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

    private PeriodicActivityClient activityClient
              = new PeriodicActivityClientImpl();

    private String state;

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

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

        // call activity
        activityClient.activity1();

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

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

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

Der generierte externe Client kann jederzeit zum Abrufen des aktuellen Status der Workflow-Ausführung verwendet werden.

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

Im obigen Beispiel wird der Ausführungsstatus in verschiedenen Stufen gemeldet. Wenn die Workflow-Instance startet, meldet `periodicWorkflow` den Anfangsstatus "Just Started" (Soeben gestartet). Mit jedem Aufruf an `callPeriodicActivity` wird der Workflow-Status anschließend aktualisiert. Sobald `activity1` 100 Mal aufgerufen wurde, wird die Methode zurückgegeben und die Workflow-Instance abgeschlossen.

Workflow-Lokale

Es kann vorkommen, das Sie statische Variablen in Ihrer Workflow-Implementierung verwenden müssen. Beispielsweise wenn Sie einen Zähler speichern möchten, auf den von verschiedenen Orten (möglicherweise unterschiedlichen Klassen) in der Implementierung des Workflows zugegriffen werden kann. Sie können jedoch nicht auf statische Variablen in Ihren Workflows vertrauen, da diese für verschiedene Threads freigegeben sind. Dies ist problematisch, da ein Worker möglicherweise verschiedene Entscheidungsaufgaben in unterschiedlichen Threads verarbeitet. Alternativ können Sie einen solchen Status in einem Feld in der Workflow-Implementierung speichern. In diesem Fall müssen Sie allerdings das Implementierungsobjekt weitergeben. Für diesen Fall stellt das Framework eine `WorkflowExecutionLocal<?>`-Klasse zur Verfügung. Jeder Status, der eine statische Variable wie Semantik erfordert, sollte als Instance-Lokal unter Verwendung von `WorkflowExecutionLocal<?>` gespeichert werden. Sie können eine statische Variable dieses Typs deklarieren und verwenden. Im folgenden Ausschnitt wird eine `WorkflowExecutionLocal<String>` zum Speichern eines Benutzernamens verwendet.

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

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

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

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

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

Implementierung von Aktivitäten

Die Aktivitäten werden durch eine Implementierung der `@Activities`-Schnittstelle realisiert. AWS Flow Framework for Java verwendet die auf dem Worker konfigurierten Implementierungsinstanzen für Aktivitäten, um Aktivitätsaufgaben zur Laufzeit zu verarbeiten. Der Worker sucht automatisch die Aktivitätsimplementierung des entsprechenden Typs.

Über Eigenschaften und Felder können Sie Ressourcen an Aktivitäts-Instances wie z. B. Datenbankverbindungen übergeben. Da auf das Objekt zur Implementierung der Aktivität von

mehreren Threads aus zugegriffen werden kann, müssen gemeinsam genutzte Ressourcen threadsicher sein.

Beachten Sie, dass die Aktivitätsimplementierung keine Parameter vom Typ `Promise<>` oder Rückgabeobjekte dieses Typs akzeptiert. Die Implementierung der Aktivität sollte nicht davon abhängen, wie sie aufgerufen wurde (synchron oder asynchron).

Die zuvor gezeigte Aktivitätsschnittstelle kann folgendermaßen implementiert werden:

```
public class MyActivitiesImpl implements MyActivities {

   @Override
   @ManualActivityCompletion
   public int activity1(){
      //implementation
   }

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

Der Aktivitätsimplementierung steht ein Thread-lokaler Kontext zur Verfügung, über den das verwendete Aufgabenobjekt, Datenkonverter-Objekt etc. abgerufen werden kann. Auf den aktuellen Kontext kann über `ActivityExecutionContextProvider.getActivityExecutionContext()` zugegriffen werden. Weitere Informationen finden Sie in der AWS SDK für Java Dokumentation `ActivityExecutionContext` und im Abschnitt Ausführungskontext.

Aktivitäten manuell abschließen

Die `@ManualActivityCompletion`-Annotation im obigen Beispiel ist eine optionale Annotation. Sie ist nur bei Methoden erlaubt, die eine Aktivität implementieren. Sie wird verwendet, um die Aktivität so zu konfigurieren, dass sie bei der Rückkehr aus der Aktivitätsmethode nicht automatisch beendet wird. Dies kann nützlich sein, wenn Sie die Aktivität asynchron abschließen möchten, z. B. manuell, nachdem eine menschliche Aktion abgeschlossen wurde.

Standardmäßig sieht das Framework die Aktivität als abgeschlossen an, sobald Ihre Aktivitätsmethode die Kontrolle zurückgibt. Das bedeutet, dass der Activity Worker Amazon SWF den Abschluss der Aktivitätsaufgabe meldet und ihm die Ergebnisse (falls vorhanden)

zur Verfügung stellt. Es gibt jedoch Anwendungsfälle, in denen die Aktivitätsaufgabe bei der Rückkehr aus der Aktivitätsmethode nicht als abgeschlossen gekennzeichnet werden soll. Dies ist besonders hilfreich, wenn Sie menschliche Aufgaben modellieren. Beispielsweise kann die Aktivitätsmethode eine E-Mail an eine Person senden, die eine Aufgabe erledigen muss, bevor die Aktivitätsaufgabe abgeschlossen ist. In solchen Fällen können Sie die Aktivitätsmethode mit der `@ManualActivityCompletion`-Annotation definieren, um dem Aktivitäts-Worker mitzuteilen, dass er die Aktivität nicht automatisch abschließen soll. Um die Aktivität manuell abzuschließen, können Sie entweder die im Framework `ManualActivityCompletionClient` bereitgestellte Methode oder die `RespondActivityTaskCompleted` Methode auf dem Amazon SWF-Java-Client verwenden, die im Amazon SWF SDK bereitgestellt wird. Weitere Informationen finden Sie in der AWS SDK für Java Dokumentation.

Um die Aktivitätsaufgabe abzuschließen, müssen Sie ein Aufgaben-Token bereitstellen. Das Aufgaben-Token wird von Amazon SWF verwendet, um Aufgaben eindeutig zu identifizieren. In Ihrer Aktivitätsimplementierung können Sie über `ActivityExecutionContext` auf das Token zugreifen. Sie müssen dieses Token an denjenigen übergeben, der für die Erledigung der Aufgabe verantwortlich ist. Das Token kann durch den Aufruf von `ActivityExecutionContextProvider.getActivityExecutionContext().getTaskToken()` von `ActivityExecutionContext` abgerufen werden.

Die `getName`-Aktivität des HelloWorld-Beispiels kann implementiert werden, um eine E-Mail mit der Bitte um eine Begrüßungsnachricht an jemanden zu senden:

```
@ManualActivityCompletion
@Override
public String getName() throws InterruptedException {
    ActivityExecutionContext executionContext
         = contextProvider.getActivityExecutionContext();
    String taskToken = executionContext.getTaskToken();
    sendEmail("abc@xyz.com",
         "Please provide a name for the greeting message and close task with token: " +
 taskToken);
    return "This will not be returned to the caller";
}
```

Der folgende Code kann verwendet werden, um die Begrüßung bereitzustellen und die Aufgabe mit `ManualActivityCompletionClient` zu schließen. Alternativ können Sie die Aufgabe auch fehlschlagen lassen:

```
public class CompleteActivityTask {
```

```
    public void completeGetNameActivity(String taskToken) {

        AmazonSimpleWorkflow swfClient
           = new AmazonSimpleWorkflowClient(...); // use AWS access keys
        ManualActivityCompletionClientFactory manualCompletionClientFactory
           = new ManualActivityCompletionClientFactoryImpl(swfClient);
        ManualActivityCompletionClient manualCompletionClient
           = manualCompletionClientFactory.getClient(taskToken);
        String result = "Hello World!";
        manualCompletionClient.complete(result);
    }

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

AWS Lambda Aufgaben umsetzen

Themen

- Über AWS Lambda
- Vorteile und Einschränkungen der Verwendung von Lambda-Aufgaben
- Verwenden von Lambda-Aufgaben in Ihren AWS Flow Framework Workflows für Java
- Sehen Sie sich das Beispiel an HelloLambda

Über AWS Lambda

AWS Lambda ist ein vollständig verwalteter Rechenservice, der Ihren Code als Reaktion auf Ereignisse ausführt, die durch benutzerdefinierten Code oder durch verschiedene AWS Dienste wie Amazon S3, DynamoDB, Amazon Kinesis, Amazon SNS und Amazon Cognito generiert wurden. Weitere Informationen zu Lambda finden Sie im AWS Lambda Entwicklerhandbuch.

Amazon Simple Workflow Service bietet eine Lambda-Aufgabe, sodass Sie Lambda-Funktionen anstelle von oder zusammen mit herkömmlichen Amazon SWF SWF-Aktivitäten ausführen können.

> **Important**
>
> Ihr AWS Konto wird für Lambda-Ausführungen (Anfragen) belastet, die von Amazon SWF in Ihrem Namen ausgeführt werden. Einzelheiten zu den Lambda-Preisen finden Sie unter https://aws.amazon.com/lambda/pricing/.

Vorteile und Einschränkungen der Verwendung von Lambda-Aufgaben

Die Verwendung von Lambda-Aufgaben anstelle einer herkömmlichen Amazon SWF SWF-Aktivität bietet eine Reihe von Vorteilen:

- Lambda-Aufgaben müssen nicht wie Amazon SWF SWF-Aktivitätstypen registriert oder versioniert werden.
- Sie können alle vorhandenen Lambda-Funktionen verwenden, die Sie bereits in Ihren Workflows definiert haben.
- Lambda-Funktionen werden direkt von Amazon SWF aufgerufen. Sie müssen kein Worker-Programm implementieren, um sie auszuführen, wie dies bei herkömmlichen Aktivitäten der Fall ist.
- Lambda stellt Ihnen Metriken und Protokolle zur Verfügung, mit denen Sie Ihre Funktionsausführungen verfolgen und analysieren können.

Bei Lambda-Aufgaben sind jedoch einige Einschränkungen zu beachten:

- Lambda-Aufgaben können nur in AWS Regionen ausgeführt werden, die Lambda unterstützen. Einzelheiten zu den derzeit unterstützten Regionen für Lambda finden Sie unter Lambda Regions and Endpoints in der Amazon Web Services General Reference.
- Lambda-Aufgaben werden derzeit nur von der SWF-Basis-SWF-HTTP-API und in der AWS Flow Framework für Java unterstützt. Derzeit gibt es keine Unterstützung für Lambda-Aufgaben in der AWS Flow Framework für Ruby.

Verwenden von Lambda-Aufgaben in Ihren AWS Flow Framework Workflows für Java

Für die Verwendung von Lambda-Aufgaben in Ihren Workflows AWS Flow Framework für Java gelten drei Voraussetzungen:

- Eine Lambda Lambda-Funktion. Sie können jede Lambda-Funktion verwenden, die Sie definiert haben. Weitere Informationen zum Erstellen von Lambda-Funktionen finden Sie im AWS Lambda Entwicklerhandbuch.
- Eine IAM-Rolle, die Zugriff auf die Ausführung von Lambda-Funktionen aus Ihren Amazon SWF SWF-Workflows bietet.
- Code zum Planen der Lambda-Aufgabe in Ihrem Workflow.

Einrichten einer IAM-Rolle

Bevor Sie Lambda-Funktionen von Amazon SWF aufrufen können, müssen Sie eine IAM-Rolle bereitstellen, die den Zugriff auf Lambda von Amazon SWF aus ermöglicht. Führen Sie dazu einen der folgenden Schritte aus:

- Wählen Sie eine vordefinierte Rolle, AWSLambdaRolle, um Ihren Workflows die Erlaubnis zu geben, alle Lambda-Funktionen aufzurufen, die mit Ihrem Konto verknüpft sind.
- Definieren Sie Ihre eigene Richtlinie und die zugehörige Rolle, um Workflows die Erlaubnis zu erteilen, bestimmte Lambda-Funktionen aufzurufen, die durch ihre Amazon-Ressourcennamen () ARNs spezifiziert sind.

Beschränken Sie die Berechtigungen für eine IAM-Rolle

Sie können die Berechtigungen für eine IAM-Rolle, die Sie Amazon SWF zur Verfügung stellen, einschränken, indem Sie die `SourceAccount` Kontextschlüssel `SourceArn` und in Ihrer Resource Trust Policy verwenden. Diese Schlüssel schränken die Verwendung einer IAM-Richtlinie ein, sodass sie nur für Amazon Simple Workflow Service-Ausführungen verwendet wird, die zum angegebenen Domain-ARN gehören. Wenn Sie beide Kontextschlüssel für globale Bedingungen verwenden, müssen der `aws:SourceAccount` Wert und das Konto, auf das im `aws:SourceArn` Wert verwiesen wird, dieselbe Konto-ID verwenden, wenn sie in derselben Richtlinienerklärung verwendet werden.

Im folgenden Beispiel für eine Vertrauensrichtlinie verwenden wir den `SourceArn` Kontextschlüssel, um die IAM-Servicerolle so zu beschränken, dass sie nur in Amazon Simple Workflow Service-Ausführungen verwendet wird, die zu `someDomain` dem Konto gehören,. `123456789012`

```
{
  "Version": "2012-10-17",
  "Statement": [
     {
       "Sid": "",
       "Effect": "Allow",
       "Principal": {
         "Service": "swf.amazonaws.com"
       },
       "Action": "sts:AssumeRole",
       "Condition": {
         "ArnLike": {
           "aws:SourceArn": "arn:aws:swf:*:123456789012:/domain/someDomain"
       }
      }
    }
  ]
}
```

Im folgenden Beispiel für eine Vertrauensrichtlinie verwenden wir den `SourceAccount` Kontextschlüssel, um die IAM-Servicerolle so zu beschränken, dass sie nur in Amazon Simple Workflow Service-Ausführungen im Konto, verwendet wird. `123456789012`

```
{
  "Version": "2012-10-17",
  "Statement": [
     {
       "Sid": "",
       "Effect": "Allow",
       "Principal": {
         "Service": "swf.amazonaws.com"
       },
       "Action": "sts:AssumeRole",
       "Condition": {
         "StringLike": {
           "aws:SourceAccount": "123456789012"
       }
      }
```

```
    }
  ]
}
```

Amazon SWF Zugriff zum Aufrufen beliebiger Lambda-Rollen gewähren

Sie können die vordefinierte Rolle Rolle verwenden, AWSLambda um Ihren Amazon SWF SWF-Workflows die Möglichkeit zu geben, jede Lambda-Funktion aufzurufen, die mit Ihrem Konto verknüpft ist.

So verwenden Sie AWSLambda Role, um Amazon SWF Zugriff zum Aufrufen von Lambda-Funktionen zu gewähren

1. Öffnen Sie die Amazon IAM-Konsole.
2. Wählen Sie Roles und anschließend Create New Role aus.
3. Geben Sie einen Namen für die Rolle ein, z. B. `swf-lambda`, und klicken Sie auf Next Step.
4. Wählen Sie unter AWS Service Roles Amazon SWF und dann Next Step aus.
5. Wählen Sie auf dem Bildschirm „Richtlinie anhängen“ die Option AWSLambdaRolle aus der Liste aus.
6. Klicken Sie auf Next Step und auf Create Role, sobald Sie die Rolle überprüft haben.

Definition einer IAM-Rolle für den Zugriff auf den Aufruf einer bestimmten Lambda-Funktion

Wenn Sie Zugriff zum Aufrufen einer bestimmten Lambda-Funktion aus Ihrem Workflow gewähren möchten, müssen Sie Ihre eigene IAM-Richtlinie definieren.

So erstellen Sie eine IAM-Richtlinie für den Zugriff auf eine bestimmte Lambda-Funktion

1. Öffnen Sie die Amazon IAM-Konsole.
2. Wählen Sie Policies und dann Create Policy aus.
3. Wählen Sie „ AWS Verwaltete Richtlinie kopieren“ und wählen Sie „AWSLambdaRolle“ aus der Liste aus. Es wird eine Richtlinie erstellt. Sie können ihren Namen und die Beschreibung nach Bedarf ändern.
4. Fügen Sie im Feld Ressource des Richtliniendokuments den ARN Ihrer Lambda-Funktion (en) hinzu. Zum Beispiel:

```
{
```

```
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": [
        "arn:aws:lambda:us-east-1:111111000000:function:hello_lambda_function"
      ]
    }
  ]
}
```

> **Note**
>
> Eine vollständige Beschreibung der Angabe von Ressourcen in einer IAM-Rolle finden Sie unter Überblick über IAM-Richtlinien in Using IAM.

5. Wählen Sie Create policy aus, um Ihre Richtlinie zu erstellen.

Sie können diese Richtlinie dann auswählen, wenn Sie eine neue IAM-Rolle erstellen, und diese Rolle verwenden, um Aufrufzugriff auf Ihre Amazon SWF SWF-Workflows zu gewähren. Dieses Verfahren ist dem Erstellen einer Rolle mit der Rollenrichtlinie sehr ähnlich. Wählen Sie stattdessen Ihre eigene Richtlinie, wenn Sie die AWSLambdaRolle erstellen.

So erstellen Sie eine Amazon SWF SWF-Rolle mithilfe Ihrer Lambda-Richtlinie

1. Öffnen Sie die Amazon IAM-Konsole.
2. Wählen Sie Roles und anschließend Create New Role aus.
3. Geben Sie einen Namen für die Rolle ein, z. B. `swf-lambda-function`, und klicken Sie auf Next Step.
4. Wählen Sie unter AWS Service Roles Amazon SWF und dann Next Step aus.
5. Wählen Sie auf dem Bildschirm Attach Policy Ihre funktionsspezifische Lambda-Richtlinie aus der Liste aus.
6. Klicken Sie auf Next Step und auf Create Role, sobald Sie die Rolle überprüft haben.

Eine Lambda-Aufgabe für die Ausführung planen

Sobald Sie eine IAM-Rolle definiert haben, mit der Sie Lambda-Funktionen aufrufen können, können Sie deren Ausführung als Teil Ihres Workflows planen.

> **Note**
>
> Dieser Prozess wird anhand des HelloLambda Beispiels in der vollständig demonstriert. AWS SDK für Java

So planen Sie die Ausführung einer Lambda-Task

1. Rufen Sie in Ihrer Workflow-Implementierung eine Instance des `LambdaFunctionClient` ab, indem Sie `getLambdaFunctionClient()` für eine `DecisionContext`-Instance aufrufen.

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

2. Planen Sie die Aufgabe mithilfe der `scheduleLambdaFunction()` Methode auf der `LambdaFunctionClient` und übergeben Sie ihr den Namen der Lambda-Funktion, die Sie erstellt haben, sowie alle Eingabedaten für die Lambda-Aufgabe.

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

lambdaClient.scheduleLambdaFunction(lambda_function_name, lambda_function_input);
```

3. Fügen Sie in Ihrem Workflow-Ausführungsstarter die IAM-Lambda-Rolle zu Ihren Standard-Workflow-Optionen hinzu, indem Sie die Optionen verwenden`StartWorkflowOptions.withLambdaRole()`, und übergeben Sie sie dann, wenn Sie den Workflow starten.

```
// Workflow client classes are generated for you when you use the @Workflow
// annotation on your workflow interface declaration.
```

```
MyWorkflowClientExternalFactory clientFactory =
  new MyWorkflowClientExternalFactoryImpl(sdk_swf_client, swf_domain);

MyWorkflowClientExternal workflow_client = clientFactory.getClient();

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

StartWorkflowOptions workflow_options =
  new StartWorkflowOptions().withLambdaRole(lambda_iam_role);

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

Sehen Sie sich das Beispiel an HelloLambda

Ein Beispiel, das eine Implementierung eines Workflows bietet, der eine Lambda-Aufgabe verwendet, finden Sie in der AWS SDK für Java. Laden Sie die Quelle herunter, um es anzuzeigen und/oder auszuführen.

Eine vollständige Beschreibung der Erstellung und Ausführung des HelloLambdaBeispiels finden Sie in der README-Datei, die den Java-Beispielen AWS Flow Framework beiliegt.

Ausführen von Programmen, die mit dem AWS Flow Framework für Java geschrieben wurden

Themen

- WorkflowWorker
- ActivityWorker
- Worker-Threading-Modell
- Worker-Erweiterbarkeit

Das Framework stellt Worker-Klassen zur Initialisierung der Runtime AWS Flow Framework für Java und zur Kommunikation mit Amazon SWF bereit. Um einen Workflow- oder Aktivitäts-Worker zu implementieren, müssen Sie zuerst eine Instance einer Worker-Klasse erstellen und starten. Diese

Worker-Klassen sind für die Verwaltung laufender asynchroner Vorgänge, das Aufrufen asynchroner Methoden, die entsperrt werden, und für die Kommunikation mit Amazon SWF verantwortlich. Sie können mit Workflow- und Aktivitätsimplementierungen, der Anzahl an Threads, der abzufragenden Aufgabenliste usw. konfiguriert werden.

Das Framework enthält zwei Worker-Klassen, eine für Aktivitäten und eine für Workflows. Zum Ausführen der Workflow-Logik verwenden Sie die `WorkflowWorker`-Klasse. Analog verwenden Sie für Aktivitäten die `ActivityWorker`-Klasse. Diese Klassen fragen Amazon SWF automatisch nach Aktivitätsaufgaben ab und rufen die entsprechenden Methoden in Ihrer Implementierung auf.

Im folgenden Beispiel wird gezeigt, wie ein `WorkflowWorker` instanziiert wird und Aufgaben abgerufen werden.

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

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

Im Folgenden sehen Sie die grundlegenden Schritte zum Erstellen einer Instance von `ActivityWorker` und dem Abrufen von Aufgaben:

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

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

Wenn Sie eine Aktivität oder einen Entscheider beenden möchten, sollte Ihre Anwendung die Instances der verwendeten Worker-Klassen sowie die Amazon SWF SWF-Java-Client-Instance herunterfahren. So können Sie sicher sein, dass alle Ressourcen, die von den Worker-Klassen verwendet werden, ordnungsgemäß freigegeben werden.

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

Um mit einer Ausführung zu beginnen, erstellen Sie einfach eine Instance des generierten externen Client und rufen Sie die `@Execute`-Methode auf.

```
MyWorkflowClientExternalFactory factory = new MyWorkflowClientExternalFactoryImpl();
MyWorkflowClientExternal client = factory.getClient();
client.start();
```

WorkflowWorker

Wie der Name schon sagt, dient diese Worker-Klasse zur Verwendung mit der Workflow-Implementierung. Sie wird mit einer Aufgabenliste und dem Workflow-Implementierungstyp konfiguriert. Die Worker-Klasse führt eine Schleife zur Abfrage von Entscheidungsaufgaben in der angegebenen Aufgabenliste aus. Wenn eine Entscheidungsaufgabe empfangen wird, erstellt sie eine Instance der Workflow-Implementierung und ruft die `@Execute`-Methode zur Verarbeitung der Aufgabe auf.

ActivityWorker

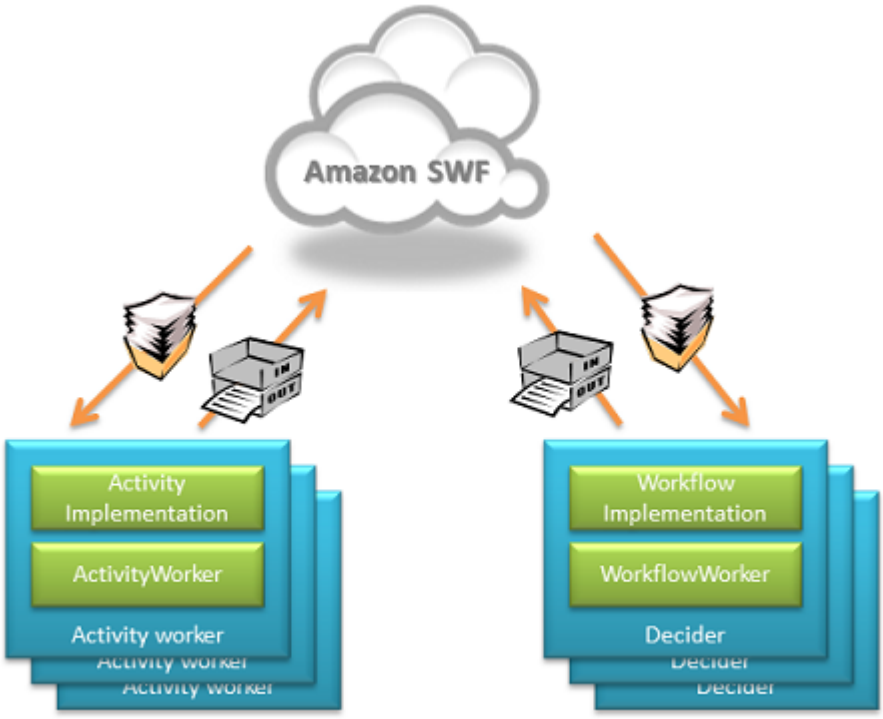
Zur Implementierung von Aktivitäts-Workern können Sie die `ActivityWorker`-Klasse verwenden und einfach eine Aufgabenliste nach Aktivitätsaufgaben abfragen. Sie konfigurieren den Aktivitäts-Worker mit Aktivitäts-Implementierungsobjekten. Diese Worker-Klasse führt eine Schleife zur Abfrage von Aktivitätsaufgaben in der angegebenen Aufgabenliste aus. Wenn eine Aktivitätssaufgabe empfangen wird, sucht sie die geeignete von Ihnen bereitgestellte Implementierung und ruft die Aktivitätsmethode zur Verarbeitung der Aufgabe auf. Im Gegensatz zur Worker-Klasse `WorkflowWorker`, die die Factory aufruft, um für jede Entscheidungsaufgabe eine neue Instance zu erstellen, verwendet `ActivityWorker` nur das von Ihnen bereitgestellte Objekt.

Die `ActivityWorker` Klasse verwendet die Anmerkungen AWS Flow Framework für Java, um die Registrierungs- und Ausführungsoptionen zu bestimmen.

Worker-Threading-Modell

AWS Flow Framework Bei Java ist die Verkörperung einer Aktivität oder eines Entscheiders eine Instanz der Arbeiterklasse. Ihre Anwendung ist verantwortlich für die Konfiguration und die

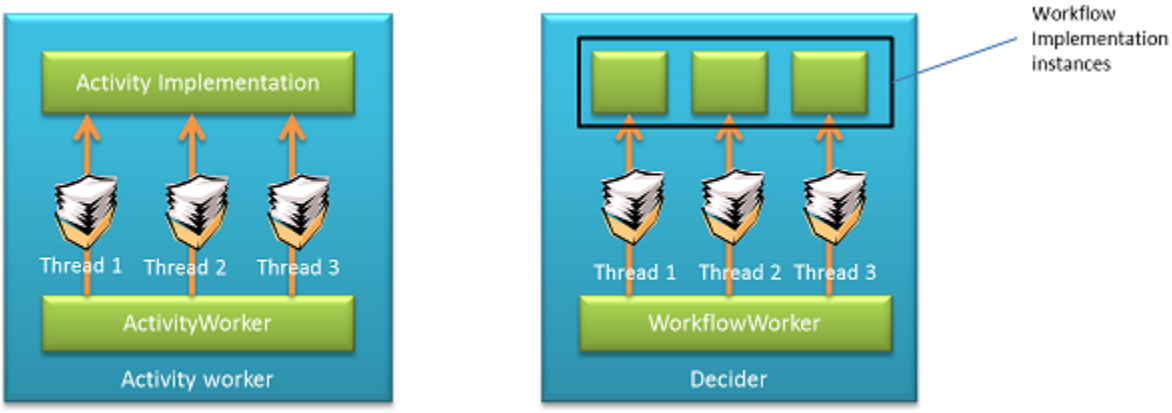
Instanziierung des Worker-Objekts auf jedem Computer und Prozess, der als Worker eingesetzt wird. Das Worker-Objekt empfängt dann automatisch Aufgaben von Amazon SWF, leitet sie an Ihre Aktivitäts- oder Workflow-Implementierung weiter und meldet die Ergebnisse an Amazon SWF. Es ist möglich, dass eine einzige Workflow-Instance viele Worker umfasst. Wenn Amazon SWF eine oder mehrere ausstehende Aktivitätsaufgaben hat, weist es dem ersten verfügbaren Mitarbeiter eine Aufgabe zu, dann dem nächsten usw. So können Aufgaben, die zur selben Workflow-Instance gehören, gleichzeitig in unterschiedlichen Workern verarbeitet werden.



Zusätzlich kann jeder Worker so konfiguriert werden, dass er Aufgaben in mehreren Threads verarbeitet. Das bedeutet, dass die Aktivitätsaufgaben einer Workflow-Instance gleichzeitig ausgeführt werden können, selbst wenn nur ein Worker zur Verfügung steht.

Entscheidungsaufgaben verhalten sich ähnlich, mit der Ausnahme, dass Amazon SWF garantiert, dass für eine bestimmte Workflow-Ausführung jeweils nur eine Entscheidung ausgeführt werden kann. Eine einzelne Workflow-Ausführung erfordert im Allgemeinen mehrere Entscheidungsaufgaben. Deshalb kommt es oft zu Ausführungen in mehreren Prozessen und Threads. Der Entscheider wird mit dem Workflow-Implementierungstyp konfiguriert. Wenn eine Entscheidungsaufgabe vom Entscheider empfangen wird, erstellt er eine Instance (ein Objekt) der Workflow-Implementierung. Das Framework stellt ein erweiterbares Factory-Muster für die Erstellung dieser Instances bereit. Die standardmäßige Workflow-Factory erstellt jedes Mal ein neues Objekt. Um dieses Verhalten zu umgehen, können Sie benutzerdefinierte Factories bereitstellen.

Im Gegensatz zu Entscheidern, die mit Workflow-Implementierungstypen konfiguriert werden, werden Aktivitäts-Worker mit Instances (Objekten) der Aktivitätsimplementierungen konfiguriert. Wenn eine Aktivitätssaufgabe vom Aktivitäts-Worker empfangen wird, wird sie an das geeignete Implementierungsobjekt der Aktivität gesendet.



Der Workflow-Worker verwaltet einen einzigen Thread-Pool und führt den Workflow auf demselben Thread aus, der für die Abfrage von Amazon SWF für die Aufgabe verwendet wurde. Da Aktivitäten lange dauern (zumindest im Vergleich zur Workflow-Logik), verwaltet die Activity Worker-Klasse zwei separate Thread-Pools: einen für die Abfrage von Amazon SWF nach Aktivitätsaufgaben und den anderen für die Verarbeitung von Aufgaben durch Ausführung der Aktivitätsimplementierung. So können Sie die Anzahl der Threads zum Abrufen von Aufgaben separat von der Anzahl der Threads konfigurieren, die sie ausführen. Beispielsweise kann eine kleine Anzahl an Threads zum Abrufen verfügbar sein und eine große Anzahl für die Ausführung der Aufgaben. Die Activity Worker-Klasse fragt Amazon SWF nur dann nach einer Aufgabe ab, wenn sie über einen freien Abfrage-Thread sowie einen freien Thread zur Bearbeitung der Aufgabe verfügt.

Dieses Threading- und Instancing-Verhalten zeigt Folgendes:

1. Aktivitätsimplementierungen müssen zustandslos sein. Sie sollten Instanzvariablen nicht dazu verwenden, den Anwendungszustand in Aktivitätsobjekten zu speichern. Über Felder können Sie jedoch Ressourcen wie Datenbankverbindungen speichern.
2. Aktivitätsimplementierungen müssen threadsicher sein. Da dieselbe Instanz verwendet werden kann, um Aufgaben aus verschiedenen Threads gleichzeitig zu verarbeiten, muss der Zugriff auf gemeinsam genutzte Ressourcen aus dem Aktivitätscode synchronisiert werden.
3. Die Workflow-Implementierung kann zustandsbehaftet sein und Instance-Variablen können zum Speichern des Status verwendet werden. Auch wenn eine neue Instance der Workflow-Implementierung erstellt wurde, um jede Entscheidungsaufgabe zu verarbeiten, stellt das Framework sicher, dass der Status ordnungsgemäß wiederhergestellt wird. Allerdings muss die

Implementierung Ihres Workflows deterministisch sein. Weitere Details finden Sie im Abschnitt Eine Aufgabe in AWS Flow Framework für Java verstehen.

4. Workflow-Implementierungen müssen nicht threadsicher sein, wenn die Standard-Factory verwendet wird. Durch die Standardimplementierung wird sichergestellt, dass nur ein Thread gleichzeitig eine Instanz der Implementierung Ihres Workflows verwendet.

Worker-Erweiterbarkeit

Die AWS Flow Framework für Java enthält auch einige Low-Level-Worker-Klassen, die Ihnen eine detaillierte Steuerung und Erweiterbarkeit bieten. Damit können Sie die Registrierung vom Workflow- und Aktivitäts-Typ genau anpassen und Factories für die Erstellung von Implementierungsobjekten bestimmen. Diese Worker sind `GenericWorkflowWorker` und `GenericActivityWorker`.

`GenericWorkflowWorker` kann mit einer Factory zur Erstellung von Factories für Workflow-Definitionen konfiguriert werden. Die Factory für Workflow-Definitionen ist verantwortlich für die Erstellung von Instances der Workflow-Implementierung und für die Bereitstellung von Konfigurationseinstellungen wie den Registrierungsoptionen. Unter normalen Umständen sollten Sie die `WorkflowWorker`-Klasse direkt verwenden. Sie erstellt und konfiguriert die Implementierung der bereitgestellten Factories in das Framework, `POJOWorkflowDefinitionFactoryFactory` und `POJOWorkflowDefinitionFactory`. Die Factory setzt voraus, dass die Workflow-Implementierungsklasse über einen Konstruktor verfügt, der keine Argumente annimmt. Dieser Konstruktor wird verwendet, um Instances des Workflow-Objekts zur Laufzeit zu erstellen. Die Factory prüft die Anmerkungen, die Sie in der Workflow-Schnittstelle und der Implementierung verwendet haben, um geeignete Registrierungs- und Ausführungsoptionen zu erstellen.

Sie können eine eigene Implementierung der Factories bereitstellen, indem Sie `WorkflowDefinitionFactory`, `WorkflowDefinitionFactoryFactory` und `WorkflowDefinition` implementieren. Die `WorkflowDefinition`-Klasse wird von der Worker-Klasse dazu verwendet, Entscheidungsaufgaben und Signale zu versenden. Wenn Sie diese Basisklassen implementieren, können Sie die Factory und die Verteilung von Anfragen an die Workflow-Implementierung genau anpassen. Sie können diese Erweiterbarkeitspunkte dazu verwenden, ein benutzerdefiniertes Programmierungsmodell zum Schreiben von Workflows bereitzustellen, z. B. basierend auf Ihren eigenen Anmerkungen oder durch Generieren aus WSDL – anstelle des Code-First-Ansatzes, der vom Framework verwendet wird. Um Ihre benutzerdefinierten Factories nutzen zu können, müssen Sie die `GenericWorkflowWorker`-Klasse verwenden. Weitere Informationen zu diesen Klassen finden Sie in der Dokumentation. AWS SDK für Java

In ähnlicher Weise bietet auch `GenericActivityWorker` die Möglichkeit, eine benutzerdefinierte Factory für Aktivitätsimplementierungen bereitzustellen. Wenn Sie die Klassen `ActivityImplementationFactory` und `ActivityImplementation` implementieren, können Sie die Instanziierung komplett steuern und die Registrierungs- und Ausführungsoptionen selbst definieren. Weitere Informationen zu diesen Klassen finden Sie in der AWS SDK für Java Dokumentation.

Ausführungskontext

Themen

- Entscheidungskontext
- Aktivitätsausführungskontext

Das Framework gibt dem Workflow und den Aktivitätsimplementierungen einen Umgebungskontext. Dieser Kontext bezieht sich jeweils auf die ausgeführte Aufgabe und stellt einige Dienstprogramme bereit, die Sie in der Implementierung verwenden können. Ein Kontextobjekt wird jedes Mal erstellt, wenn eine neue Aufgabe vom Auftragnehmer verarbeitet wird.

Entscheidungskontext

Wenn eine Entscheidungsaufgabe ausgeführt wird, stellt das Framework den Kontext für die Workflow-Implementierung über die `DecisionContext`-Klasse zur Verfügung. `DecisionContext` liefert kontextsensitive Informationen wie die ID des Workflow-Ausführungslaufs und die Takt- und Timerfunktionalität.

Zugriff DecisionContext bei der Workflow-Implementierung

Sie können auf den `DecisionContext` in Ihrer Workflow-Implementierung unter Verwendung der `DecisionContextProviderImpl`-Klasse zugreifen. Alternativ können Sie den Kontext in einem Feld oder einer Eigenschaft Ihrer Workflow-Implementierung angeben. Verwenden Sie dazu Spring, wie im Abschnitt "Prüfbarkeit und Dependency Injection" beschrieben.

```
DecisionContextProvider contextProvider
    = new DecisionContextProviderImpl();
DecisionContext context = contextProvider.getDecisionContext();
```

Erstellen einer Uhr und eines Timers

Der `DecisionContext` enthält eine Eigenschaft vom Typ `WorkflowClock`, die eine Timer- und Uhrfunktion bereitstellt. Da die Workflow-Logik deterministisch sein muss, sollten Sie die Systemuhr in Ihrer Workflow-Implementierung nicht direkt verwenden. Die `currentTimeMills`-Methode in der `WorkflowClock` gibt den Zeitpunkt des Startereignisses der zu verarbeitenden Entscheidung zurück. So wird sichergestellt, dass Sie denselben Zeitwert bei einer Wiedergabe erhalten und eine deterministische Workflow-Logik erhalten.

`WorkflowClock` umfasst auch eine `createTimer`-Methode, die ein `Promise`-Objekt zurückgibt, das nach einem festgelegten Intervall verfügbar wird. Verwenden Sie diesen Wert als Parameter für andere asynchrone Methoden, um deren Ausführung um einen festgelegten Zeitraum zu verschieben. So können Sie eine asynchrone Methode oder Aktivität effektiv für eine spätere Ausführung planen.

Im folgenden Beispiel wird gezeigt, wie Sie eine Aktivität periodisch aufrufen können.

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

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

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

public class PeriodicWorkflowImpl implements PeriodicWorkflow {

    private DecisionContextProvider contextProvider
         = new DecisionContextProviderImpl();

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

    @Override
```

```
    public void periodicWorkflow() {
        callPeriodicActivity(0);
    }

    @Asynchronous
    private void callPeriodicActivity(int count,
                                      Promise<?>... waitFor) {
        if (count == 100) {
            return;
        }
        PeriodicActivityClient client = new PeriodicActivityClientImpl();
        // call activity
        Promise<Void> activityCompletion = client.activity1();

        Promise<Void> timer = clock.createTimer(3600);

        // Repeat the activity either after 1 hour or after previous activity run
        // if it takes longer than 1 hour
        callPeriodicActivity(count + 1, timer, activityCompletion);
    }
}


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

In der Liste oben ruft die asynchrone Methode `callPeriodicActivity` `activity1` auf und erstellt einen Timer mit dem aktuellen `AsyncDecisionContext`. Sie übergibt das zurückgegebene `Promise` als Argument an einen rekursiven Aufruf von sich selbst. Dieser rekursive Aufruf wartet, bis der Timer ausgelöst wird (in diesem Beispiel eine Stunde), bevor er ausgeführt wird.

Aktivitätsausführungskontext

Genau wie der `DecisionContext` enthält der Aktivitätsausführungskontext Kontextinformationen zur Verarbeitung einer Entscheidungsaufgabe. `ActivityExecutionContext` stellt ähnliche Kontextinformationen bereit, wenn eine Aktivitätsausgabe verarbeitet wird. Dieser Kontext ist für Ihren Aktivitätscode über die Klasse `ActivityExecutionContextProviderImpl` verfügbar.

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

Mit `ActivityExecutionContext` können Sie folgende Aufgaben ausführen:

Heartbeat für eine langfristige Aktivität

Wenn die Aktivität lange andauert, muss sie ihren Fortschritt regelmäßig an Amazon SWF melden, um sie darüber zu informieren, dass die Aufgabe weiterhin voranschreitet. Wenn kein Heartbeat gesendet wird, kann eine Zeitüberschreitung auftreten, wenn diese bei der Registrierung des Aktivitätstyps oder beim Planen der Aktivität definiert wurde. Um einen Heartbeat zu senden, können Sie die `recordActivityHeartbeat`-Methode im `ActivityExecutionContext` verwenden. Ein Heartbeat kann auch dazu dienen, laufende Aktivitäten abzubrechen. Weitere Informationen sowie ein Beispiel finden Sie im Abschnitt Fehlerbehandlung.

Abrufen von Details zur Aktivitätsaufgabe

Wenn Sie möchten, können Sie alle Details der Aktivitätsaufgabe abrufen, die von Amazon SWF übergeben wurden, als der Executor die Aufgabe erhielt. Dies umfasst Informationen zu den Eingaben der Aufgabe, Aufgabentyp, Aufgabentoken usw. Wenn Sie eine Aktivität implementieren möchten, die manuell abgeschlossen wird, z. B. durch eine menschliche Aktion, müssen Sie das verwenden, um das Aufgaben-Token abzurufen und es `ActivityExecutionContext` an den Prozess weiterzuleiten, der die Aktivitätsaufgabe letztendlich abschließt. Weitere Informationen finden Sie im Abschnitt zu Aktivitäten manuell abschließen.

Ruft das Amazon SWF-Client-Objekt ab, das vom Executor verwendet wird

Das vom Executor verwendete Amazon SWF-Client-Objekt kann durch Aufrufen der `getService` Methode on abgerufen werden. `ActivityExecutionContext` Dies ist nützlich, wenn Sie den Amazon SWF-Service direkt anrufen möchten.

Untergeordnete Workflow-Ausführungen

In den bisherigen Beispielen wurde die Workflow-Ausführung direkt in einer Anwendung gestartet. Eine Workflow-Ausführung kann jedoch auch innerhalb eines Workflows gestartet werden, indem für den generierten Client die Workflow-Eintrittspunktmethode aufgerufen wird. Wenn eine Workflow-Ausführung im Kontext der Ausführung eines anderen Workflows gestartet wird, ist das eine untergeordnete Workflow-Ausführung. Damit können Sie komplexe Workflows in kleinere Einheiten

unterteilen und gegebenenfalls in verschiedenen Workflows einsetzen. Sie können zum Beispiel einen Workflow zur Zahlungsabwicklung erstellen und über den Workflow zur Abwicklung des Bestellvorgangs aufrufen.

Die untergeordnete Workflow-Ausführung erfolgt semantisch genauso wie ein eigenständiger Workflow – mit Ausnahme der folgenden Unterschiede:

1. Wenn der übergeordnete Workflow aufgrund einer expliziten Aktion des Benutzers beendet wird, z. B. durch Aufrufen der `TerminateWorkflowExecution` Amazon SWF SWF-API, oder aufgrund eines Timeouts beendet wird, wird das Schicksal der Ausführung des untergeordneten Workflows durch eine untergeordnete Richtlinie bestimmt. Sie können diese untergeordnete Richtlinie so einrichten, dass die Ausführung des untergeordneten Workflows beendet, abgebrochen oder verworfen (läuft weiter) wird.
2. Die Ausgabe des untergeordneten Workflows (Rückgabewert der Eintrittspunktmethode) kann von der übergeordneten Workflow-Ausführung genau so wie der mit einer asynchronen Methode zurückgegebene `Promise<T>` verwendet werden. Dies unterscheidet sich von eigenständigen Ausführungen, bei denen die Anwendung die Ausgabe mithilfe von Amazon SWF APIs abrufen muss.

Im folgenden Beispiel erstellt der `OrderProcessor`-Workflow einen untergeordneten Workflow `PaymentProcessor`:

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

    @Execute(version = "1.0")
    void processOrder(Order order);
}

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

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

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

}

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
                defaultTaskStartToCloseTimeoutSeconds = 10)
public interface PaymentProcessor {

    @Execute(version = "1.0")
    void processPayment(float amount, CardInfo cardInfo);

}

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

    @Override
    public void processPayment(float amount, CardInfo cardInfo) {
        Promise<PaymentType> payType = activitiesClient.getPaymentType(cardInfo);
        switch(payType.get()) {
        case Visa:
            activitiesClient.processVisa(amount, cardInfo);
            break;
        case Amex:
            activitiesClient.processAmex(amount, cardInfo);
            break;
      default:
         throw new UnSupportedPaymentTypeException();
        }
    }

}

@Activities(version = "1.0")
@ActivityRegistrationOptions(defaultTaskScheduleToStartTimeoutSeconds = 3600,
                             defaultTaskStartToCloseTimeoutSeconds = 3600)
public interface PaymentActivities {

    PaymentType getPaymentType(CardInfo cardInfo);

    void processVisa(float amount, CardInfo cardInfo);
```

```
    void processAmex(float amount, CardInfo cardInfo);

}
```

Fortlaufende Workflows

In einigen Anwendungsfällen benötigen Sie vielleicht einen Workflow, der ständig oder für eine lange Zeit ausgeführt wird – wie zum Beispiel ein Workflow, der den Zustand einer Serverflotte überwacht.

> **Note**
>
> Da Amazon SWF den gesamten Verlauf einer Workflow-Ausführung speichert, wird der Verlauf im Laufe der Zeit weiter wachsen. Bei einem erneuten Abspielen ruft das Framework diesen Verlauf von Amazon SWF ab, was bei einem zu großen Umfang des Verlaufs teuer werden kann. Bei solchen lange ausgeführten oder fortlaufenden Workflows sollten Sie die aktuelle Ausführung regelmäßig schließen und eine neue Ausführung starten, um die Verarbeitung fortzusetzen.

Das ist eine logische Fortsetzung der Workflow-Ausführung. Der generierte Self-Client kann für diesen Zeck verwendet werden. Rufen Sie in Ihrer Workflow-Implementierung einfach die `@Execute`-Methode für den Self-Client auf. Sobald die aktuelle Ausführung abgeschlossen ist, startet das Framework mit derselben Workflow-ID eine neue Ausführung.

Sie können die Ausführung auch fortsetzen, indem Sie die `continueAsNewOnCompletion`-Methode, die Sie vom aktuellen `DecisionContext` abrufen können, für den `GenericWorkflowClient` aufrufen. Mit der folgenden Workflow-Implementierung wird zum Beispiel ein Timer festgelegt. Dieser wird nach einem Tag ausgelöst und ruft einen eigenen Eintrittspunkt auf, der eine neue Ausführung startet.

```
public class ContinueAsNewWorkflowImpl implements ContinueAsNewWorkflow {

    private DecisionContextProvider contextProvider
         = new DecisionContextProviderImpl();

    private ContinueAsNewWorkflowSelfClient selfClient
         = new ContinueAsNewWorkflowSelfClientImpl();
```

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

    @Override
    public void startWorkflow() {
        Promise<Void> timer = clock.createTimer(86400);
        continueAsNew(timer);
    }

    @Asynchronous
    void continueAsNew(Promise<Void> timer) {
        selfClient.startWorkflow();
    }
}
```

Wenn sich ein Workflow rekursiv selbst aufruft, schließt das Framework den aktuellen Workflow nach Abschluss aller ausstehenden Aufgaben und startet eine neue Workflow-Ausführung. Solange noch Aufgaben ausstehen, wird die aktuelle Workflow-Ausführung nicht geschlossen. Die neue Ausführung erbt nicht automatisch den Verlauf oder Daten aus der ursprünglichen Ausführung. Wenn Sie bestimmte Statusangaben in die neue Ausführung übernehmen möchten, müssen Sie diese ausdrücklich als Eingabe übergeben.

Aufgabenpriorität in Amazon SWF festlegen

Standardmäßig werden Aufgaben in einer Aufgabenliste basierend auf ihrer Ankunftszeit bereitgestellt: Aufgaben, die zuerst geplant wurden, werden möglichst zuerst ausgeführt. Indem Sie eine optionale Aufgabenpriorität festlegen, können Sie bestimmten Aufgaben Priorität einräumen: Amazon SWF versucht, Aufgaben mit höherer Priorität auf einer Aufgabenliste vor Aufgaben mit niedrigerer Priorität zuzuweisen.

Sie können die Aufgabenpriorität sowohl für Workflows als auch Aktivitäten einrichten. Die Aufgabenpriorität eines Workflows wirkt sich weder auf die Priorität von durch den Workflow geplanten Aktivitätsaufgaben noch auf vom Workflow gestartete untergeordnete Workflows aus. Die Standardpriorität für eine Aktivität oder einen Workflow wird bei der Registrierung festgelegt (entweder von Ihnen oder von Amazon SWF), und die registrierte Aufgabenpriorität wird immer verwendet, sofern sie nicht beim Planen der Aktivität oder beim Starten einer Workflow-Ausführung außer Kraft gesetzt wird.

Die Werte für die Aufgabenpriorität müssen im Bereich von "-2147483648" und "2147483647" liegen. Höhere Zahlen geben dabei eine höhere Priorität an. Wenn Sie für eine Aktivität oder einen Workflow keine Aufgabenpriorität festlegen, wird eine Priorität von Null ("0") zugewiesen.

Themen

- Einrichten der Aufgabenpriorität für Workflows
- Einrichten der Aufgabenpriorität für Aktivitäten

Einrichten der Aufgabenpriorität für Workflows

Sie können die Aufgabenpriorität für einen Workflow beim Registrieren oder Starten des Workflows einrichten. Die beim Registrieren eines Workflowtyps festgelegte Aufgabenpriorität wird standardmäßig für alle Workflow-Ausführungen dieses Typs verwendet, sofern sie beim Starten der Workflow-Ausführung nicht überschrieben wird.

Um einen Workflow-Typ mit einer standardmäßigen Aufgabenpriorität zu registrieren, legen Sie WorkflowRegistrationOptionsbei der Deklaration die defaultTaskPriorityOption fest:

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

Sie können auch die taskPriority für einen Workflow festlegen, wenn Sie ihn starten, wobei die registrierte (standardmäßige) Aufgabenpriorität überschrieben wird.

```
StartWorkflowOptions priorityWorkflowOptions
    = new StartWorkflowOptions().withTaskPriority(10);

PriorityWorkflowClientExternalFactory cf
    = new PriorityWorkflowClientExternalFactoryImpl(swfService, domain);

priority_workflow_client = cf.getClient();

priority_workflow_client.startWorkflow(
```

```
        "Smith, John", priorityWorkflowOptions);
```

Zusätzlich können Sie die Aufgabenpriorität festlegen, wenn Sie einen untergeordneten Workflow starten oder einen Workflow als neu fortsetzen. Sie können beispielsweise die Option TaskPriority in ContinueAsNewWorkflowExecutionParameters oder in StartChildWorkflowExecutionParameters festlegen.

Einrichten der Aufgabenpriorität für Aktivitäten

Sie können die Aufgabenpriorität für eine Aktivität entweder beim Registrieren oder Planen der Aufgabe einrichten. Die beim Registrieren eines Aktivitätstyps festgelegte Aufgabenpriorität wird standardmäßig beim Ausführen der Aktivität verwendet, sofern sie beim Planen der Aktivität nicht überschrieben wird.

Um einen Aktivitätstyp mit einer standardmäßigen Aufgabenpriorität zu registrieren, legen Sie die defaultTaskPriorityOption ActivityRegistrationOptions bei der Deklaration fest:

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

Sie können auch die taskPriority für eine Aktivität festlegen, wenn Sie ihn planen, wobei die registrierte (standardmäßige) Aufgabenpriorität überschrieben wird.

```
ActivitySchedulingOptions activityOptions = new
 ActivitySchedulingOptions.withTaskPriority(10);

ImportantActivitiesClient activityClient = new ImportantActivitiesClientImpl();

activityClient.doSomethingImportant(activityOptions);
```

DataConverters

Wenn Ihre Workflow-Implementierung eine Remote-Aktivität aufruft, werden die Eingaben dorthin übergeben und das Ergebnis der Ausführung der Aktivität muss serialisiert werden, sodass sie über den Draht gesendet werden können. Das Framework verwendet die DataConverter

Klasse für diesen Zweck. Dies ist eine abstrakte Klasse, die Sie implementieren können, um Ihren eigenen Serializer anzugeben. Eine auf dem Jackson-Serializer basierende Standardimplementierung`JsonDataConverter`, ist im Framework enthalten. Weitere Einzelheiten dazu finden Sie in der AWS SDK für Java -Dokumentation. In der Jackson JSON Processor-Dokumentation finden Sie weitere Informationen dazu, wie Jackson die Serialisierung durchführt, sowie Jackson-Annotationen, die für deren Beeinflussung verwendet werden können. Das verwendete Drahtformat wird als Teil des Vertrags angesehen. Sie können also einen `DataConverter` auf Ihren Aktivitäten und Workflow-Schnittstellen angeben, indem Sie die `DataConverter`-Eigenschaft der `@Activities`- und `@Workflow`-Annotationen festlegen.

Das Framework erzeugt Objekte vom `DataConverter`-Typ, den Sie in der `@Activities`-Annotation angegeben haben, um die Eingaben für die Aktivität zu serialisieren und ihr Ergebnis zu deserialisieren. Ähnlich werden Objekte vom `DataConverter`-Typ, den Sie in der `@Workflow`-Annotation angeben, verwendet, um Parameter, die Sie an den Workflow übergeben, zu serialisieren und im Fall eines untergeordneten Workflows das Ergebnis zu deserialisieren. Zusätzlich zu den Eingaben übergibt das Framework auch zusätzliche Daten an Amazon SWF, z. B. Ausnahmedetails. Der Workflow-Serializer wird auch für die Serialisierung dieser Daten verwendet.

Sie können auch eine Instance des `DataConverter` angeben, wenn Sie nicht möchten, dass das Framework sie automatisch erstellt. Die generierten Clients haben Konstruktorüberlastungen, die einen `DataConverter` akzeptieren.

Wenn Sie keinen `DataConverter`-Typ angeben und kein `DataConverter`-Objekt übergeben, wird der `JsonDataConverter` standardmäßig verwendet.

Datenübergabe an asynchrone Methoden

Themen

- Übergabe von Collections und Maps an asynchrone Methoden
- Einstellbare <T>
- @NoWait
- Promise <Void>
- AndPromise und OrPromise

Die Verwendung von `Promise<`*`T`*`>` wurde in den vorangegangenen Abschnitten erläutert. Hier werden einige fortgeschrittene Anwendungsfälle von `Promise<`*`T`*`>` besprochen.

Übergabe von Collections und Maps an asynchrone Methoden

Das Framework unterstützt die Übergabe von Arrays, Collections und Maps als `Promise`-Typen an asynchrone Methoden. Beispielsweise kann eine asynchrone Methode, wie im Folgenden gezeigt, `Promise<ArrayList<String>>` als Argument entgegennehmen.

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

Semantisch verhält sich diese Variante wie jeder andere typisierte `Promise`-Parameter und die asynchrone Methode wartet, bis die Collection verfügbar wird, bevor sie ausgeführt wird. Wenn die Mitglieder einer Collection `Promise`-Objekte sind, können Sie das Framework warten lassen, bis alle Mitglieder bereit sind. Dies ist im folgenden Snippet zu sehen. Dadurch wartet die asynchrone Methode auf die Verfügbarkeit aller Mitglieder der Collection.

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

Beachten Sie, dass die `@Wait`-Annotation für den Parameter verwendet werden muss. Diese zeigt an, das `Promise`-Objekte enthalten sind.

Beachten Sie außerdem, dass die Aktivität `printActivity` ein `String`-Argument entgegennimmt, die entsprechende Methode im generierten Client jedoch ein Promise<String>-Argument erwartet. Wir rufen die Methode für den Client auf. Wir rufen nicht die Aktivitätsmethode direkt auf.

Einstellbare <T>

`Settable<T>` ist ein von `Promise<T>` abgeleiteter Typ, der eine Set-Methode zur Verfügung stellt, mit der Sie den `Promise`-Wert manuell einstellen können. Beispielsweise wartet der folgende Workflow auf den Empfang eines Signals, indem er auf ein `Settable<?>` wartet, das in der Signalmethode festgelegt ist:

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

   //@Execute method
   @Override
   public Promise<String> start() {
      return done(result);
   }

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

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

Ein `Settable<?>`-Wert kann außerdem mit einem anderen Promise-Objekt verkettet werden. Mit `AndPromise` und `OrPromise` können Sie Promise-Objekte gruppieren. Sie können die Verkettung eines verketteten `Settable` aufheben, indem Sie die `unchain()`-Methode aufrufen. Wenn eine Verkettung vorhanden ist, steht `Settable<?>` automatisch bereit, wenn das verkettete Promise-Objekt bereit ist. Die Verkettung ist besonders dann nützlich, wenn Sie ein im Rahmen eines `doTry()`-Aufrufes zurückgegebenes Promise-Objekt in anderen Teilen Ihres Programms verwenden wollen. Da `TryCatchFinally` es sich um eine verschachtelte Klasse handelt, können Sie a nicht `Promise<>` im Gültigkeitsbereich des übergeordneten Objekts deklarieren und festlegen. `doTry()` Dies liegt daran, dass in Java Variablen im übergeordneten Bereich deklariert und in verschachtelten Klassen verwendet werden müssen, um als endgültig markiert zu werden. Zum Beispiel:

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

    new TryFinally() {

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

```
            activity2(resultToChain);

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

        @Override
        protected void doFinally() throws Throwable {
            if (result.isReady()) { // Was a result returned before the exception?
                // Do cleanup here
            }
        }
    };

    return result;
}
```

Ein `Settable` kann jeweils mit einem Promise-Objekt verkettet werden. Sie können die Verkettung eines verketteten `Settable` aufheben, indem Sie die `unchain()`-Methode aufrufen.

@NoWait

Wenn Sie ein `Promise` an eine asynchrone Methode übergeben, wartet das Framework standardmäßig, bis die `Promise`(s) bereit sind, bevor es die Methode ausführt (außer bei Collection-Typen). Sie können dieses Verhalten überschreiben, indem Sie in der Deklaration der asynchronen Methode die `@NoWait`-Notation für die Parameter verwenden. Dies ist dann nützlich, wenn Sie in `Settable<T>` Werte übergeben, die durch die asynchrone Methode selbst festgelegt werden.

Promise <Void>

Abhängigkeiten in asynchronen Methoden werden implementiert, indem das von einer Methode zurückgegebene `Promise`-Objekt als Argument an eine andere Methode übergeben wird. Es kann jedoch Fälle geben, in denen Sie aus einer Methode einen `void`-Wert zurückgeben möchten, aber dennoch andere asynchrone Methoden nach ihrer Beendigung ausführen möchten. In solchen Fällen können Sie `Promise<Void>` als Rückgabetyp der Methode verwenden. Die Klasse `Promise` stellt eine statische `Void`-Methode zur Verfügung, mit der Sie ein `Promise<Void>`-Objekt anlegen können. Dieses `Promise`-Objekt ist dann bereit, wenn die asynchrone Methode die Ausführung beendet. Sie können das `Promise` wie jedes andere `Promise`-Objekt an eine andere asynchrone Methode übergeben. Wenn Sie `Settable<Void>` verwenden, dann rufen Sie zur Bereitstellung dessen Set-Methode mit "null" auf.

AndPromise und OrPromise

Mit `AndPromise` und `OrPromise` können Sie mehrere `Promise<>`-Objekte zu einem einzigen logischen Promise-Objekt zusammenfassen. Ein `AndPromise` ist dann bereit, wenn alle zur Erstellung verwendeten Promise-Objekte bereit sind. Ein `OrPromise` ist dann bereit, wenn alle Promise-Objekte in der zur Erstellung verwendeten Promise-Collection bereit sind. Sie können `getValues()` für `AndPromise` und `OrPromise` aufrufen, um die Werteliste der einzelnen Promise-Objekte abzurufen.

Prüfbarkeit und Dependency Injection

Themen

- Spring-Integration
- JUnit Integration

Das Framework ist auf die Unterstützung von IoC (Inversion of Control, Umkehr des Kontrollflusses) ausgelegt. Aktivitäts- und Workflow-Implementierungen sowie die vom Framework bereitgestellten Worker und Kontextobjekte können mit Containern wie Spring konfiguriert und instanziiert werden. Das Framework kann standardmäßig in das Spring Framework integriert werden. Darüber hinaus JUnit wurde eine Integration für die Implementierung von Workflows und Aktivitäten für Unit-Tests bereitgestellt.

Spring-Integration

Das Paket "com.amazonaws.services.simpleworkflow.flow.spring" enthält Klassen, die die Verwendung des Spring-Frameworks in Ihren Anwendungen vereinfacht. Dazu zählen benutzerdefinierte Scope-und-Spring-fähige Aktivitäts- und Workflow-Worker: `WorkflowScope`, `SpringWorkflowWorker` und `SpringActivityWorker`. Diese Klassen ermöglichen Ihnen die vollständige Konfiguration Ihrer Workflow- und Aktivitätsimplementierungen sowie der Worker mit Spring.

WorkflowScope

`WorkflowScope` – Eine benutzerdefinierte Spring Scope-Implementierung, die vom Framework bereitgestellt wird. Mit diesem Scope können Sie Objekte in Spring-Container erstellen, dessen Lebensdauer an die der Entscheidungsaufgabe angepasst ist. Die Beans in diesem Scope werden

immer dann instanziiert, wenn der Worker eine neue Entscheidungsaufgabe empfängt. Sie sollten diesen Scope für Workflow-Implementierungs-Beans und anderen Beans, von denen er abhängt, verwenden. Die von Spring bereitgesellten Singleton- und Prototype-Scopes sollten nicht für Workflow-Implementierungs-Beans eingesetzt werden, da das Framework erfordert, dass für jede Entscheidungsaufgabe eine neue Bean erstellt werden kann. Wenn Sie dies nicht tun, kommt es zu einem unerwünschten Verhalten.

Das folgende Beispiel zeigt einen Ausschnitt einer Spring-Konfigurationen, bei der der `WorkflowScope` registriert und anschließend für die Konfiguration einer Workflow-Implementierungs-Bean und einer Aktivitäts-Client-Bean eingesetzt wird.

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

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

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

Die Konfigurationszeile `<aop:scoped-proxy proxy-target-class="false" />`, die bei der Konfiguration der `workflowImpl`-Bean verwendet wird, ist erforderlich, da `WorkflowScope` ein Proxying mittels CGLIB nicht unterstützt. Sie sollte diese Konfiguration für alle Beans im `WorkflowScope` verwenden, die mit anderen Beans in einem anderen Scope verbunden sind. In diesem Fall muss die `workflowImpl`-Bean mit einer Workflow-Worker-Bean in einem Singleton-Scope verknüpft werden (siehe Beispiel unten).

Weitere Informationen zur Verwendung benutzerdefinierter Scopes finden Sie in der Spring Framework-Dokumentation.

Spring-fähige Worker

Bei der Arbeit mit Spring sollten Sie die Spring-fähigen Worker-Klassen nutzen, die vom Framework bereitgestellt werden: `SpringWorkflowWorker` und `SpringActivityWorker`. Diese Worker können mittels Spring in Ihre Anwendung eingefügt werden, wie im folgenden Beispiel gezeigt. Die Spring-fähigen Worker implementieren Springs `SmartLifecycle`-Schnittstelle und starten standardmäßig automatisch das Abrufen von Aufgaben, wenn der Spring-Kontext initialisiert wurde. Sie können diese Funktion deaktivieren, indem Sie die `disableAutoStartup`-Eigenschaft des Workers auf `true` setzen.

Das folgende Beispiel zeigt die Konfiguration eines Entscheiders. In diesem Beispiel werden die Schnittstellen `MyActivities` und `MyWorkflow` (hier nicht abgebildet) sowie die entsprechenden Implementierungen `MyActivitiesImpl` und `MyWorkflowImpl` verwendet. Die generierten Client-Schnittstellen und -Implementierungen sind `MyWorkflowClient`/`MyWorkflowClientImpl` und `MyActivitiesClient`/`MyActivitiesClientImpl` (ebenfalls nicht abgebildet).

Der Aktivitäts-Client wird über die "auto wire"-Funktion von Spring in die Workflow-Implementierung eingefügt:

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

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

Die Spring-Konfiguration des Entscheiders sieht wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop="http://www.springframework.org/schema/aop"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans.xsd
```

```
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/
spring-aop-2.5.xsd
   http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <!-- register custom workflow scope -->
   <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
      <property name="scopes">
       <map>
         <entry key="workflow">
          <bean
 class="com.amazonaws.services.simpleworkflow.flow.spring.WorkflowScope" />
         </entry>
       </map>
      </property>
   </bean>
   <context:annotation-config/>

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

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

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

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

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

```
   </bean>

   <!-- workflow worker -->
   <bean id="workflowWorker"
      class="com.amazonaws.services.simpleworkflow.flow.spring.SpringWorkflowWorker">
      <constructor-arg ref="swfClient" />
      <constructor-arg value="domain1" />
      <constructor-arg value="tasklist1" />
      <property name="registerDomain" value="true" />
      <property name="domainRetentionPeriodInDays" value="1" />
      <property name="workflowImplementations">
         <list>
            <ref bean="workflowImpl" />
         </list>
      </property>
   </bean>
</beans>
```

Da der in Spring vollständig konfiguriert `SpringWorkflowWorker` ist und bei der Initialisierung des Spring-Kontextes automatisch mit der Abfrage beginnt, ist der Hostprozess für den Entscheider einfach:

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

Entsprechend kann auch der Aktivitäts-Worker wie folgt konfiguriert werden:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop="http://www.springframework.org/schema/aop"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/
spring-aop-2.5.xsd
   http://www.springframework.org/schema/context
```

```
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <!-- register custom scope -->
  <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
     <property name="scopes">
        <map>
           <entry key="workflow">
              <bean

class="com.amazonaws.services.simpleworkflow.flow.spring.WorkflowScope" />
           </entry>
        </map>
     </property>
  </bean>

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

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

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

  <!-- activities impl -->
  <bean name="activitiesImpl" class="asadj.spring.test.MyActivitiesImpl">
  </bean>

  <!-- activity worker -->
  <bean id="activityWorker"
     class="com.amazonaws.services.simpleworkflow.flow.spring.SpringActivityWorker">
     <constructor-arg ref="swfClient" />
     <constructor-arg value="domain1" />
     <constructor-arg value="tasklist1" />
     <property name="registerDomain" value="true" />
     <property name="domainRetentionPeriodInDays" value="1" />
```

```
      <property name="activitiesImplementations">
         <list>
            <ref bean="activitiesImpl" />
         </list>
      </property>
   </bean>
</beans>
```

Der Hostprozess des Aktivitäts-Workers ähnelt dem des Entscheiders:

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

Einfügen des Entscheidungskontexts

Wie Ihre Workflow-Implementierung von den Kontextobjekten abhängt, können Sie diese ebenfalls ganz einfach mit Spring einfügen. Das Framework registriert kontextbasierte Beans automatisch im Spring-Container. Im folgenden Codeausschnitt wurden beispielsweise verschiedene Kontextobjekte automatisch verknüpft. Eine weitere Spring-Konfiguration der Kontextobjekte ist nicht erforderlich.

```
public class MyWorkflowImpl implements MyWorkflow {
   @Autowired
   public MyActivitiesClient client;
   @Autowired
   public WorkflowClock clock;
   @Autowired
   public DecisionContext dcContext;
   @Autowired
   public GenericActivityClient activityClient;
   @Autowired
   public GenericWorkflowClient workflowClient;
   @Autowired
   public WorkflowContext wfContext;
   @Override
   public void start() {
      client.activity1();
   }
```

```
}
```

Wenn Sie die Kontextobjekte in der Workflow-Implementierung über die Spring-XML-Konfiguration konfigurieren möchten, verwenden Sie die Bean-Namen, die in der `WorkflowScopeBeanNames`-Klasse im Paket "com.amazonaws.services.simpleworkflow.flow.spring" deklariert sind. Zum Beispiel:

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

Alternativ können Sie auch einen `DecisionContextProvider` in die Bean der Workflow-Implementierung einfügen und zum Erstellen des Kontexts verwenden. Dies ist hilfreich, wenn Sie benutzerdefinierte Implementierungen des Providers und Kontexts bereitstellen möchten.

Einfügen von Ressourcen in Aktivitäten

Sie können Aktivitätsimplementierungen mit einem IoC-Container instanziieren und konfigurieren und Ressourcen wie Datenbankverbindungen einfügen, indem Sie diese als Eigenschaften der Klasse der Aktivitätsimplementierung deklarieren. Diese Ressourcen werden in der Regeln als Singletons definiert. Beachten Sie, dass Aktivitätsimplementierungen vom Aktivitäts-Worker auf verschiedenen Threads aufgerufen werden. Deshalb muss der Zugriff auf freigegebene Ressourcen synchronisiert werden.

JUnit Integration

Das Framework bietet JUnit Erweiterungen sowie Testimplementierungen der Kontextobjekte, z. B. eine Testuhr, mit der Sie Komponententests schreiben und ausführen können. JUnit Mit diesen Erweiterungen ist ein lokaler Inline-Test der Workflow-Implementierung möglich.

Schreiben eines einfachen Einheitentests

Verwenden Sie zum Entwerfen von Tests für Ihren Workflow die `WorkflowTest`-Klasse aus dem Paket "com.amazonaws.services.simpleworkflow.flow.junit". Bei dieser Klasse handelt es sich um

eine Framework-spezifische JUnit `MethodRule` Implementierung. Sie führt Ihren Workflow-Code lokal aus und ruft Aktivitäten inline auf, anstatt Amazon SWF zu verwenden. Dadurch haben Sie die Möglichkeit, Ihre Test so oft Sie möchten, auszuführen, ohne dass Gebühren anfallen.

Wenn Sie diese Klasse verwenden möchten, deklarieren Sie einfach ein Feld vom Typ `WorkflowTest` und versehen es mit der Anmerkung `@Rule`. Erstellen Sie vor der Ausführung Ihrer Tests ein neues `WorkflowTest`-Objekt und fügen Sie diesem Ihre Aktivitäts- und Workflow-Implementierungen hinzu. Sie können die generierte Workflow-Client-Factory zum Erstellen eines Clients und zum Starten der Ausführung des Workflows verwenden. Das Framework bietet auch einen benutzerdefinierten JUnit Runner`FlowBlockJUnit4ClassRunner`, den Sie für Ihre Workflow-Tests verwenden müssen. Zum Beispiel:

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

    @Rule
    public WorkflowTest workflowTest = new WorkflowTest();

    List<String> trace;

    private BookingWorkflowClientFactory workflowFactory
         = new BookingWorkflowClientFactoryImpl();

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

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

    @Test
    public void testReserveBoth() {
        BookingWorkflowClient workflow = workflowFactory.getClient();
        Promise<Void> booked = workflow.makeBooking(123, 345, true, true);
        List<String> expected = new ArrayList<String>();
```

```
        expected.add("reserveCar-123");
        expected.add("reserveAirline-123");
        expected.add("sendConfirmation-345");
        AsyncAssert.assertEqualsWaitFor("invalid booking", expected, trace, booked);
    }
}
```

Sie können zudem für jede Aktivitätsimplementierung, die Sie zu `WorkflowTest` hinzufügen, eine separate Aufgabenliste angeben. Wenn Sie beispielsweise eine Workflow-Implementierung haben, die Aktivitäten in hostspezifischen Aufgabenlisten plant, können Sie die Aktivität in der Aufgabenliste der einzelnen Hosts registrieren:

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

Beachten Sie, dass der Code in `@Test` asynchron ist. Deshalb sollten Sie die Ausführung mit dem asynchronen Workflow-Client starten. Zur Überprüfung der Testergebnisse steht eine `AsyncAssert`-Hilfsklasse zur Verfügung. Diese Klasse ermöglicht Ihnen das Warten auf sog. Promises, die darüber informieren, dass die Operation vor der Verifizierung der Ergebnisse abgeschlossen ist. In diesem Beispiel wird auf das Ergebnis der Workflow-Ausführung gewartet, um vor dem Verifizieren der Testausgabe fertig zu sein.

Wenn Sie Spring benutzen, dann kann die `SpringWorkflowTest`-Klasse anstelle der `WorkflowTest`-Klasse verwendet werden. `SpringWorkflowTest` stellt Eigenschaften bereit, die Sie verwenden können, um Aktivitäts- und Workflow-Implementierungen einfach über die Spring-Konfiguration zu konfigurieren. Genau wie die Spring-fähigen Worker sollten Sie zum Konfigurieren von Workflow-Implementierungs-Beans den `WorkflowScope` verwenden. Das sorgt dafür, dass für jede Entscheidungsaufgabe eine neue Workflow-Implementierungs-Bean generiert wird. Stellen Sie sicher, dass Sie diese Beans so konfigurieren, dass die proxy-target-class Scoped-Proxy-Einstellung auf eingestellt ist. `false` Weitere Informationen finden Sie im Abschnitt zur Spring-Integration. Das Beispiel der Spring-Konfiguration, das in diesem Abschnitt gezeigt wird, kann geändert werden, um den Workflow mit `SpringWorkflowTest` zu testen:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://
www.springframework.org/schema/aop"
```

```
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans ht
tp://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop http://www.springframe
work.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

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

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

  <!-- workflow implementation -->
  <bean id="workflowImpl" class="aws.flow.sample.MyWorkflowImpl"
    scope="workflow">
```

```
    <property name="client" ref="activitiesClient" />
    <aop:scoped-proxy proxy-target-class="false" />
  </bean>

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

Nachahmen von Aktivitätsimplementierungen

Sie können während des Testens echte Aktivitätsimplementierungen verwenden. Wenn Sie aber nur einen Einheitentest für die Workflow-Logik durchführen möchten, sollten Sie die Aktivitäten nachahmen. Dazu stellen Sie eine Mock-Implementierung der Aktivitätsschnittstelle für die `WorkflowTest`-Klasse bereit. Zum Beispiel:

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

    @Rule
    public WorkflowTest workflowTest = new WorkflowTest();

    List<String> trace;

    private BookingWorkflowClientFactory workflowFactory
         = new BookingWorkflowClientFactoryImpl();
```

```
    @Before
    public void setUp() throws Exception {
        trace = new ArrayList<String>();
        // Create and register mock activity implementation to be used during test run
        BookingActivities activities = new BookingActivities() {

            @Override
            public void sendConfirmationActivity(int customerId) {
                trace.add("sendConfirmation-" + customerId);
            }

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

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

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

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

Alternativ können Sie eine Mock-Implementierung des Aktivitäts-Clients bereitstellen und in Ihre Workflow-Implementierung einfügen.

Testen von Kontextobjekten

Wenn Ihre Workflow-Implementierung von den Framework-Kontextobjekten abhängt, z. B. von, müssen `DecisionContext` Sie nichts Besonderes tun, um solche Workflows zu testen. Wird ein Test mittels `WorkflowTest` durchgeführt, werden automatisch Testkontextobjekte eingefügt. Wenn Ihre Workflow-Implementierung auf die Kontextobjekte zugreift — zum Beispiel mithilfe —, wird sie die Testimplementierung erhalten. `DecisionContextProviderImpl` Sie können diese Testkontextobjekte in Ihrem Testcode ändern (`@Test`-Methode), um relevante Testfälle zu entwerfen. Erstellt Ihr Workflow beispielsweise einen Timer, können Sie dafür sorgen, dass der Timer ausgelöst wird, indem Sie die `clockAdvanceSeconds`-Methode auf der `WorkflowTest`-Klasse aufrufen, um die Uhr vorzustellen. Mit der `ClockAccelerationCoefficient`-Eigenschaft im `WorkflowTest` können Sie ebenfalls die Uhrzeit vorstellen, damit der Timer früher als üblich ausgelöst wird. Erstellt Ihr Workflow beispielsweise einen Timer für eine Stunde, können Sie `ClockAccelerationCoefficient` auf 60 setzen, damit der Timer in einer Minute ausgelöst wird. Standardmäßig ist `ClockAccelerationCoefficient` auf "1" gesetzt.

Weitere Informationen zu den Paketen "com.amazonaws.services.simpleworkflow.flow.test" und "com.amazonaws.services.simpleworkflow.flow.junit" finden Sie in der AWS SDK für Java - Dokumentation.

Fehlerbehandlung

Themen

- TryCatchFinally Semantik
- Abbruch
- Verschachtelt TryCatchFinally

Das Konstrukt `try/catch/finally` in Java vereinfacht die Fehlerbehandlung und wird sehr häufig eingesetzt. Es ermöglicht die Verknüpfung von Fehler-Handlern mit einem Codeblock. Dies geschieht intern durch die Anhäufung von Metadaten zu den Fehler-Handlern auf dem Aufruf-Stack. Wird eine Ausnahme ausgelöst, sucht die Laufzeit beim Aufruf-Stack nach einem zugehörigen Fehler-Handler und ruft diesen auf. Wird kein passender gefunden, wird die Ausnahme an die Aufruf-Kette weitergegeben.

Dies funktioniert gut bei synchronem Code. Die Fehlerbehandlung in asynchronen und verteilten Programmen stellt jedoch einige Herausforderungen dar. Da ein asynchroner Aufruf sofort zurückkehrt, befindet sich der Aufrufer nicht auf der Aufrufliste, wenn der asynchrone Code ausgeführt wird. Das bedeutet, dass nicht behandelte Ausnahmen in einem asynchronen Code vom Aufrufer nicht in der üblichen Weise behandelt werden können. In der Regel werden Ausnahmen, die in einem asynchronen Code auftreten, behandelt, indem der Fehlerstatus an ein Callback übergeben wird, das an die asynchrone Methode übermittelt wird. Alternativ erfolgt bei Verwendung von `Future<?>` die Meldung eines Fehlers, wenn Sie versuchen, darauf zuzugreifen. Dies ist keineswegs ideal, da dem Code, der die Ausnahme empfängt (das Callback oder den Code, das bzw. der `Future<?>` verwendet), der Kontext des ursprünglichen Aufrufs fehlt und er die Ausnahme möglicherweise nicht adäquat behandeln kann. Darüber hinaus kann es bei einem verteilten asynchronen System, bei dem mehrere Komponenten parallel ausgeführt werden, gleichzeitig zu mehreren Fehlern kommen. Dabei kann es sich um unterschiedliche Fehlertypen von unterschiedlichem Schweregrad handeln, die alle entsprechend behandelt werden müssen.

Das Bereinigen einer Ressource nach einem asynchronen Aufruf ist ebenfalls schwierig. Im Gegensatz zu synchronem Code können Sie den Code try/catch/finally im aufrufenden Code nicht verwenden, um Ressourcen zu bereinigen, da die im Try-Block eingeleitete Arbeit möglicherweise noch andauert, wenn der Finally-Block ausgeführt wird.

Das Framework bietet einen Mechanismus, der die Fehlerbehandlung in verteiltem asynchronem Code der von Java ähnelt und fast so einfach wie die von Java ist. try/catch/finally

```
ImageProcessingActivitiesClient activitiesClient
     = new ImageProcessingActivitiesClientImpl();

public void createThumbnail(final String webPageUrl) {

  new TryCatchFinally() {

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

```
    }

    @Override
    protected void doCatch(Throwable e) throws Throwable {

      // Handle exception and rethrow failures
      LoggingActivitiesClient logClient = new LoggingActivitiesClientImpl();
      logClient.reportError(e);
      throw new RuntimeException("Failed to process images", e);
    }

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

Die `TryCatchFinally`-Klasse und deren Varianten `TryFinally` und `TryCatch` funktionieren ähnlich wie Javas `try`/`catch`/`finally`. Mit dieser Lösung können Sie Ausnahme-Handler mit Blöcken von Workflow-Code verknüpfen, die als asynchrone und Remote-Aufgaben ausgeführt werden können. Die `doTry()`-Methode entspricht logisch dem `try`-Block. Das Framework führt den Code automatisch in `doTry()` aus. Eine Liste von `Promise`-Objekten kann an den Konstruktor von `TryCatchFinally` übergeben werden. Die `doTry`-Methode wird ausgeführt, wenn alle `Promise` -Objekte, die an den Konstruktor übergeben wurden, bereit sind. Wird eine Ausnahme von einem Code ausgelöst, der asynchron innerhalb von `doTry()` aufgerufen wurde, werden alle Vorgänge in `doTry()` abgebrochen und `doCatch()` aufgerufen, um die Ausnahme zu behandeln. Wenn beispielsweise in der obigen Auflistung `downloadImage` eine Ausnahme auslöst, dann werden `createThumbnail` und `uploadImage` abgebrochen. Wenn alle asynchronen Vorgänge beendet wurden (abgeschlossen, fehlgeschlagen oder abgebrochen), wird abschließend `doFinally()` aufgerufen. Es kann zum Bereinigen von Ressourcen verwendet werden. Sie können diese Klassen auch gemäß Ihren Anforderungen verschachteln.

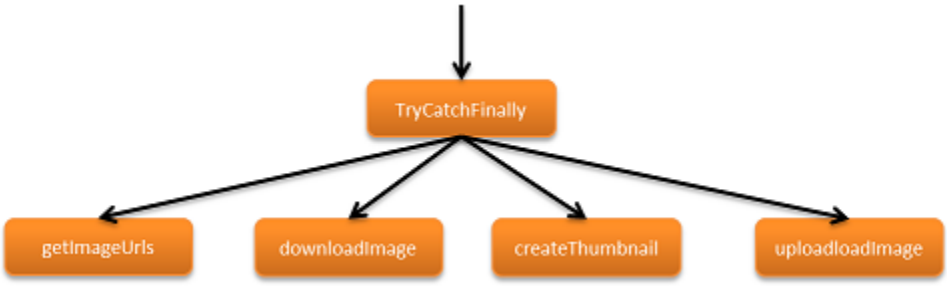
Wenn eine Ausnahme in `doCatch()` gemeldet wird, stellt das Framework einen vollständigen logischen Aufruf-Stack mit asynchronen und Remote-Aufrufen bereit. Dies kann beim Debuggen nützlich sein, insbesondere bei asynchronen Methoden, die andere asynchrone Methoden aufrufen. Eine Ausnahme von downloadImage führt beispielsweise zu einer Ausnahme wie der folgenden:

```
RuntimeException: error downloading image
```

```
   at downloadImage(Main.java:35)
   at ---continuation---.(repeated:1)
   at errorHandlingAsync$1.doTry(Main.java:24)
   at ---continuation---.(repeated:1)
…
```

TryCatchFinally Semantik

Die Ausführung eines Programms AWS Flow Framework für Java kann als Baum gleichzeitig ausgeführter Zweige visualisiert werden. Durch den Aufruf einer asynchronen Methode, einer Aktivität oder `TryCatchFinally` wird eine neue Verzweigung in dieser Baumstruktur der Ausführung angelegt. Der Bildverarbeitungs-Workflow beispielsweise ist in Form einer Baumstruktur auf folgender Abbildung zu sehen.



Ein Fehler in einer Verzweigung der Ausführung führt zu einer Entladung der Verzweigung, genau wie eine Ausnahme die Entladung eines Aufruf-Stacks in einem Java-Programm verursacht. Dieser Vorgang setzt sich fort, bis entweder der Fehler behandelt oder der Stamm erreicht ist. In diesem Fall wird die Workflow-Ausführung beendet.

Das Framework meldet Fehler, die bei der Verarbeitung von Aufgaben auftreten, als Ausnahmen. Es verknüpft die Ausnahme-Handler (`doCatch()`-Methoden), die in `TryCatchFinally` definiert sind, mit allen Aufgaben, die vom Code im entsprechenden `doTry()` erstellt wurden. Wenn eine Aufgabe fehlschlägt, z. B. aufgrund eines Timeouts oder einer unbehandelten Ausnahme, wird die entsprechende Ausnahme ausgelöst und die entsprechende wird aufgerufen, um sie zu behandeln. `doCatch()` Um dies zu erreichen, arbeitet das Framework mit Amazon SWF zusammen, um Remote-Fehler zu verbreiten und sie als Ausnahmen im Kontext des Aufrufers wieder aufleben zu lassen.

Abbruch

Tritt eine Ausnahme im asynchronen Code auf, springt das Steuerelement direkt zum `catch`-Block und überspringt den verbleibenden Code im `try`-Block. Zum Beispiel:

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

Bei diesem Code wird, wenn `b()` eine Ausnahme auslöst, `c()` niemals aufgerufen. Vergleichen Sie dies mit einem Workflow:

```
new TryCatch() {

    @Override
    protected void doTry() throws Throwable {
        activityA();
        activityB();
        activityC();
    }

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

Hier werden Aufrufe von `activityA`, `activityB` und `activityC` erfolgreich zurückgegeben und führen zur Erstellung dreier Aufgaben, die asynchron ausgeführt werden. Angenommen, die Aufgabe für `activityB` verursacht zu einem späteren Zeitpunkt einen Fehler. Dieser Fehler wird von Amazon SWF in der Historie aufgezeichnet. Aus diesem Grund versucht das Framework zunächst alle anderen Aufgaben abzubrechen, die aus dem Bereich desselben `doTry()` stammen. In diesem Fall sind das `activityA`und `activityC`. Nach Beendigung aller Aufgaben (durch Abbrechen, Fehlschlagen oder erfolgreichem Abschließen), wird die entsprechende `doCatch()`-Methode aufgerufen, um den Fehler zu behandeln.

Im Gegensatz zum synchronen Beispiel, bei dem `c()` niemals ausgeführt wurde, wurde `activityC` hier aufgerufen. Zudem wurde eine Aufgabe für die Ausführung eingeplant. Deshalb versucht das Framework einen Abbruch, für dessen Erfolg es aber keine Garantie gibt. Der Abbruch kann nicht garantiert werden, da die Aktivität möglicherweise bereits abgeschlossen ist, die Abbruchanforderung

ignoriert oder fehlschlägt. Das Framework garantiert jedoch, dass `doCatch()` nur aufgerufen wird, wenn alle Aufgaben, die über das entsprechende `doTry()` gestartet wurden, abgeschlossen sind. Es garantiert zudem, dass `doFinally()` nur aufgerufen wird, wenn alle Aufgaben, die vom `doTry()`- und `doCatch()`-Block gestartet wurden, abgeschlossen sind. Wenn die Aktivitäten im obigen Beispiel beispielsweise voneinander abhängen, beispielsweise von `activityA` und `activityC` von, dann erfolgt die Stornierung von `activityC` sofort`activityB`, da sie erst in Amazon SWF geplant ist, wenn Folgendes `activityB` abgeschlossen ist: `activityB`

```
new TryCatch() {

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

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

Aktivitäts-Heartbeat

Mit dem kooperativen Stornierungsmechanismus von AWS Flow Framework for Java können Aufgaben während des Fluges problemlos storniert werden. Wird der Abbruch ausgelöst, werden blockierte Aufgaben oder Aufgaben, die darauf warten, zu einem Worker zugewiesen zu werden, automatisch abgebrochen. Wenn eine Aufgabe aber bereits einem Worker zugewiesen wurde, fordert das Framework den Abbruch der Aktivität an. Ihre Aktivitätsimplementierung muss diese Abbruchanforderungen explizit behandeln können. Dies geschieht durch das Übermitteln von Heartbeats Ihrer Aktivität.

Durch das Senden von Heartbeats ist die Aktivitätsimplementierung in der Lage, den Fortschritt einer andauernden Aufgabe zu melden. Dies unterstützt die Überwachung und ermöglicht der Aktivität zu prüfen, ob Abbruchanforderungen vorliegen. Die `recordActivityHeartbeat`-Methode löst bei Anforderung eines Abbruchs eine `CancellationException` aus. Die Aktivitätsimplementierung kann diese Ausnahme abfangen und auf die Abbruchanforderung reagieren oder die Anforderung durch "Verschlucken" der Ausnahme ignorieren. Um der Abbruchanforderung Rechnung zu tragen, sollte die Aktivität die gewünschte Bereinigung vornehmen, sofern erforderlich,

und dann `CancellationException` erneut auslösen. Wird diese Ausnahme von einer Aktivitätsimplementierung ausgelöst, erfasst das Framework, dass die Aktivitätsaufgabe im abgebrochenen Status beendet wurde.

Das folgende Beispiel zeigt eine Aufgabe, bei der Bilder heruntergeladen und verarbeitet werden. Es kommt nach jeder Verarbeitung eines Bilds zu einem Heartbeat. Wird ein Abbruch gefordert, wird bereinigt und die Ausnahme zur Bestätigung des Abbruchs erneut ausgelöst.

```
@Override
public void processImages(List<String> urls) {
    int imageCounter = 0;
    for (String url: urls) {
        imageCounter++;
        Image image = download(url);
        process(image);
        try {
            ActivityExecutionContext context
                 = contextProvider.getActivityExecutionContext();
            context.recordActivityHeartbeat(Integer.toString(imageCounter));
        } catch(CancellationException ex) {
            cleanDownloadFolder();
            throw ex;
        }
    }
}
```

Das Senden von Aktivitäts-Heartbeats ist nicht erforderlich, wird aber empfohlen, wenn die Ausführung der Aktivität lange dauert oder dabei kostenintensive Operationen ausgeführt werden, die im Falle eines Fehlers abgebrochen werden sollten. Sie sollten `heartbeatActivityTask` periodisch von der Aktivitätsimplementierung aus aufrufen.

Kommt es bei der Ausführung der Aktivität zu einer Zeitüberschreitung, wird die `ActivityTaskTimedOutException` ausgelöst und `getDetails` auf dem Ausnahmeobjekt gibt für die entsprechende Aktivitätsaufgabe die Daten zurück, die an den letzten erfolgreichen Aufruf von `heartbeatActivityTask` übergeben wurden. Die Workflow-Implementierung kann anhand dieser Informationen feststellen, wie weit die Ausführung fortgeschritten war, ehe es zu einer Zeitüberschreitung bei der Aktivitätsaufgabe kam.

> **Note**
>
> Es empfiehlt sich nicht, zu häufig Heartbeat-Anfragen zu drosseln, da Amazon SWF Heartbeat-Anfragen drosseln kann. Informationen zu den von Amazon SWF festgelegten Beschränkungen finden Sie im Amazon Simple Workflow Service Developer Guide.

Explizites Abbrechen einer Aufgabe

Abgesehen von Fehlerbedingungen gibt es noch andere Fälle, in denen eine Aufgabe explizit abzubrechen ist. So muss beispielsweise eine Aktivität zur Verarbeitung von Zahlungen mit der Kreditkarte abgebrochen werden, wenn der Benutzer den Auftrag storniert. Das Framework ermöglicht das explizite Abbrechen von Aufgaben, die mit `TryCatchFinally` erstellt wurden. Im folgenden Beispiel wird die Zahlungsaufgabe abgebrochen, wenn während der Verarbeitung der Zahlung ein Signal empfangen wird.

```
public class OrderProcessorImpl implements OrderProcessor {
    private PaymentProcessorClientFactory factory
        = new PaymentProcessorClientFactoryImpl();
    boolean processingPayment = false;
    private TryCatchFinally paymentTask = null;

    @Override
    public void processOrder(int orderId, final float amount) {
        paymentTask = new TryCatchFinally() {

            @Override
            protected void doTry() throws Throwable {
                processingPayment = true;

                PaymentProcessorClient paymentClient = factory.getClient();
                paymentClient.processPayment(amount);
            }

            @Override
            protected void doCatch(Throwable e) throws Throwable {
                if (e instanceof CancellationException) {
                    paymentClient.log("Payment canceled.");
                } else {
                    throw e;
                }
```

```
            }

            @Override
            protected void doFinally() throws Throwable {
                processingPayment = false;
            }
        };

    }

    @Override
    public void cancelPayment() {
        if (processingPayment) {
            paymentTask.cancel(null);
        }
    }
}
```

Empfangen von Benachrichtigungen über abgebrochene Aufgaben

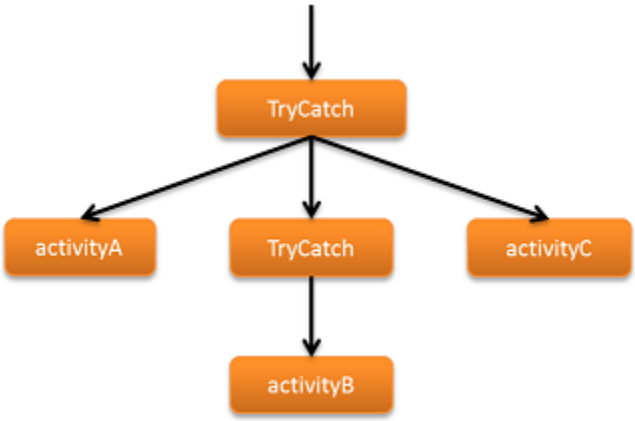
Wird eine Aufgabe im abgebrochenen Status beendet, informiert das Framework die Workflow-Logik durch Auslösen einer `CancellationException`. Wenn eine Aktivität im abgebrochenen Status beendet wird, wird ein Datensatz zum Verlauf hinzugefügt und das Framework ruft das erforderliche `doCatch()` mit einer `CancellationException` auf. Wie im vorherigen Beispiel gezeigt, empfängt der Workflow eine `CancellationException`, wenn die Aufgabe der Zahlungsverarbeitung abgebrochen wird.

Eine unbehandelte `CancellationException` wird wie jede andere Ausnahme in der Ausnahmeverzweigung weiter nach oben gereicht. Die `doCatch()`-Methode empfängt die `CancellationException` aber nur, wenn es im Scope keine weitere Ausnahme gibt. Andere Ausnahmen werden höher priorisiert als der Abbruch.

Verschachtelt TryCatchFinally

Sie können `TryCatchFinally` gemäß Ihren Anforderungen verschachteln. Da jeder Zweig in der Ausführungsstruktur einen neuen Zweig `TryCatchFinally` erstellt, können Sie verschachtelte Bereiche erstellen. Ausnahmen im übergeordneten Scope führen zu Abbruchversuchen bei allen Aufgaben, die durch ein verschachteltes `TryCatchFinally`' in ihnen initiiert wurden. Allerdings werden Ausnahmen in einem verschachtelten `TryCatchFinally` nicht automatisch an das übergeordnete Element weitergegeben. Wenn Sie eine Ausnahme aus einem verschachtelten

`TryCatchFinally` an das enthaltene `TryCatchFinally` weitergeben möchten, sollten Sie die Ausnahme in `doCatch()` erneut auslösen. Anderes ausgedrückt: Nur unbehandelte Ausnahmen steigen wie Javas `try/catch`-Konstrukt auf. Wenn Sie ein verschachteltes `TryCatchFinally` durch Aufruf der Abbruchmethode abbrechen, wird das verschachtelte `TryCatchFinally` abgebrochen, aber nicht automatisch auch das enthaltene `TryCatchFinally`.




```
new TryCatch() {
    @Override
    protected void doTry() throws Throwable {
        activityA();

        new TryCatch() {
            @Override
            protected void doTry() throws Throwable {
                activityB();
            }

            @Override
            protected void doCatch(Throwable e) throws Throwable {
                reportError(e);
            }
        };

        activityC();
    }

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

Wiederholen fehlgeschlagener Aktivitäten

Gelegentlich schlagen Aktivitäten aus temporären Gründen fehl, z. B. aufgrund eines vorübergehenden Verbindungsverlusts. In anderen Fällen wird die Aktivität möglicherweise erfolgreich durchgeführt, daher besteht das geeignete Verfahren zum Umgang mit dem Aktivitätsfehler häufig im (ggf. mehrmaligen) Wiederholen der Aktivität.

Es gibt verschiedene Strategien zum Wiederholen von Aktivitäten. Welche am besten geeignet ist, hängt von den Details in Ihrem Workflow ab. Die Strategien lassen sich grundsätzlich in drei Kategorien einteilen:

- Die retry-until-success Strategie wiederholt die Aktivität einfach so lange, bis sie abgeschlossen ist.
- Die exponentielle Wiederholungsstrategie verlängert das Zeitintervall zwischen den Versuchen exponentiell, bis die Aktivität abgeschlossen wird oder der Vorgang eine bestimmte Stoppmarke erreicht, beispielsweise eine maximale Anzahl an Versuchen.
- Die benutzerdefinierte Wiederholungsstrategie legt fest, ob und wie die Aktivität nach einem fehlgeschlagenen Versuch wiederholt wird.

In den folgenden Abschnitten wird die Implementierung dieser Strategien beschrieben. In diesem Beispiel nutzen die Workflow-Worker alle eine einzige Aktivität, `unreliableActivity`, die willkürlich eine der folgenden Verhaltensweisen zeigt:

- Wird umgehend abgeschlossen
- Schlägt beabsichtigt fehl durch Überschreiten des Timeout-Wertes
- Schlägt beabsichtigt fehl durch Ausgeben von `IllegalStateException`

Retry-Until-Success Strategie

Die einfachste Wiederholungsstrategie besteht darin, die Aktivität nach jedem Fehler zu wiederholen, bis sie schließlich erfolgreich durchgeführt werden kann. Das grundlegende Muster ist:

1. Implementieren Sie eine verschachtelte `TryCatch`- oder `TryCatchFinally`-Klasse in die Eintrittspunktmethode Ihres Workflows.
2. Führen Sie die Aktivität in `doTry` aus.
3. Falls die Aktivität fehlschlägt, ruft das Framework `doCatch` auf, wodurch die Eintrittspunktmethode erneut ausgeführt wird.

4. Wiederholen Sie die Schritte 2 bis 3, bis die Aktivität erfolgreich abgeschlossen wird.

Der folgende Workflow implementiert die retry-until-success Strategie. Die Workflow-Schnittstelle wird in `RetryActivityRecipeWorkflow` implementiert und enthält die Methode `runUnreliableActivityTillSuccess`, die den Eintrittspunkt des Workflows darstellt. Der Workflow-Worker wird in `RetryActivityRecipeWorkflowImpl` wie folgt implementiert:

```
public class RetryActivityRecipeWorkflowImpl
    implements RetryActivityRecipeWorkflow {

    @Override
    public void runUnreliableActivityTillSuccess() {
        final Settable<Boolean> retryActivity = new Settable<Boolean>();

        new TryCatch() {
            @Override
            protected void doTry() throws Throwable {
                Promise<Void> activityRanSuccessfully
                    = client.unreliableActivity();
                setRetryActivityToFalse(activityRanSuccessfully, retryActivity);
            }

            @Override
            protected void doCatch(Throwable e) throws Throwable {
                retryActivity.set(true);
            }
        };
        restartRunUnreliableActivityTillSuccess(retryActivity);
    }

    @Asynchronous
    private void setRetryActivityToFalse(
            Promise<Void> activityRanSuccessfully,
            @NoWait Settable<Boolean> retryActivity) {
        retryActivity.set(false);
    }

    @Asynchronous
    private void restartRunUnreliableActivityTillSuccess(
            Settable<Boolean> retryActivity) {
        if (retryActivity.get()) {
            runUnreliableActivityTillSuccess();
```

```
        }
    }
}
```

Der Workflow funktioniert folgendermaßen:

1. `runUnreliableActivityTillSuccess` erstellt ein `Settable<Boolean>`-Objekt namens `retryActivity`, das verwendet wird, um anzugeben, ob die Aktivität fehlgeschlagen ist und erneut getestet werden sollte. `Settable<T>` ist von `Promise<T>` abgeleitet und funktioniert zwar ähnlich, jedoch legen Sie den Wert eines `Settable<T>`-Objekts manuell fest.
2. `runUnreliableActivityTillSuccess` implementiert eine anonyme verschachtelte `TryCatch`-Klasse zur Verarbeitung von Ausnahmen, die von der `unreliableActivity`-Aktivität ausgegeben werden. Weitere Informationen zum Umgang mit Ausnahmen, die von asynchronem Code ausgegeben werden, finden Sie unter Fehlerbehandlung.
3. `doTry` führt die `unreliableActivity`-Aktivität aus, die ein `Promise<Void>`-Objekt namens `activityRanSuccessfully` zurückgibt.
4. `doTry` ruft die asynchrone `setRetryActivityToFalse`-Methode auf, die zwei Parameter umfasst:
   - `activityRanSuccessfully` übernimmt das `Promise<Void>`-Objekt, das von der `unreliableActivity`-Aktivität zurückgegeben wird.
   - `retryActivity` übernimmt das `retryActivity`-Objekt.

   Bei Abschluss von `unreliableActivity` wird `activityRanSuccessfully` einsatzbereit und `setRetryActivityToFalse` legt `retryActivity` auf "false" fest. Andernfalls wird `activityRanSuccessfully` nicht einsatzbereit und `setRetryActivityToFalse` wird nicht ausgeführt.
5. Wenn `unreliableActivity` eine Ausnahme ausgibt, ruft das Framework `doCatch` auf und übergibt es an das Ausnahmeobjekt. `doCatch` legt `retryActivity` auf "true" fest.
6. `runUnreliableActivityTillSuccess` ruft die asynchrone `restartRunUnreliableActivityTillSuccess`-Methode auf und übergibt ihr das `retryActivity`-Objekt. Da `retryActivity` ein `Promise<T>`-Typ ist, verschiebt `restartRunUnreliableActivityTillSuccess` die Ausführung, bis `retryActivity` einsatzbereit ist. Dies ist der Fall, sobald `TryCatch` abgeschlossen wird.
7. Wenn `retryActivity` einsatzbereit ist, extrahiert `restartRunUnreliableActivityTillSuccess` den Wert.

- Wenn der Wert `false` ist, war die Wiederholung erfolgreich. `restartRunUnreliableActivityTillSuccess` unternimmt nichts und die Wiederholungssequenz wird beendet.
- Wenn als Wert "true" ausgegeben wird, ist der Wiederholungsversuch fehlgeschlagen. `restartRunUnreliableActivityTillSuccess` ruft `runUnreliableActivityTillSuccess` auf, um die Aktivität erneut auszuführen.

8. Die Schritte 1 bis 7 werden wiederholt, bis `unreliableActivity` abgeschlossen wird.

> **Note**
>
> `doCatch` verarbeitet die Ausnahme nicht, sondern legt nur das `retryActivity`-Objekt auf "true" fest, um anzugeben, dass die Aktivität fehlgeschlagen ist. Die Wiederholung wird von der asynchronen `restartRunUnreliableActivityTillSuccess`-Methode verarbeitet, die die Ausführung verschiebt, bis `TryCatch` abgeschlossen wird. Der Grund für diesen Ansatz ist, dass Sie eine Aktivität, die Sie in `doCatch` wiederholen, nicht beenden können. Wenn die Aktivität in `restartRunUnreliableActivityTillSuccess` wiederholt wird, können Sie Aktivitäten ausführen, die sich beenden lassen.

Exponentielle Wiederholungsstrategie

Bei der exponentiellen Wiederholungsstrategie führt das Framework eine fehlgeschlagene Aktivität nach einem festgelegten Zeitraum (N Sekunden) erneut aus. Schlägt dieser Versuch fehl, wiederholt das Framework die Aktivität nach 2N Sekunden, dann nach 4N Sekunden usw. Da die Wartezeit sehr lang werden kann, werden Sie die Wiederholungen nicht endlos fortsetzen, sondern den Vorgang irgendwann beenden.

Das Framework bietet drei Möglichkeiten zur Implementierung einer exponentiellen Wiederholungsstrategie:

- Die `@ExponentialRetry`-Anmerkung ist der einfachste Ansatz. Sie müssen die Wiederholungsoptionen jedoch bei der Kompilierung festlegen.
- Die `RetryDecorator`-Klasse ermöglicht es Ihnen, die Wiederholungskonfiguration zur Laufzeit festzulegen und bei Bedarf zu ändern.
- Die `AsyncRetryingExecutor`-Klasse ermöglicht es Ihnen, die Wiederholungskonfiguration zur Laufzeit festzulegen und bei Bedarf zu ändern. Darüber hinaus ruft das Framework eine vom

Benutzer implementierte `AsyncRunnable.run`-Methode zur Ausführung jedes neuen Versuchs auf.

Alle Ansätze unterstützen folgende Konfigurationsoptionen, wobei die Werte für die Zeit in Sekunden angegeben werden:

- Die erste Wiederholungswartezeit.
- Den Backoff-Koeffizienten, der verwendet wird, um die Wiederholungsintervalle folgendermaßen zu errechnen:

```
retryInterval = initialRetryIntervalSeconds * Math.pow(backoffCoefficient,
 numberOfTries - 2)
```

  Der Standardwert lautet 2.0.
- Die maximale Anzahl an Wiederholungen. Der Standardwert ist unbegrenzt.
- Das maximale Wiederholungsintervall. Der Standardwert ist unbegrenzt.
- Die Ablaufzeit. Es werden keine Wiederholungen mehr ausgeführt, wenn die Gesamtdauer des Vorgangs diesen Wert übersteigt. Der Standardwert ist unbegrenzt.
- Die Ausnahmen, die den Wiederholungsvorgang auslösen. Standardmäßig löst jede Ausnahme den Wiederholungsvorgang aus.
- Die Ausnahmen, die keinen Wiederholungsvorgang auslösen. Standardmäßig sind keine Ausnahmen ausgeschlossen.

In den folgenden Abschnitten werden die verschiedenen Methoden zur Implementierung einer exponentiellen Wiederholungsstrategie beschrieben.

Exponentieller Wiederholungsversuch mit @ ExponentialRetry

Die einfachste Möglichkeit zur Implementierung einer exponentiellen Wiederholungsstrategie für eine Aktivität ist die Anwendung einer `@ExponentialRetry` -Anmerkung auf die Aktivität in der Schnittstellendefinition. Schlägt die Aktivität fehl, verarbeitet das Framework den Wiederholungsvorgang automatisch basierend auf den festgelegten Optionen. Das grundlegende Muster ist:

1. Wenden Sie `@ExponentialRetry` auf die entsprechenden Aktivitäten an und legen Sie die Wiederholungskonfiguration fest.

2. Schlägt eine mit einer Anmerkung versehene Aktivität fehl, wiederholt das Framework die Aktivität automatisch basierend auf der durch die Anmerkungsargumente festgelegten Konfiguration.

Der `ExponentialRetryAnnotationWorkflow`-Workflow-Worker implementiert die exponentielle Wiederholungsstrategie durch Verwendung einer `@ExponentialRetry`-Anmerkung. Er verwendet eine `unreliableActivity`-Aktivität, deren Schnittstellendefinition wie folgt in `ExponentialRetryAnnotationActivities` implementiert wird:

```
@Activities(version = "1.0")
@ActivityRegistrationOptions(
    defaultTaskScheduleToStartTimeoutSeconds = 30,
    defaultTaskStartToCloseTimeoutSeconds = 30)
public interface ExponentialRetryAnnotationActivities {
    @ExponentialRetry(
        initialRetryIntervalSeconds = 5,
        maximumAttempts = 5,
        exceptionsToRetry = IllegalStateException.class)
    public void unreliableActivity();
}
```

Die `@ExponentialRetry`-Optionen legen folgende Strategie fest:

- Nur wiederholen, wenn die Aktivität `IllegalStateException` ausgibt.
- Eine erste Wartezeit von 5 Sekunden verwenden.
- Nicht mehr als 5 Wiederholungen.

Die Workflow-Schnittstelle wird in `RetryWorkflow` implementiert und enthält die Methode `process`, die den Eintrittspunkt des Workflows darstellt. Der Workflow-Worker wird in `ExponentialRetryAnnotationWorkflowImpl` wie folgt implementiert:

```
public class ExponentialRetryAnnotationWorkflowImpl implements RetryWorkflow {
    public void process() {
        handleUnreliableActivity();
    }

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

Der Workflow funktioniert folgendermaßen:

1. `process` führt die synchrone `handleUnreliableActivity`-Methode aus.
2. `handleUnreliableActivity` führt die `unreliableActivity`-Aktivität aus.

Schlägt die Aktivität fehl, indem `IllegalStateException` ausgegeben wird, führt das Framework automatisch die in `ExponentialRetryAnnotationActivities` festgelegte Wiederholungsstrategie aus.

Exponentielle Wiederholung mit der Klasse RetryDecorator

`@ExponentialRetry` ist benutzerfreundlich. Allerdings ist die Konfiguration statisch und wird bei der Kompilierung festgelegt, sodass das Framework bei jedem Fehler der Aktivität dieselbe Wiederholungsstrategie anwendet. Sie können eine flexiblere exponentielle Wiederholungsstrategie implementieren, indem Sie die `RetryDecorator`-Klasse verwenden, mit der Sie die Konfiguration zur Laufzeit festlegen und bei Bedarf ändern können. Das grundlegende Muster ist:

1. Erzeugen und konfigurieren Sie ein `ExponentialRetryPolicy`-Objekt, das die Wiederholungskonfiguration festlegt.
2. Erzeugen Sie ein `RetryDecorator`-Objekt und geben Sie das `ExponentialRetryPolicy`-Objekt aus Schritt 1 an den Konstruktor weiter.
3. Wenden Sie das Decorator-Objekt auf die Aktivität an, indem Sie den Klassennamen des Aktivitäts-Clients auf die Ausstattungsmethode des `RetryDecorator`-Objekts übergeben.
4. Führen Sie die Aktivität aus.

Schlägt die Aktivität fehl, wiederholt das Framework die Aktivität basierend auf der `ExponentialRetryPolicy`-Objektkonfiguration. Sie können die Wiederholungskonfiguration bei Bedarf ändern, indem Sie dieses Objekt anpassen.

> **Note**
>
> Die `@ExponentialRetry`-Anmerkung und die `RetryDecorator`-Klasse schließen sich gegenseitig aus. Sie können `RetryDecorator` nicht verwenden, um eine Wiederholungsrichtlinie, die von einer `@ExponentialRetry`-Anmerkung festgelegt wird, dynamisch zu überschreiben.

Die folgende Workflow-Implementierung zeigt, wie die `RetryDecorator`-Klasse verwendet wird, um eine exponentielle Wiederholungsstrategie zu implementieren. Sie verwendet eine `unreliableActivity`-Aktivität, die nicht über eine `@ExponentialRetry`-Anmerkung verfügt. Die Workflow-Schnittstelle wird in `RetryWorkflow` implementiert und enthält die Methode `process`, die den Eintrittspunkt des Workflows darstellt. Der Workflow-Worker wird in `DecoratorRetryWorkflowImpl` wie folgt implementiert:

```
public class DecoratorRetryWorkflowImpl implements RetryWorkflow {
   ...
  public void process() {
      long initialRetryIntervalSeconds = 5;
      int maximumAttempts = 5;
      ExponentialRetryPolicy retryPolicy = new ExponentialRetryPolicy(
              initialRetryIntervalSeconds).withMaximumAttempts(maximumAttempts);

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

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

Der Workflow funktioniert folgendermaßen:

1. `process` erzeugt und konfiguriert ein `ExponentialRetryPolicy`-Objekt folgendermaßen:
   - Das erste Wiederholungsintervall wird an den Konstruktor übergeben.
   - Aufrufen der `withMaximumAttempts`-Methode des Objekts, um die maximale Anzahl der Versuche auf 5 festzulegen. `ExponentialRetryPolicy` zeigt andere `with`-Objekte an, mit denen Sie andere Konfigurationsoptionen angeben können.
2. `process` erzeugt ein `RetryDecorator`-Objekt namens `retryDecorator` und übergibt das `ExponentialRetryPolicy`-Objekt aus Schritt 1 an den Konstruktor.
3. `process` wendet den Decorator auf die Aktivität an, indem es die `retryDecorator.decorate`-Methode aufruft und ihr den Klassennamen des Aktivitäts-Clients übergibt.
4. `handleUnreliableActivity` führt die Aktivität aus.

Schlägt die Aktivität fehl, wiederholt das Framework sie basierend auf der in Schritt 1 festgelegten Konfiguration.

> **Note**
>
> Einige `with`-Methoden der `ExponentialRetryPolicy`-Klasse besitzen eine entsprechende `set`-Methode, die Sie jederzeit aufrufen können, um die entsprechende Konfigurationsoption anzupassen: `setBackoffCoefficient`, `setMaximumAttempts`, `setMaximumRetryIntervalSeconds` und `setMaximumRetryExpirationIntervalSeconds`.

Exponentielle Wiederholung mit der Klasse AsyncRetryingExecutor

Die `RetryDecorator`-Klasse bietet mehr Flexibilität bei der Konfiguration des Wiederholungsvorgangs als `@ExponentialRetry`, allerdings führt das Framework dennoch automatisch die Wiederholungen basierend auf der aktuellen Konfiguration des `ExponentialRetryPolicy`-Objekts aus. Ein flexiblerer Ansatz ist die Verwendung der `AsyncRetryingExecutor`-Klasse. Sie haben nicht nur die Möglichkeit, den Wiederholungsvorgang zur Laufzeit zu konfigurieren, sondern das Framework ruft zudem eine vom Benutzer implementierte `AsyncRunnable.run`-Methode zur Ausführung jeder Wiederholung auf, statt die Aktivität einfach auszuführen.

Das grundlegende Muster ist:

1. Erzeugen und konfigurieren Sie ein `ExponentialRetryPolicy`-Objekt, um die Wiederholungskonfiguration festzulegen.
2. Erzeugen Sie ein `AsyncRetryingExecutor`-Objekt und übergeben Sie ihm das `ExponentialRetryPolicy`-Objekt und eine Instanz der Workflow-Uhr.
3. Implementieren Sie eine anonyme verschachtelte `TryCatch`- oder `TryCatchFinally`-Klasse.
4. Implementieren Sie eine anonyme `AsyncRunnable`-Klasse und überschreiben Sie die `run`-Methode, um den benutzerdefinierten Code zur Ausführung der Aktivität zu implementieren.
5. Überschreiben Sie `doTry`, um die `execute`-Methode des `AsyncRetryingExecutor`-Objekts aufzurufen, und übergeben Sie ihr die `AsyncRunnable`-Klasse aus Schritt 4. Das `AsyncRetryingExecutor`-Objekt ruft `AsyncRunnable.run` auf, um die Aktivität auszuführen.

6. Schlägt die Aktivität fehl, ruft das `AsyncRetryingExecutor`-Objekt in Einklang mit der Wiederholungsrichtlinie, die in Schritt 1 festgelegt wurde, die `AsyncRunnable.run`-Methode erneut auf.

Der folgende Workflow zeigt, wie die `AsyncRetryingExecutor`-Klasse verwendet wird, um eine exponentielle Wiederholungsstrategie zu implementieren. Er verwendet dieselbe `unreliableActivity`-Aktivität wie der zuvor behandelte `DecoratorRetryWorkflow`-Workflow. Die Workflow-Schnittstelle wird in `RetryWorkflow` implementiert und enthält die Methode `process`, die den Eintrittspunkt des Workflows darstellt. Der Workflow-Worker wird in `AsyncExecutorRetryWorkflowImpl` wie folgt implementiert:

```
public class AsyncExecutorRetryWorkflowImpl implements RetryWorkflow {
  private final RetryActivitiesClient client = new RetryActivitiesClientImpl();
  private final DecisionContextProvider contextProvider = new
 DecisionContextProviderImpl();
  private final WorkflowClock clock =
 contextProvider.getDecisionContext().getWorkflowClock();

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

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

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

```
            protected void doCatch(Throwable e) throws Throwable {
            }
        };
    }
}
```

Der Workflow funktioniert folgendermaßen:

1. `process` ruft die `handleUnreliableActivity`-Methode auf und übergibt ihr die Konfigurationseinstellungen.
2. `handleUnreliableActivity` verwendet die Konfigurationseinstellungen aus Schritt 1, um das `ExponentialRetryPolicy`-Objekt `retryPolicy` zu erzeugen.
3. `handleUnreliableActivity` erzeugt das `AsyncRetryExecutor`-Objekt `executor` und übergibt das `ExponentialRetryPolicy`-Objekt aus Schritt 2 und eine Instanz der Workflow-Uhr an den Konstruktor.
4. `handleUnreliableActivity` implementiert eine anonyme verschachtelte `TryCatch`-Klasse und überschreibt die `doTry`- und `doCatch`-Methode, um die Wiederholungen auszuführen und mögliche Ausnahmen zu verarbeiten.
5. `doTry` erzeugt eine anonyme `AsyncRunnable`-Klasse und überschreibt die `run`-Methode, um den benutzerdefinierten Code zur Ausführung von `unreliableActivity` zu implementieren. Der Einfachheit halber führt `run` nur die Aktivität aus, Sie können bei Bedarf jedoch komplexere Ansätze implementieren.
6. `doTry` ruft `executor.execute` auf und übergibt es an das `AsyncRunnable`-Objekt. `execute` ruft die `run`-Methode des `AsyncRunnable`-Objekts auf, um die Aktivität auszuführen.
7. Schlägt die Aktivität fehl, ruft der Executor erneut `run` auf, basierend auf der Konfiguration des `retryPolicy`-Objekts.

Weitere Informationen zur Verwendung der `TryCatch`-Klasse zur Fehlerbehandlung finden Sie unter AWS Flow Framework für Java-Ausnahmen.

Benutzerdefinierte Wiederholungsstrategie

Der flexibelste Ansatz zur Wiederholung fehlgeschlagener Aktivitäten ist eine benutzerdefinierte Strategie, bei der rekursiv eine asynchrone Methode aufgerufen wird, die den Wiederholungsversuch ausführt, ähnlich wie bei der Strategie. retry-until-success Statt die Aktivität einfach erneut auszuführen, implementieren Sie jedoch eine benutzerdefinierte Logik, die entscheidet, ob und wie jede Wiederholung ausgeführt werden soll. Das grundlegende Muster ist:

1. Erzeugen Sie ein `Settable<T>`-Statusobjekt, das verwendet wird, um anzugeben, ob die Aktivität fehlgeschlagen ist.
2. Implementieren Sie eine verschachtelte `TryCatch`- oder `TryCatchFinally`-Klasse.
3. `doTry` führt die Aktivität aus.
4. Schlägt die Aktivität fehl, legt `doCatch` das Statusobjekt fest, um anzugeben, dass die Aktivität fehlgeschlagen ist.
5. Rufen Sie eine asynchrone Fehlerbehandlungsmethode auf und übergeben Sie ihr das Statusobjekt. Die Methode verschiebt die Ausführung, bis `TryCatch` oder `TryCatchFinally` abgeschlossen wird.
6. Die Fehlerbehandlungsmethode entscheidet, ob und wann die Aktivität wiederholt werden soll.

Der folgende Workflow zeigt, wie eine benutzerdefinierte Wiederholungsstrategie implementiert wird. Er verwendet dieselbe `unreliableActivity`-Aktivität wie der `DecoratorRetryWorkflow`- und `AsyncExecutorRetryWorkflow`-Workflow. Die Workflow-Schnittstelle wird in `RetryWorkflow` implementiert und enthält die Methode `process`, die den Eintrittspunkt des Workflows darstellt. Der Workflow-Worker wird in `CustomLogicRetryWorkflowImpl` wie folgt implementiert:

```
public class CustomLogicRetryWorkflowImpl implements RetryWorkflow {
  ...
  public void process() {
      callActivityWithRetry();
  }
  @Asynchronous
  public void callActivityWithRetry() {
      final Settable<Throwable> failure = new Settable<Throwable>();
      new TryCatchFinally() {
          protected void doTry() throws Throwable {
              client.unreliableActivity();
          }
          protected void doCatch(Throwable e) {
              failure.set(e);
          }
          protected void doFinally() throws Throwable {
              if (!failure.isReady()) {
                  failure.set(null);
              }
          }
      };
      retryOnFailure(failure);
```

```
  }
  @Asynchronous
  private void retryOnFailure(Promise<Throwable> failureP) {
      Throwable failure = failureP.get();
      if (failure != null && shouldRetry(failure)) {
          callActivityWithRetry();
      }
  }
  protected Boolean shouldRetry(Throwable e) {
      //custom logic to decide to retry the activity or not
      return true;
  }
}
```

Der Workflow funktioniert folgendermaßen:

1. `process` ruft die asynchrone `callActivityWithRetry`-Methode auf.
2. `callActivityWithRetry` erstellt ein `Settable<Throwable>`-Objekt namens "failure" (Fehler), mit dem angezeigt wird, dass die Aktivität fehlgeschlagen ist. `Settable<T>` ist von `Promise<T>` abgeleitet und funktioniert zwar ähnlich, jedoch legen Sie den Wert eines `Settable<T>`-Objekts manuell fest.
3. `callActivityWithRetry` implementiert eine anonyme verschachtelte `TryCatchFinally`-Klasse zur Verarbeitung von Ausnahmen, die von `unreliableActivity` ausgegeben werden. Weitere Informationen zum Umgang mit Ausnahmen, die von asynchronem Code ausgegeben werden, finden Sie unter AWS Flow Framework für Java-Ausnahmen.
4. `doTry` führt `unreliableActivity` aus.
5. Wenn `unreliableActivity` eine Ausnahme auslöst, ruft das Framework `doCatch` auf und übergibt sie an das Ausnahmeobjekt. `doCatch` legt `failure` auf das Ausnahmeobjekt fest, was anzeigt, dass die Aktivität fehlgeschlagen ist, und versetzt das Objekt in einen betriebsbereiten Zustand.
6. `doFinally` überprüft, ob `failure` einsatzbereit ist, was nur der Fall ist, wenn `failure` von `doCatch` festgelegt wurde.
   - Wenn es bereit `failure` ist, tut es nichts. `doFinally`
   - Wenn `failure` nicht einsatzbereit ist, wird die Aktivität abgeschlossen und `doFinally` legt "failure" auf `null` fest.
7. `callActivityWithRetry` ruft die asynchrone `retryOnFailure`-Methode auf und übergibt ihr "failure". Da "failure" ein `Settable<T>`-Typ ist, verschiebt `callActivityWithRetry`

die Ausführung, bis "failure" einsatzbereit ist. Dies ist der Fall, sobald `TryCatchFinally` abgeschlossen wird.

8. `retryOnFailure` ruft den Wert von "failure" ab.
   - Wenn der Fehler auf Null gesetzt ist, war der Wiederholungsversuch erfolgreich. `retryOnFailure` unternimmt nichts, wodurch der Wiederholungsprozess beendet wird.
   - Wenn "failure" auf ein Ausnahmeobjekt festgelegt ist und `shouldRetry` "true" zurückgibt, ruft `retryOnFailure` `callActivityWithRetry` auf, um die Aktivität zu wiederholen.

     `shouldRetry` implementiert eine benutzerdefinierte Logik, um zu entscheiden, ob eine fehlgeschlagene Aktivität wiederholt werden soll. Der Einfachheit halber gibt `shouldRetry` immer `true` zurück und `retryOnFailure` führt die Aktivität sofort aus, Sie können bei Bedarf jedoch eine komplexere Logik implementieren.
9. Die Schritte 2—8 werden wiederholt, bis der Vorgang `unreliableActivity` abgeschlossen ist oder `shouldRetry` beendet werden soll.

> **Note**
>
> `doCatch` verarbeitet den Wiederholungsvorgang nicht, sondern legt nur "failure" fest, um anzugeben, dass die Aktivität fehlgeschlagen ist. Der Wiederholungsvorgang wird von der asynchronen `retryOnFailure`-Methode verarbeitet, die die Ausführung verschiebt, bis `TryCatch` abgeschlossen wird. Der Grund für diesen Ansatz ist, dass Sie eine Aktivität, die Sie in `doCatch` wiederholen, nicht beenden können. Wenn die Aktivität in `retryOnFailure` wiederholt wird, können Sie Aktivitäten ausführen, die sich beenden lassen.

Daemon-Aufgaben

Das AWS Flow Framework für Java ermöglicht das Markieren bestimmter Aufgaben als`daemon`. Mithilfe dieser Markierung können Sie Aufgaben zum Ausführen von Hintergrundroutinen erstellen, die abgebrochen werden sollen, wenn alle Routinen beendet sind. Eine Aufgabe zum Überwachen des Status soll beispielsweise abgebrochen werden, wenn der Rest des Workflows abgeschlossen ist. Legen Sie dazu das `daemon`-Flag für eine asynchrone Methode oder Instance von `TryCatchFinally` fest. Im folgenden Beispiel wird die asynchrone Methode `monitorHealth()` als `daemon` markiert.

```
public class MyWorkflowImpl implements MyWorkflow {
  MyActivitiesClient activitiesClient = new MyActivitiesClientImpl();

  @Override
  public void startMyWF(int a, String b) {
    activitiesClient.doUsefulWorkActivity();
    monitorHealth();
  }

  @Asynchronous(daemon=true)
  void monitorHealth(Promise<?>... waitFor) {
    activitiesClient.monitoringActivity();
  }
}
```

Im obigen Beispiel wird bei Abschluss von `doUsefulWorkActivity` `monitoringHealth` automatisch abgebrochen. Dadurch wird der gesamte Ausführungszweig, der aus dieser asynchronen Methode stammt, abgebrochen. Die Semantik dieses Abbruchs entspricht der in `TryCatchFinally`. Entsprechend können Sie einen `TryCatchFinally`-Daemon markieren, indem Sie ein boolesches Flag an den Konstruktor übergeben.

```
public class MyWorkflowImpl implements MyWorkflow {
    MyActivitiesClient activitiesClient = new MyActivitiesClientImpl();

    @Override
    public void startMyWF(int a, String b) {
        activitiesClient.doUsefulWorkActivity();
        new TryFinally(true) {
            @Override
            protected void doTry() throws Throwable {
                activitiesClient.monitoringActivity();
            }

            @Override
            protected void doFinally() throws Throwable {
                // clean up
            }
        };
    }
}
```

Eine Daemon-Aufgabe, die innerhalb einer gestartet wird, `TryCatchFinally` ist auf den Kontext beschränkt, in dem sie erstellt wurde, d. h. sie wird entweder auf die Methoden, oder beschränkt. `doTry()` `doCatch()` `doFinally()` Im folgenden Beispiel wird die asynchrone startMonitoring-Methode als Daemon markiert und von `doTry()` aufgerufen. Die dafür erstellte Aufgabe wird abgebrochen, sobald die anderen Aufgaben (`doUsefulWorkActivity` in diesem Fall), die in `doTry()` gestartet wurden, abgeschlossen sind.

```
public class MyWorkflowImpl implements MyWorkflow {
    MyActivitiesClient activitiesClient = new MyActivitiesClientImpl();

    @Override
    public void startMyWF(int a, String b) {
        new TryFinally() {
            @Override
            protected void doTry() throws Throwable {
                activitiesClient.doUsefulWorkActivity();
                startMonitoring();
            }

            @Override
            protected void doFinally() throws Throwable {
                // Clean up
            }
        };
    }

    @Asynchronous(daemon = true)
    void startMonitoring(){
      activitiesClient.monitoringActivity();
    }
```

AWS Flow Framework für Java Replay Behavior

In diesem Thema werden Beispiele für Replay-Verhalten unter Verwendung von Beispielen im Abschnitt Was ist das AWS Flow Framework für Java? erläutert. Sowohl synchrone als auch asynchrone Szenarien werden behandelt.

Beispiel 1: Synchrones Replay

Ein Beispiel dafür, wie die Wiedergabe in einem synchronen Workflow funktioniert, finden Sie, indem Sie die HelloWorldWorkflowWorkflow- und Aktivitätsimplementierungen wie folgt ändern, indem Sie innerhalb der jeweiligen Implementierungen `println` Aufrufe hinzufügen:

```
public class GreeterWorkflowImpl implements GreeterWorkflow {
...
   public void greet() {
      System.out.println("greet executes");
      Promise<String> name = operations.getName();
      System.out.println("client.getName returns");
      Promise<String> greeting = operations.getGreeting(name);
      System.out.println("client.greeting returns");
      operations.say(greeting);
      System.out.println("client.say returns");
   }
}
**************
public class GreeterActivitiesImpl implements GreeterActivities {
   public String getName() {
      System.out.println("activity.getName completes");
      return "World";
   }

   public String getGreeting(String name) {
      System.out.println("activity.getGreeting completes");
      return "Hello " + name + "!";
   }

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

Details zum Code finden Sie unter HelloWorldWorkflow Bewerbung. Im Folgenden sehen Sie eine bearbeitete Version der Ausgabe mit Kommentaren, die den Start jedes Replay-Abschnitts angeben.

```
//Episode 1
greet executes
client.getName returns
client.greeting returns
```

```
client.say returns

activity.getName completes
//Episode 2
greet executes
client.getName returns
client.greeting returns
client.say returns

activity.getGreeting completes
//Episode 3
greet executes
client.getName returns
client.greeting returns
client.say returns

Hello World! //say completes
//Episode 4
greet executes
client.getName returns
client.greeting returns
client.say returns
```

Der Replay-Prozess für dieses Beispiel funktioniert wie folgt:

- Im ersten Abschnitt wird die `getName`-Aktivitätsaufgabe geplant, die keine Abhängigkeiten hat.
- Im zweiten Abschnitt wird die `getGreeting`-Aktivitätsaufgabe geplant, die von `getName` abhängt.
- Im dritten Abschnitt wird die `say`-Aktivitätsaufgabe geplant, die von `getGreeting` abhängt.
- Im letzten Abschnitt werden keine zusätzlichen Aufgaben geplant und keine nicht abgeschlossenen Aktivitäten gefunden, wodurch die Workflow-Ausführung beendet wird.

> **Note**
>
> Die drei Aktivitäten-Client-Methoden werden einmal für jeden Abschnitt aufgerufen. Allerdings ergibt sich nur aus einem dieser Aufrufe eine Aktivitätsaufgabe, sodass jede Aufgabe nur einmal durchgeführt wird.

Beispiel 2: Asynchrones Replay

Ähnlich wie im Beispiel für synchrones Replay können Sie HelloWorldWorkflowAsyncBewerbung ändern, um zu sehen, wie ein asynchrones Replay funktioniert. Es erzeugt folgende Ausgabe:

```
//Episode 1
greet executes
client.name returns
workflow.getGreeting returns
client.say returns

activity.getName completes
//Episode 2
greet executes
client.name returns
workflow.getGreeting returns
client.say returns
workflow.getGreeting completes

Hello World! //say completes
//Episode 3
greet executes
client.name returns
workflow.getGreeting returns
client.say returns
workflow.getGreeting completes
```

HelloWorldAsync verwendet drei Wiederholungsepisoden, da es nur zwei Aktivitäten gibt. Die `getGreeting`-Aktivität wurde durch die asynchrone Workflow-Methode getGreeting ersetzt, die keinen Replay-Abschnitt initiiert, wenn sie abgeschlossen wird.

Der erste Abschnitt ruft `getGreeting` nicht auf, da er vom Abschluss der Aktivität name abhängt. Aber nachdem getName abgeschlossen wurde, ruft Replay getGreeting einmal für jeden nachfolgenden Abschnitt auf.

Weitere Informationen finden Sie unter:

- AWS Flow Framework Grundbegriffe: Verteilte Ausführung

Bewährte Methoden

Verwenden Sie diese bewährten Methoden, um das Beste aus dem AWS Flow Framework für Java herauszuholen.

Themen

- Vornehmen von Änderungen am Entscheidercode: Versioning und Funktions-Flags

Vornehmen von Änderungen am Entscheidercode: Versioning und Funktions-Flags

In diesem Abschnitt erfahren Sie, wie Sie Änderungen am Entscheidercode vornehmen, um die Abwärtskompatibilität sicherzustellen. Hierfür haben Sie zwei Möglichkeiten:

- Versioning ist eine grundlegende Lösung.
- Version mit Funktions-Flags baut auf reinem Versioning auf: Es wird keine neue Version des Workflows eingeführt und für die Versionsaktualisierung ist kein neuer Code erforderlich.

Bevor Sie diese Lösungen ausprobieren, sollten Sie sich mit dem Abschnitt Beispielszenario vertraut machen. Dort werden die Ursachen und Auswirkungen von Änderungen am Entscheidercode erläutert, die zu Abwärtsinkompatibilität führen.

Wiedergabe und Codeänderungen

Wenn ein Decider-Worker AWS Flow Framework für Java eine Entscheidungsaufgabe ausführt, muss er zunächst den aktuellen Status der Ausführung neu erstellen, bevor er weitere Schritte hinzufügen kann. Der Entscheider verwendet hierfür die sogenannte Wiedergabe.

Beim Wiedergabeprozess wird der Entscheidercode von Anfang an erneut ausgeführt und gleichzeitig der Ereignisverlauf durchgegangen. Dadurch kann das Framework auf Signale oder den Abschluss einer Aufgabe reagieren und `Promise`-Objekte im Code freigeben.

Wenn das Framework den Decider-Code ausführt, weist es jeder geplanten Aufgabe (einer Aktivität, Lambda-Funktion, einem Timer, einem untergeordneten Workflow oder einem ausgehenden Signal) eine ID zu, indem es einen Zähler erhöht. Das Framework übermittelt diese ID an Amazon SWF und fügt die ID zu Verlaufsereignissen hinzu, z. `ActivityTaskCompleted`

Damit der Wiedergabeprozess erfolgreich ist, muss der Entscheidercode deterministisch sein und dieselben Aufgaben für jede Entscheidung bei jeder Workflow-Ausführung in derselben Reihenfolge planen. Wenn diese Anforderung nicht erfüllt ist, kann es beispielsweise vorkommen, dass das Framework die ID in einem `ActivityTaskCompleted`-Ereignis nicht einem vorhandenen `Promise`-Objekt zuordnen kann.

Beispielszenario

Es gibt eine Klasse von Codeänderungen, die als abwärtsinkompatibel gilt. Zu diesen Änderungen gehören Aktualisierungen, bei denen die Anzahl, der Typ oder die Reihenfolge der geplanten Aufgaben verändert werden. Betrachten Sie das folgende Beispiel:

Sie schreiben Entscheidercode, um zwei Timer-Aufgaben zu planen. Sie beginnen mit der Ausführung und führen eine Entscheidung aus. Daher werden zwei Timer-Aufgaben mit IDs 1 und 2 geplant.

Wenn Sie den Entscheidercode so aktualisieren, dass nur ein Timer geplant wird, bevor die nächste Entscheidung ausgeführt wird, kann das Framework bei der nächsten Entscheidungsaufgabe das zweite `TimerFired`-Ereignis nicht wiedergeben, da die ID 2 nicht mit einer Timer-Aufgabe übereinstimmt, die vom Code erzeugt wurde.

Überblick über das Szenario

Der folgende Überblick zeigt die einzelnen Schritte dieses Szenarios. Endziel des Szenarios ist es, eine Migration auf ein System durchzuführen, bei dem nur ein Timer geplant wird, ohne dass dies dazu führt, dass Ausführungen fehlschlagen, die vor der Migration gestartet wurden.

1. Die erste Entscheiderversion
   a. Schreiben Sie den Entscheider.
   b. Starten Sie den Entscheider.
   c. Der Entscheider plant zwei Timer.
   d. Der Entscheider startet fünf Ausführungen.
   e. Halten Sie den Entscheider an.
2. Eine abwärtsinkompatible Änderung am Entscheider
   a. Ändern Sie den Entscheider.
   b. Starten Sie den Entscheider.
   c. Der Entscheider plant einen Timer.

d. Der Entscheider startet fünf Ausführungen.

Die folgenden Abschnitte enthalten Java-Beispielcode, mit dem sich dieses Szenario implementieren lässt. Die Codebeispiele im Abschnitt Lösungen zeigen zwei Möglichkeiten, abwärtsinkompatible Änderungen zu beheben.

> **Note**
>
> Sie können die aktuelle Version von AWS SDK für Java verwenden, um diesen Code auszuführen.

Gängiger Code

Der folgende Java-Code ändert sich zwischen den Beispielen in diesem Szenario nicht.

`SampleBase.java`

```
package sample;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
import com.amazonaws.services.simpleworkflow.flow.JsonDataConverter;
import com.amazonaws.services.simpleworkflow.model.DescribeWorkflowExecutionRequest;
import com.amazonaws.services.simpleworkflow.model.DomainAlreadyExistsException;
import com.amazonaws.services.simpleworkflow.model.RegisterDomainRequest;
import com.amazonaws.services.simpleworkflow.model.Run;
import com.amazonaws.services.simpleworkflow.model.StartWorkflowExecutionRequest;
import com.amazonaws.services.simpleworkflow.model.TaskList;
import com.amazonaws.services.simpleworkflow.model.WorkflowExecution;
import com.amazonaws.services.simpleworkflow.model.WorkflowExecutionDetail;
import com.amazonaws.services.simpleworkflow.model.WorkflowType;

public class SampleBase {

    protected String domain = "DeciderChangeSample";
    protected String taskList = "DeciderChangeSample-" + UUID.randomUUID().toString();
```

```
   protected AmazonSimpleWorkflow service =
AmazonSimpleWorkflowClientBuilder.defaultClient();
   {
       try {
           AmazonSimpleWorkflowClientBuilder.defaultClient().registerDomain(new
RegisterDomainRequest().withName(domain).withDescription("desc").withWorkflowExecutionRetentio
       } catch (DomainAlreadyExistsException e) {
       }
   }

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

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

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

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

```
                    sleep(1000);
                    continue loop;
                }
            }
            return;
        }
    }

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

}
```

Input.java

```
package sample;

public class Input {

    private Boolean skipSecondTimer;

    public Input() {
    }

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

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

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

}
```

Schreiben des ersten Entscheidercodes

Nachfolgend sehen Sie den ersten Java-Code des Entscheiders. Er ist als Version 1 registriert und plant zwei fünfsekündige Timer-Aufgaben.

`InitialDecider.java`

```
package sample.v1;

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

import sample.Input;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
 defaultTaskStartToCloseTimeoutSeconds = 5)
public interface Foo {

    @Execute(version = "1")
    public void sample(Input input);

    public static class Impl implements Foo {

        private DecisionContext decisionContext = new
 DecisionContextProviderImpl().getDecisionContext();
        private WorkflowClock clock = decisionContext.getWorkflowClock();

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

    }
}
```

Simulieren einer abwärtsinkompatiblen Änderung

Der folgende, modifizierte Java-Code des Entscheiders ist ein gutes Beispiel für eine abwärtsinkompatible Änderung. Der Code ist weiterhin als Version 1 registriert, plant jedoch nur noch einen Timer.

`ModifiedDecider.java`

```
package sample.v1.modified;

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

import sample.Input;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
 defaultTaskStartToCloseTimeoutSeconds = 5)
public interface Foo {

    @Execute(version = "1")
    public void sample(Input input);

    public static class Impl implements Foo {

        private DecisionContext decisionContext = new
 DecisionContextProviderImpl().getDecisionContext();
        private WorkflowClock clock = decisionContext.getWorkflowClock();

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

    }
}
```

Mit dem folgenden Java-Code können Sie das Problem einer abwärtsinkompatiblen Änderung simulieren, indem Sie den modifizierten Entscheider ausführen.

`RunModifiedDecider.java`

```
package sample;

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

public class BadChange extends SampleBase {

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

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

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

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

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

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

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

        printExecutionResults();
    }

}
```

Wenn Sie das Programm ausführen, scheitern die drei Ausführungen, die mit der ersten Version des Entscheiders gestartet und nach der Migration fortgesetzt wurden.

Lösungen

Verwenden Sie eine der folgenden Lösungen, um abwärtsinkompatible Änderungen zu vermeiden. Weitere Informationen finden Sie unter Vornehmen von Änderungen am Entscheidercode und Beispielszenario.

Verwenden von Versioning

Für diese Lösung kopieren Sie den Entscheider in eine neue Klasse, modifizieren ihn und registrieren den Entscheider dann unter einer neuen Workflow-Version.

`VersionedDecider.java`

```
package sample.v2;

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

import sample.Input;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
 defaultTaskStartToCloseTimeoutSeconds = 5)
public interface Foo {

    @Execute(version = "2")
    public void sample(Input input);

    public static class Impl implements Foo {

        private DecisionContext decisionContext = new
 DecisionContextProviderImpl().getDecisionContext();
        private WorkflowClock clock = decisionContext.getWorkflowClock();
```

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

    }

}
```

Im aktualisierten Java-Code führt der zweite Entscheiderauftragnehmer beide Versionen des Workflows aus. Dadurch können laufende Ausführungen unabhängig von den Änderungen in Version 2 fortgesetzt werden.

RunVersionedDecider.java

```
package sample;

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

public class VersionedChange extends SampleBase {

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

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

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

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

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

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

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

        printExecutionResults();
    }

}
```

Wenn Sie das Programm ausführen, werden alle Ausführungen erfolgreich abgeschlossen.

Verwenden von Funktions-Flags

Eine weitere Lösung für Probleme mit der Abwärtskompatibilität besteht darin, den Code basierend auf Eingabedaten anstelle von Workflow-Versionen in zwei Implementierungen in derselben Klasse aufzuteilen.

Wenn Sie diesen Ansatz wählen, fügen Sie Ihren Eingabeobjekten jedes Mal, wenn Sie sensible Änderungen vornehmen, Felder hinzu (oder modifizieren vorhandene Felder Ihrer Eingabeobjekte). Für Ausführungen, die vor der Migration beginnen, enthält das Eingabeobjekt das Feld nicht (oder es enthält einen anderen Wert). Daher müssen Sie die Versionsnummer nicht erhöhen.

> **Note**
>
> Wenn Sie neue Felder hinzufügen, stellen Sie sicher, dass der JSON-Deserialisierungsprozess abwärtskompatibel ist. Objekte, die vor der Einführung des Felds serialisiert wurden, sollten nach der Migration weiterhin erfolgreich deserialisiert werden können. Da JSON einen `null`-Wert festlegt, wenn ein Feld fehlt, verwenden Sie grundsätzlich gepackte Typen (`Boolean` anstelle von `boolean`) und verarbeiten Sie Fälle, in denen der Wert `null` ist.

`FeatureFlagDecider.java`

```
package sample.v1.featureflag;

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

import sample.Input;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
 defaultTaskStartToCloseTimeoutSeconds = 5)
public interface Foo {

    @Execute(version = "1")
    public void sample(Input input);

    public static class Impl implements Foo {

        private DecisionContext decisionContext = new
 DecisionContextProviderImpl().getDecisionContext();
        private WorkflowClock clock = decisionContext.getWorkflowClock();

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

    }
}
```

Im aktualisierten Java-Code ist der Code beider Versionen des Workflows weiterhin für Version 1 registriert. Nach der Migration beginnen neue Ausführungen dagegen mit dem Feld `skipSecondTimer` der Eingabedaten mit dem Wert `true`.

RunFeatureFlagDecider.java

```
package sample;

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

public class FeatureFlagChange extends SampleBase {

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

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

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

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

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

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

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

        printExecutionResults();
    }

}
```

Wenn Sie das Programm ausführen, werden alle Ausführungen erfolgreich abgeschlossen.

Tipps zur Fehlerbehebung und zum Debuggen AWS Flow Framework für Java

Themen

- Fehler beim Kompilieren
- Unbekannter Ressourcenfehler
- Ausnahmen beim Aufrufen von get () für ein Promise
- Nichtdeterministische Workflows
- Probleme aufgrund der Versionierung
- Problembehandlung und Debuggen einer Workflow-Ausführung
- Verlorene Aufgaben
- Die Überprüfung ist aufgrund von Längenbeschränkungen für API-Parameter fehlgeschlagen

In diesem Abschnitt werden einige häufige Fallstricke beschrieben, auf die Sie bei der Entwicklung von Workflows mit AWS Flow Framework for Java stoßen könnten. Außerdem erhalten Sie einige Tipps, die Ihnen bei der Diagnose und der Behebung von Problemen helfen.

Fehler beim Kompilieren

Wenn Sie die AspectJ-Compile-Time-Weaving-Option verwenden, treten möglicherweise Kompilierzeitfehler auf, bei denen der Compiler die generierten Client-Klassen für Ihren Workflow und Ihre Aktivitäten nicht finden kann. Die wahrscheinliche Ursache solcher Kompilierfehler ist, dass der AspectJ-Builder die generierten Clients während der Kompilierung ignoriert hat. Sie können dieses Problem beheben, indem Sie die AspectJ-Funktion aus dem Projekt entfernen und erneut aktivieren. Beachten Sie, dass Sie dies jedes Mal durchführen müssen, wenn sich Ihre Workflow- oder Aktivitätsschnittstellen ändern. Aufgrund dieses Problems wird empfohlen, stattdessen die Load-Time-Weaving-Option zu verwenden. Weitere Details finden Sie im Abschnitt Einrichtung des AWS Flow Framework für Java.

Unbekannter Ressourcenfehler

Amazon SWF gibt einen unbekannten Ressourcenfehler zurück, wenn Sie versuchen, einen Vorgang mit einer Ressource durchzuführen, die nicht verfügbar ist. Die häufigen Ursachen für diesen Fehler sind:

- Sie konfigurieren einen Worker mit einer Domäne, die nicht vorhanden ist. Um dieses Problem zu beheben, registrieren Sie zunächst die Domain mit der Amazon SWF-Konsole oder der Amazon SWF-Service API.
- Sie versuchen, Workflow-Ausführungs- oder Aktivitätsaufgaben für Typen durchzuführen, die nicht registriert wurden. Dies kann passieren, wenn Sie versuchen, die Workflow-Ausführung zu erstellen, bevor die Worker ausgeführt wurden. Da Worker ihre Typen registrieren, wenn sie zum ersten Mal ausgeführt werden, müssen Sie sie mindestens einmal ausführen, bevor Sie versuchen, Ausführungen zu starten (oder die Typen manuell über die Konsole oder die Service-API registrieren). Beachten Sie, dass Sie, sobald Typen registriert wurden, Ausführungen auch dann erstellen können, wenn keine Worker ausgeführt werden.
- Ein Worker versucht, eine Aufgabe abzuschließen, die bereits das Zeitlimit überschritten hat. Wenn ein Worker beispielsweise zu lange für die Bearbeitung einer Aufgabe benötigt und ein Timeout überschreitet, wird ein UnknownResource Fehler angezeigt, wenn er versucht, die Aufgabe abzuschließen oder fehlschlägt. Die AWS Flow Framework Mitarbeiter werden weiterhin Amazon SWF abfragen und zusätzliche Aufgaben bearbeiten. Sie sollten jedoch in Betracht ziehen, die Zeitbeschränkung anzupassen. Zum Anpassen der Zeitbeschränkung müssen Sie eine neue Version des Aktivitätstyps registrieren.

Ausnahmen beim Aufrufen von get () für ein Promise

Anders als Java Future ist `Promise` ein blockierungsfreies Konstrukt und das Aufrufen von `get()` für ein noch nicht bereites `Promise`-Objekt führt zu einer Ausnahme anstelle einer Blockierung. Die korrekte Verwendung von a `Promise` besteht darin, es an eine asynchrone Methode (oder eine Aufgabe) zu übergeben und in der asynchronen Methode auf seinen Wert zuzugreifen. AWS Flow Framework for Java stellt sicher, dass eine asynchrone Methode nur aufgerufen wird, wenn alle an sie übergebenen `Promise` Argumente bereit sind. Wenn Sie glauben, dass Ihr Code korrekt ist, oder wenn Sie beim Ausführen eines der AWS Flow Framework Beispiele darauf stoßen, liegt dies höchstwahrscheinlich daran, dass AspectJ nicht richtig konfiguriert ist. Details hierzu finden Sie im Abschnitt Einrichtung des AWS Flow Framework für Java.

Nichtdeterministische Workflows

Wie im Abschnitt Nichtdeterminismus beschrieben, muss die Implementierung Ihres Workflows deterministisch sein. Einige häufige Fehler, die zu Nichtdeterminismus führen können, sind die Verwendung der Systemuhr, die Verwendung von Zufallszahlen und die Generierung von. GUIDs Da diese Konstrukte zu unterschiedlichen Zeiten unterschiedliche Werte zurückgeben können, kann die Ablaufsteuerung Ihres Workflows bei jeder Ausführung unterschiedliche Wege einschlagen (weitere Informationen finden Sie in den Abschnitten AWS Flow Framework Grundbegriffe: Verteilte Ausführung undEine Aufgabe in AWS Flow Framework für Java verstehen). Wenn das Framework während der Ausführung des Workflows Nichtdeterminismus erkennt, wird eine Ausnahme ausgelöst.

Probleme aufgrund der Versionierung

Wenn Sie eine neue Version Ihres Workflows oder Ihrer Aktivität implementieren, z. B. wenn Sie ein neues Feature hinzufügen, sollten Sie die Version des Typs erhöhen, indem Sie die entsprechende Anmerkung verwenden:, oder. `@Workflow` `@Activites` `@Activity` Wenn neue Versionen eines Workflows bereitgestellt werden, haben Sie oft Ausführungen der bestehenden Version, die bereits ausgeführt werden. Sie müssen daher sicherstellen, dass Worker mit der entsprechenden Version Ihres Workflows und Ihrer Aktivitäten die Aufgabe erhalten. Sie können dies erreichen, indem Sie verschiedene Aufgabenlisten für jede Version verwenden. Sie können beispielsweise die Versionsnummer an den Namen der Aufgabenliste anfügen. Dadurch wird sichergestellt, dass Aufgaben, die zu unterschiedlichen Versionen des Workflows und der Aktivitäten gehören, den entsprechenden Workern zugewiesen werden.

Problembehandlung und Debuggen einer Workflow-Ausführung

Der erste Schritt bei der Fehlerbehebung bei der Ausführung eines Workflows besteht darin, die Amazon SWF SWF-Konsole zu verwenden, um sich den Workflow-Verlauf anzusehen. Der Workflow-Verlauf ist ein kompletter und autoritativer Datensatz aller Ereignissen, die den Ausführungsstatus der Workflow-Ausführung geändert haben. Dieser Verlauf wird von Amazon SWF verwaltet und ist für die Diagnose von Problemen von unschätzbarem Wert. Mit der Amazon SWF SWF-Konsole können Sie nach Workflow-Ausführungen suchen und einzelne Verlaufsereignisse aufschlüsseln.

AWS Flow Framework bietet eine `WorkflowReplayer` Klasse, mit der Sie eine Workflow-Ausführung lokal wiedergeben und debuggen können. Mit dieser Klasse können Sie geschlossene und laufende Workflow-Ausführungen debuggen. `WorkflowReplayer`stützt sich auf den in

Amazon SWF gespeicherten Verlauf, um die Wiedergabe durchzuführen. Sie können es auf eine Workflow-Ausführung in Ihrem Amazon SWF-Konto verweisen oder es mit den Verlaufsereignissen versehen (Sie können beispielsweise den Verlauf von Amazon SWF abrufen und ihn für die spätere Verwendung lokal serialisieren). Wenn Sie eine Workflow-Ausführung mithilfe von `WorkflowReplayer` erneut abspielen, wirkt sich dies nicht auf die Workflow-Ausführung aus, die in Ihrem Konto ausgeführt wird. Das erneute Abspielen findet vollständig auf dem Client statt. Sie können wie gewohnt mithilfe Ihrer Debugging-Tools den Workflow debuggen, Haltepunkte erstellen und in den Code hineinzuspringen. Wenn Sie Eclipse verwenden, sollten Sie erwägen, Schrittfilter zum Filtern von AWS Flow Framework Paketen hinzuzufügen.

Der folgende Codeausschnitt beispielsweise kann verwendet werden, um eine Workflow-Ausführung erneut abzuspielen:

```
String workflowId = "testWorkflow";
String runId = "<run id>";
Class<HelloWorldImpl> workflowImplementationType = HelloWorldImpl.class;
WorkflowExecution workflowExecution = new WorkflowExecution();
workflowExecution.setWorkflowId(workflowId);
workflowExecution.setRunId(runId);

WorkflowReplayer<HelloWorldImpl> replayer = new WorkflowReplayer<HelloWorldImpl>(
    swfService, domain, workflowExecution, workflowImplementationType);

System.out.println("Beginning workflow replay for " + workflowExecution);
Object workflow = replayer.loadWorkflow();
System.out.println("Workflow implementation object:");
System.out.println(workflow);
System.out.println("Done workflow replay for " + workflowExecution);
```

AWS Flow Framework ermöglicht es Ihnen auch, einen asynchronen Thread-Dump Ihrer Workflow-Ausführung zu erstellen. Dieser Thread-Dump liefert Ihnen die Aufruflisten aller offenen asynchronen Aufgaben. Diese Informationen können beim Bestimmen, welche Aufgaben in der Ausführung noch ausstehen und möglicherweise hängen geblieben sind, helfen. Zum Beispiel:

```
String workflowId = "testWorkflow";
String runId = "<run id>";
Class<HelloWorldImpl> workflowImplementationType = HelloWorldImpl.class;
WorkflowExecution workflowExecution = new WorkflowExecution();
workflowExecution.setWorkflowId(workflowId);
workflowExecution.setRunId(runId);
```

```
WorkflowReplayer<HelloWorldImpl> replayer = new WorkflowReplayer<HelloWorldImpl>(
    swfService, domain, workflowExecution, workflowImplementationType);

try {
    String flowThreadDump = replayer.getAsynchronousThreadDumpAsString();
    System.out.println("Workflow asynchronous thread dump:");
    System.out.println(flowThreadDump);
}
catch (WorkflowException e) {
    System.out.println("No asynchronous thread dump available as workflow has failed: "
 + e);
}
```

Verlorene Aufgaben

Manchmal fahren Sie vielleicht in kurzer Abfolge Worker herunter und starten neue, nur um festzustellen, dass Aufgaben an dieselben alten Worker übermittelt werden. Dies kann aufgrund von Race Conditions im System passieren, das über mehrere Prozesse hinweg verteilt ist. Das Problem kann außerdem auftreten, wenn Sie Komponententests in einer engen Schleife ausführen. Das Beenden eines Tests in Eclipse kann dies manchmal auch verursachen, da heruntergefahrene Handler möglicherweise nicht aufgerufen werden.

Um sicherzustellen, dass das Problem tatsächlich darauf zurückzuführen ist, dass alte Worker Aufgaben erhalten, sollten Sie sich den Workflow-Verlauf ansehen, um zu bestimmen, welcher Prozess die Aufgabe erhalten hat, von der Sie erwartet hatten, dass der neue Worker sie erhält. Beispielsweise enthält das `DecisionTaskStarted`-Ereignis im Verlauf die Identität des Workflow-Workers, der die Aufgabe erhalten hat. Die vom Flow Framework verwendete ID hat die Form: {*`processId`*} @ {*`host name`*}. Im Folgenden finden Sie beispielsweise die Details des `DecisionTaskStarted` Ereignisses in der Amazon SWF SWF-Konsole für eine Beispielausführung:

Ereigniszeitstempel	Mon Feb 20 11:52:40 GMT-800 2012
Identität	2276 @ip -0A6C1 DF5
ID des geplanten Events	33

Um diese Situation zu vermeiden, verwenden Sie unterschiedliche Aufgabenlisten für jeden Test. Ziehen Sie außerdem in Betracht, eine Verzögerung zwischen dem Herunterfahren alter Worker und dem Starten neuer Worker hinzuzufügen.

Die Überprüfung ist aufgrund von Längenbeschränkungen für API-Parameter fehlgeschlagen

Amazon SWF erzwingt Längenbeschränkungen für API-Parameter. Sie erhalten eine `HTTP 400` Fehlermeldung, wenn Ihre Workflow- oder Aktivitätsimplementierung die Beschränkungen überschreitet. Wenn Sie beispielsweise aufrufen`recordActivityHeartbeat`, `ActivityExecutionContext` um einen Heartbeat für eine laufende Aktivität zu senden, darf die Zeichenfolge nicht länger als 2048 Zeichen sein.

Ein anderes häufiges Szenario ist, wenn eine Aktivität aufgrund einer Ausnahme fehlschlägt. Das Framework meldet Amazon SWF einen Aktivitätsfehler, indem es aufruft RespondActivityTaskFailedmit der serialisierten Ausnahme als Details. Der API-Aufruf meldet einen 400-Fehler, wenn die serialisierte Ausnahme eine Länge von mehr als 32.768 Byte hat. Um dieser Situation entgegenzuwirken, können Sie die Ausnahmemeldung oder die Ursachen kürzen, damit sie der Längenbeschränkung entsprechen.

AWS Flow Framework für Java-Referenz

Themen

- AWS Flow Framework für Java-Annotationen
- AWS Flow Framework für Java-Ausnahmen
- AWS Flow Framework für Java-Pakete

AWS Flow Framework für Java-Annotationen

Themen

- @Aktivität
- @Aktivität
- @ActivityRegistrationOptions
- @Asynchron
- @Execute
- @ExponentialRetry
- @GetState
- @ManualActivityCompletion
- @Signal
- @SkipRegistration
- @Wait und @ NoWait
- @Workflow
- @WorkflowRegistrationOptions

@Aktivität

Diese Annotation kann für eine Schnittstelle verwendet werden, um eine Gruppe von Aktivitätstypen zu deklarieren. Jede Methode in einer mit dieser Annotation versehenen Schnittstelle stellt einen Aktivitätstyp dar. Eine Schnittstelle kann nicht sowohl `@Workflow`- als auch `@Activities`-Annotationen enthalten.

Die folgenden Parameter können über dieser Annotation angegeben werden:

`activityNamePrefix`

Gibt das Präfix des Namens des in der Schnittstelle deklarierten Aktivitätstyps an. Wenn der Wert auf eine leere Zeichenfolge festgelegt ist (Standard), wird der Name der Schnittstelle gefolgt von einem Punkt (.) als Präfix verwendet.

`version`

Gibt die Standardversion der in der Schnittstelle deklarierten Aktivitätstypen an. Der Standardwert ist `1.0`.

`dataConverter`

Gibt den Typ des `DataConverter` an, der beim Erstellen von Aufgaben dieses Aktivitätstyps und ihrer Ergebnisse für die Serialisierung/Deserialisierung von Daten verwendet werden soll. Standardmäßig auf `NullDataConverter` festgelegt, was bedeutet, dass das `JsonDataConverter` verwendet werden soll.

@Aktivität

Diese Annotation kann für Methoden innerhalb einer mit `@Activities` definierten Schnittstelle verwendet werden.

Die folgenden Parameter können über dieser Annotation angegeben werden:

`name`

Gibt den Namen des Aktivitätstyps an. Der Standardwert ist eine leere Zeichenfolge. Diese gibt an, dass der Name des Aktivitätstyps über das Standardpräfix und den Namen der Aktivitätsmethode ({Präfix}{Name}) festgelegt werden soll. Beachten Sie, dass das Framework bei der Angabe eines Namens in einer `@Activity`-Annotation nicht automatisch ein Präfix voranstellt. Es steht Ihnen frei, Ihr eigenes Namensschema zu verwenden.

`version`

Gibt die Version des Aktivitätstyps an. Dies überschreibt die Standardversion, die in der `@Activities`-Annotation der enthaltenden Schnittstelle angegeben ist. Der Standardwert ist eine leere Zeichenfolge.

@ActivityRegistrationOptions

Gibt die Registrierungsoptionen für einen Aktivitätstyp an. Diese Annotation kann für eine Schnittstelle verwendet werden, die mit `@Activities` oder den darin enthaltenen Methoden definiert ist. Beim Festlegen an beiden Orten gilt die für die Methode verwendete Annotation.

Die folgenden Parameter können über dieser Annotation angegeben werden:

`defaultTasklist`

Gibt die Standard-Aufgabenliste an, die für diesen Aktivitätstyp bei Amazon SWF registriert werden soll. Dieser Standardwert kann beim Aufruf der Aktivitätsmethode für den generierten Client über den Parameter `ActivitySchedulingOptions` überschrieben werden. Standardmäßig auf `USE_WORKER_TASK_LIST` festgelegt. Dies ist ein spezieller Wert. Er gibt an, dass die Aufgabenliste des Workers verwendet werden soll, der die Registrierung durchführt.

`defaultTaskScheduleToStartTimeoutSeconds`

Gibt die bei Amazon SWF für diesen Aktivitätstyp defaultTaskSchedule ToStartTimeout registrierte Datei an. Dies ist die maximale Zeit, die eine Aufgabe dieses Aktivitätstyps warten darf, bevor sie einem Worker zugeordnet wird. Weitere Informationen finden Sie in der Amazon Simple Workflow Service API-Referenz.

`defaultTaskHeartbeatTimeoutSeconds`

Gibt die bei Amazon SWF für diesen Aktivitätstyp `defaultTaskHeartbeatTimeout` registrierte Datei an. Aktivitäts-Worker müssen innerhalb dieser Zeit einen Heartbeat liefern – andernfalls gibt es einen Timeout für die Aufgabe. Standardmäßig auf -1 festgelegt. Dieser spezielle Wert gibt an, dass der Timeout deaktiviert werden soll. Weitere Informationen finden Sie in der Amazon Simple Workflow Service API-Referenz.

`defaultTaskStartToCloseTimeoutSeconds`

Gibt die bei Amazon SWF für diesen Aktivitätstyp defaultTaskStart ToCloseTimeout registrierte Datei an. Dieser Timeout bestimmt die maximale Zeit, die ein Worker für die Bearbeitung einer Aktivität dieses Typs benötigen darf. Weitere Informationen finden Sie in der Amazon Simple Workflow Service API-Referenz.

`defaultTaskScheduleToCloseTimeoutSeconds`

Gibt die bei Amazon SWF für diesen Aktivitätstyp `defaultScheduleToCloseTimeout` registrierte Datei an. Dieser Timeout bestimmt die Gesamtdauer, für die die Aufgabe im offenen

Zustand bleiben kann. Standardmäßig auf -1 festgelegt. Dieser spezielle Wert gibt an, dass der Timeout deaktiviert werden soll. Weitere Informationen finden Sie in der Amazon Simple Workflow Service API-Referenz.

@Asynchron

Gibt bei Verwendung für eine Methode in der Workflow-Koordinationslogik an, dass die Methode asynchron ausgeführt werden soll. Ein Aufruf der Methode gibt die Kontrolle sofort zurück. Die eigentliche Ausführung erfolgt jedoch asynchron, sobald alle an die Methoden übergebenen Promise<>-Parameter bereit sind. Mit @Asynchronous definierte Methoden müssen den Rückgabetyp Promise<> oder void haben.

`daemon`

Gibt an, ob die für die asynchrone Methode erstellte Aufgabe eine Daemon-Aufgabe sein soll. Standardmäßig ist `False` festgelegt.

@Execute

Bei Verwendung für eine Methode in einer mit der `@Workflow`-Annotation definierten Schnittstelle gibt dieser Wert den Einstiegspunkt des Workflows an.

> **Important**
>
> Nur eine Methode in der Schnittstelle darf mit `@Execute` ausgezeichnet werden.

Die folgenden Parameter können über dieser Annotation angegeben werden:

`name`

Gibt den Namen des Workflowtyps an. Falls die Option nicht festgelegt ist, lautet der Name standardmäßig {prefix}{name}, wobei {prefix} der Name der Workflow-Schnittstelle ist, gefolgt von einem Punkt (.) und {name} der Name der `@Execute`-verziert-Methode im Workflow ist.

`version`

Gibt die Version des Workflowtyps an.

@ExponentialRetry

Legt bei Verwendung für eine Aktivität oder eine asynchrone Methode eine exponentielle Wiederholungsrichtlinie fest, falls die Methode eine unbehandelte Ausnahme auslöst. Ein Wiederholungsversuch erfolgt nach einer Backoff-Periode, die sich über die Anzahl der Versuche errechnet.

Die folgenden Parameter können über dieser Annotation angegeben werden:

`intialRetryIntervalSeconds`

Gibt die Dauer an, die vor dem ersten Wiederholungsversuch gewartet werden soll. Dieser Wert sollte nicht größer als `maximumRetryIntervalSeconds` und `retryExpirationSeconds` sein.

`maximumRetryIntervalSeconds`

Gibt die maximale Dauer zwischen den Wiederholungsversuchen an. Einmal erreicht, wird das Wiederholungsintervall auf diesen Wert begrenzt. Standardmäßig auf -1 festgelegt, was für eine unbegrenzte Dauer steht.

`retryExpirationSeconds`

Gibt die Dauer an, nach der die exponentielle Wiederholung gestoppt wird. Standardmäßig auf -1 festgelegt. Das heißt, es gibt keinen Ablauf.

`backoffCoefficient`

Gibt den Koeffizienten an, der zur Berechnung des Wiederholungsintervalls verwendet wird. Siehe Exponentielle Wiederholungsstrategie.

`maximumAttempts`

Gibt die Anzahl der Versuche an, nach denen die exponentielle Wiederholung gestoppt wird. Standardmäßig auf -1 festgelegt. Das heißt, es gibt keine Begrenzung der Anzahl der Wiederholungsversuche.

`exceptionsToRetry`

Gibt die Liste der Ausnahmetypen an, die einen erneuten Versuch auslösen sollen. Unbehandelte Ausnahmen dieser Typen werden nicht weitergegeben. Die Methode wird nach dem berechneten Wiederholungsintervall erneut ausgeführt. Standardmäßig enthält die Liste `Throwable`.

`excludeExceptions`

Gibt die Liste der Ausnahmetypen an, die keinen erneuten Versuch auslösen sollen. Unbehandelte Ausnahmen dieses Typs dürfen weitergegeben werden. Die Liste ist standardmäßig leer.

@GetState

Bei Verwendung für eine Methode in einer mit der `@Workflow`-Annotation definierten Schnittstelle wird die Methode verwendet, um den letzten Status der Workflow-Ausführung abzurufen. Es kann in einer Schnittstelle mit der `@Workflow`-Annotation maximal eine Methode mit dieser Annotation geben. Methoden mit dieser Annotation dürfen keine Parameter entgegennehmen und müssen einen anderen Rückgabetyp als `void` haben.

@ManualActivityCompletion

Diese Annotation kann für eine Aktivitätsmethode verwendet werden. Sie definiert, dass die Aktivitätsaufgabe bei der Rückgabe aus der Methode nicht abgeschlossen werden soll. Die Aktivitätsaufgabe wird nicht automatisch abgeschlossen und müsste manuell direkt über die Amazon SWF SWF-API abgeschlossen werden. Dies ist für Anwendungsfälle hilfreich, in denen die Aktivitätsaufgabe an ein externes System delegiert wird und dieses nicht automatisiert ist oder ein menschliches Eingreifen erfordert.

@Signal

Identifiziert bei Verwendung für eine Methode in einer mit der `@Workflow`-Annotation definierten Schnittstelle ein Signal, das von Ausführungen des von der Schnittstelle deklarierten Workflowtyps empfangen werden kann. Die Verwendung dieser Annotation ist erforderlich, um eine Signalmethode zu definieren.

Die folgenden Parameter können über dieser Annotation angegeben werden:

`name`

Gibt den Namensteil des Signalnamens an. Wenn nicht festgelegt, wird der Name der Methode verwendet.

@SkipRegistration

Wenn es auf einer Schnittstelle verwendet wird, die mit der `@Workflow` Anmerkung versehen ist, bedeutet dies, dass der Workflow-Typ nicht bei Amazon SWF registriert werden sollte. Für eine mit `@Workflow` definierte Schnittstelle muss die `@WorkflowRegistrationOptions`- oder `@SkipRegistrationOptions`-Annotationen verwendet werden. Es dürfen jedoch nicht beide verwendet werden.

@Wait und @ NoWait

Diese Anmerkungen können für einen Parameter des Typs verwendet werden, `Promise<>` um anzugeben, ob AWS Flow Framework for Java warten soll, bis er bereit ist, bevor die Methode ausgeführt wird. Standardmäßig müssen an die `@Asynchronous`-Methoden übergebene `Promise<>`-Parameter bereit sein, bevor die Methodenausführung erfolgt. In bestimmten Szenarien ist es notwendig, dieses Standardverhalten zu überschreiben. `Promise<>`-Parameter, die an `@Asynchronous`-Methoden übergeben und mit `@NoWait`-Annotationen versehen sind, werden nicht abgefragt.

Collection-Parameter (oder entsprechende Unterklassen) mit Zusagen wie `List<Promise<Int>>` müssen mit `@Wait` definiert werden. Standardmäßig wartet das Framework nicht auf die Mitglieder einer Collection.

@Workflow

Diese Annotation wird für eine Schnittstelle verwendet, um einen workflow-Typ zu deklarieren. Eine mit dieser Annotation ausgezeichnete Schnittstelle sollte genau eine Methode enthalten, die mit der @Execute-Annotation zur Deklaration des Einstiegspunkts für Ihren Workflow ausgezeichnet ist.

> **Note**
>
> Eine Schnittstelle kann nicht gleichzeitig die `@Workflow`- und `@Activities`-Annotationen verwenden. Sie schließen sich gegenseitig aus.

Die folgenden Parameter können über dieser Annotation angegeben werden:

`dataConverter`

Gibt an, welcher `DataConverter` beim Senden von Anforderungen an und beim Empfangen von Ergebnissen an/von Workflow-Ausführungen dieses Workflowtyps verwendet werden soll.

Die Standardeinstellung, auf `NullDataConverter` die wiederum zurückgegriffen wird, `JsonDataConverter` um alle Anfrage- und Antwortdaten als JavaScript Object Notation (JSON) zu verarbeiten.

Beispiel

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

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

@WorkflowRegistrationOptions

Stellt bei Verwendung auf einer mit Anmerkungen versehenen Oberfläche Standardeinstellungen bereit`@Workflow`, die von Amazon SWF bei der Registrierung des Workflow-Typs verwendet wurden.

> **Note**
>
> Für eine mit `@Workflow` definierte Schnittstelle muss `@WorkflowRegistrationOptions` oder `@SkipRegistrationOptions` verwendet werden. Es können nicht beide verwendet werden.

Die folgenden Parameter können über dieser Annotation angegeben werden:

Beschreibung

Eine optionale kurze Textbeschreibung des Workflowtyps.

`defaultExecutionStartToCloseTimeoutSeconds`

Gibt den bei Amazon SWF für den Workflowtyp `defaultExecutionStartToCloseTimeout` registrierten Typ an. Dies ist die Gesamtzeit, die eine solche Workflow-Ausführung in Anspruch nehmen kann.

Weitere Informationen zu Workflow-Timeouts finden Sie unter Amazon SWF-Timeout-Typen .

`defaultTaskStartToCloseTimeoutSeconds`

Gibt den bei Amazon SWF für den Workflowtyp `defaultTaskStartToCloseTimeout` registrierten Typ an. Gibt an, wie lange eine einzelne Entscheidungsaufgabe für eine solche Workflow-Ausführung dauern kann.

Wenn Sie `defaultTaskStartToCloseTimeout` nicht angeben, wird die Standardeinstellung auf 30 Sekunden gesetzt.

Weitere Informationen zu Workflow-Timeouts finden Sie unter Amazon SWF-Timeout-Typen .

`defaultTaskList`

Die Standard-Aufgaben, die für Entscheidungsaufgaben für Ausführungen dieses Workflowtyps verwendet wird. Die hier festgelegte Voreinstellung kann beim Starten einer Workflow-Ausführung mit `StartWorkflowOptions` überschrieben werden.

Wenn Sie `defaultTaskList` nicht angeben, wird der Wert standardmäßig auf `USE_WORKER_TASK_LIST` festgelegt. Dies bedeutet, dass die Aufgabenliste des Workers verwendet werden soll, der die Workflow-Registrierung durchführt.

`defaultChildPolicy`

Gibt die Richtlinie an, die für untergeordnete Workflows verwendet werden soll, wenn eine Ausführung dieses Typs abgebrochen wird. Der Standardwert ist `ABANDON`. Die möglichen Werte sind:

- `ABANDON`— Erlaubt, dass die untergeordneten Workflow-Ausführungen weiterlaufen
- `TERMINATE`— Beendet untergeordnete Workflow-Ausführungen
- `REQUEST_CANCEL`— Beantragen Sie die Stornierung der untergeordneten Workflow-Ausführungen

AWS Flow Framework für Java-Ausnahmen

Die folgenden Ausnahmen werden von der AWS Flow Framework für Java verwendet. Dieser Abschnitt bietet eine Übersicht über die Ausnahmen. Weitere Einzelheiten finden Sie in der AWS SDK für Java Dokumentation der einzelnen Ausnahmen.

Themen

- ActivityFailureException
- ActivityTaskException
- ActivityTaskFailedException
- ActivityTaskTimedOutException
- ChildWorkflowException
- ChildWorkflowFailedException
- ChildWorkflowTerminatedException
- ChildWorkflowTimedOutException
- DataConverterException
- DecisionException
- ScheduleActivityTaskFailedException
- SignalExternalWorkflowException
- StartChildWorkflowFailedException
- StartTimerFailedException
- TimerException
- WorkflowException

ActivityFailureException

Diese Ausnahme wird vom Framework intern verwendet, um fehlgeschlagene Aktivitäten zu kommunizieren. Wenn eine Aktivität aufgrund einer unbehandelten Ausnahme fehlschlägt, wird sie zusammengefasst `ActivityFailureException` und an Amazon SWF gemeldet. Sie müssen diese Ausnahme nur bearbeiten, wenn Sie die Erweiterbarkeitspunkte des Aktivitäts-Workers verwenden. Ihr Anwendungscode wird nie für die Bearbeitung dieser Ausnahme verwendet.

ActivityTaskException

Dies ist die Basisklasse für Ausnahmen von Fehlern bei Aktivitätsaufgaben: `ScheduleActivityTaskFailedException`, `ActivityTaskFailedException`, `ActivityTaskTimedoutException`. Sie enthält die Aufgaben-ID und den Aktivitätstyp der fehlgeschlagenen Aufgabe. Sie können diese Ausnahme in Ihrer Workflow-Implementierung abfangen, um fehlgeschlagene Aktivitäten generisch zu bearbeiten.

ActivityTaskFailedException

Unbearbeitete Ausnahmen in Aktivitäten werden der Workflow-Implementierung zurückgemeldet, indem `ActivityTaskFailedException` ausgelöst wird. Die ursprüngliche Ausnahme kann aus der cause-Eigenschaft dieser Ausnahme abgerufen werden. Die Ausnahme liefert aber auch weitere Informationen, die sich beim Debugging als hilfreich erweisen können, z. B. den eindeutigen Bezeichner der Aktivität im Verlauf.

Das Framework kann die Remote-Ausnahme bereitstellen, indem die ursprüngliche Ausnahme vom Aktivitäts-Worker serialisiert wird.

ActivityTaskTimedOutException

Diese Ausnahme wird ausgelöst, wenn bei einer Aktivität von Amazon SWF ein Timeout ausgelöst wurde. Dazu kommt es, wenn die Aktivitätsaufgabe dem Worker nicht innerhalb des erforderlichen Zeitraums zugewiesen oder vom Worker nicht in der erforderlichen Zeit abgeschlossen werden konnte. Sie können diese Timeouts in der Aktivität mit der `@ActivityRegistrationOptions`-Annotation festlegen oder beim Aufrufen der Aktivitätsmethode mit dem `ActivitySchedulingOptions`-Parameter.

ChildWorkflowException

Basisklasse für Ausnahmen, mit der fehlgeschlagene Ausführungen von untergeordneten Workflows zurückgemeldet werden. Die Ausnahme enthält die IDs der untergeordneten Workflow-Ausführung sowie den Workflow-Typ. Sie können diese Ausnahme in Ihrer Workflow-Implementierung abfangen, um fehlgeschlagene Ausführungen untergeordneter Workflows generisch zu bearbeiten.

ChildWorkflowFailedException

Unbearbeitete Ausnahmen in untergeordneten Workflows werden der übergeordneten Workflow-Implementierung zurückgemeldet, indem `ChildWorkflowFailedException` ausgelöst wird. Die

ursprüngliche Ausnahme kann aus der `cause`-Eigenschaft dieser Ausnahme abgerufen werden. Die Ausnahme liefert aber auch weitere Informationen, die sich beim Debugging als hilfreich erweisen können, z. B. den eindeutigen Bezeichner der untergeordneten Ausführung.

ChildWorkflowTerminatedException

Diese Ausnahme wird in übergeordneten Workflow-Ausführungen ausgelöst, um eine beendete untergeordnete Workflow-Ausführung zu melden. Sie sollten diese Ausnahme abfangen, wenn Sie den beendeten untergeordneten Workflow bearbeiten möchten, z. B. um eine Bereinigung oder Erstattung durchzuführen.

ChildWorkflowTimedOutException

Diese Ausnahme wird bei der Ausführung eines übergeordneten Workflows ausgelöst, um zu melden, dass bei der Ausführung eines untergeordneten Workflows das Timeout überschritten und von Amazon SWF geschlossen wurde. Sie sollten diese Ausnahme abfangen, wenn Sie den untergeordneten Workflow, der beendet werden musste, bearbeiten möchten, z. B. um eine Bereinigung oder Erstattung durchzuführen.

DataConverterException

Das Framework verwendet die `DataConverter`-Komponente für das Marshalling und Unmarshalling von Daten, die remote übertragen wurden. Diese Ausnahme wird ausgelöst, wenn das Marshalling oder Unmarshalling von Daten durch `DataConverter` fehlschlägt. Dafür gibt es viele mögliche Gründe, beispielsweise wenn die `DataConverter`-Komponenten, die für das Marshalling und Unmarshalling von Daten verwendet werden, nicht übereinstimmen.

DecisionException

Dies ist die Basisklasse für Ausnahmen, die darauf hindeuten, dass eine Entscheidung von Amazon SWF nicht umgesetzt wurde. Sie können diese Ausnahme in Ihrer Workflow-Implementierung abfangen, um solche Ausnahmen generisch zu bearbeiten.

ScheduleActivityTaskFailedException

Diese Ausnahme wird ausgelöst, wenn Amazon SWF eine Aktivitätsaufgabe nicht planen kann. Dies kann verschiedene Gründe haben — zum Beispiel, weil die Aktivität veraltet war oder ein Amazon SWF SWF-Limit für Ihr Konto erreicht wurde. Die `failureCause`-Eigenschaft in der Ausnahme gibt den genauen Grund für die fehlgeschlagene Planung einer Aktivität an.

SignalExternalWorkflowException

Diese Ausnahme wird ausgelöst, wenn Amazon SWF eine Anforderung der Workflow-Ausführung nicht verarbeitet, um eine weitere Workflow-Ausführung zu signalisieren. Dies passiert, wenn die Ziel-Workflow-Ausführung nicht gefunden werden konnte — das heißt, die von Ihnen angegebene Workflow-Ausführung existiert nicht oder befindet sich im geschlossenen Zustand.

StartChildWorkflowFailedException

Diese Ausnahme wird ausgelöst, wenn Amazon SWF die Ausführung eines untergeordneten Workflows nicht starten kann. Dies kann verschiedene Gründe haben, z. B. weil der angegebene Workflow-Typ für untergeordnete Benutzer veraltet ist oder ein Amazon SWF SWF-Limit für Ihr Konto erreicht wurde. Die `failureCause`-Eigenschaft in der Ausnahme gibt den genauen Grund für den fehlgeschlagene Start einer untergeordneten Workflow-Ausführung an.

StartTimerFailedException

Diese Ausnahme wird ausgelöst, wenn Amazon SWF einen von der Workflow-Ausführung angeforderten Timer nicht starten kann. Dies kann passieren, wenn die angegebene Timer-ID bereits verwendet wird oder ein Amazon SWF SWF-Limit für Ihr Konto erreicht wurde. Die `failureCause`-Eigenschaft in der Ausnahme gibt den genauen Grund für den Fehler an.

TimerException

Dies ist die Basisklasse für Ausnahmen, die mit Timern im Zusammenhang stehen.

WorkflowException

Diese Ausnahme wird vom Framework intern verwendet, um Fehler in Workflow-Ausführungen zu kommunizieren. Sie müssen diese Ausnahme nur verarbeiten, wenn Sie einen Erweiterbarkeitspunkt des Workflow-Workers verwenden.

AWS Flow Framework für Java-Pakete

Dieser Abschnitt bietet einen Überblick über die Pakete, die in der AWS Flow Framework für Java enthalten sind. Weitere Informationen zu den einzelnen Paketen finden Sie unter com.amazonaws.services.simpleworkflow.flow in der API-Referenz.AWS SDK für Java

com.amazonaws.services.simpleworkflow.flow

Enthält Komponenten, die in Amazon SWF integriert sind.

com.amazonaws.services.simpleworkflow.flow.annotations

Enthält die Anmerkungen, die vom Programmiermodell für Java verwendet werden. AWS Flow Framework

com.amazonaws.services.simpleworkflow.flow.aspectj

Enthält für Java Komponenten, die für Funktionen wie und erforderlich sind. AWS Flow Framework @Asynchron @ExponentialRetry

com.amazonaws.services.simpleworkflow.flow.common

Enthält gängige Dienstprogramme wie Framework-definierte Konstanten.

com.amazonaws.services.simpleworkflow.flow.core

Enthält Kernfunktionen wie `Task` und `Promise`.

com.amazonaws.services.simpleworkflow.flow.generic

Enthält Kernkomponenten wie generische Clients, auf die andere Funktionen aufbauen.

com.amazonaws.services.simpleworkflow.flow.interceptors

Enthält Implementierungen der vom Framework bereitgestellten Decorators, einschließlich `RetryDecorator`.

com.amazonaws.services.simpleworkflow.flow.junit

Enthält Komponenten, die Junit-Integration zur Verfügung stellen.

com.amazonaws.services.simpleworkflow.flow.pojo

Enthält Klassen, die Aktivitäts- und Workflow-Definitionen für das Annotationsbasierte Programmierungsmodell implementieren.

com.amazonaws.services.simpleworkflow.flow.spring

Enthält Komponenten, die Spring-Integration zur Verfügung stellen.

com.amazonaws.services.simpleworkflow.flow.test

Enthält Helferobjektklassen, wie `TestWorkflowClock`, für Workflow-Implementierungen zum Testen der Einheit.

com.amazonaws.services.simpleworkflow.flow.worker

Enthält Implementierungen von Aktivitäts- und Workflow-Workern.

Dokumentverlauf

In der folgenden Tabelle werden die wichtigen Änderungen an der Dokumentation seit der letzten Version des AWS Flow Framework for Java Developer Guide beschrieben.

- API-Version: 2012-01-25
- Letzte Aktualisierung der Dokumentation: 25. Juni 2018

Änderung	Beschreibung	Änderungsdatum
Aktualisierung	Es wurde ein Fehler in der `backoffCoefficient` - Beschreibung für `@ExponentialRetry` behoben. Siehe @ExponentialRetry.	25. Juni 2018
Aktualisierung	Im gesamten Handbuch wurden die Codebeispiele bereinigt.	5. Juni 2017
Aktualisierung	Die Anordnung und die Inhalte des Handbuchs wurden vereinfacht und verbessert.	19. Mai 2017
Aktualisierung	Der Abschnitt Vornehmen von Änderungen am Entscheidercode: Versioning und Funktions-Flags wurde vereinfacht und verbessert.	10. April 2017
Aktualisierung	Der neue Bewährte Methoden-Abschnitt mit neuer Anleitung zum Ändern des Decider-Codes wurde hinzugefügt.	3. März 2017
Neues Feature	Sie können Lambda-Aufgaben zusätzlich zu herkömmlichen Aktivitätsaufgaben in Ihren Workflows angeben. Weitere Informationen finden Sie unter AWS Lambda Aufgaben umsetzen.	21. Juli 2015
Neues Feature	Amazon SWF unterstützt das Festlegen der Aufgabenpriorität in einer Aufgabenliste und versucht, die Aufgaben mit höherer Priorität vor Aufgaben mit niedrigerer Priorität	17. Dezember 2014

Änderung	Beschreibung	Änderungsdatum
	zu liefern. Weitere Informationen finden Sie unter Aufgabenpriorität in Amazon SWF festlegen.	
Aktualisierung	Aktualisierungen und Korrekturen wurden vorgenommen.	1. August 2013
Aktualisierung	• Aktualisierungen und Korrekturen, einschließlich Aktualisierungen der Einrichtungsanweisungen für Eclipse 4.3 und AWS SDK für Java 1.4.7 wurden vorgenommen. • Neue Tutorials zum Erstellen von Starter-Szenarien wurden hinzugefügt.	28. Juni 2013
Neues Feature	Die erste Version von AWS Flow Framework für Java.	27. Februar 2012

Die vorliegende Übersetzung wurde maschinell erstellt. Im Falle eines Konflikts oder eines Widerspruchs zwischen dieser übersetzten Fassung und der englischen Fassung (einschließlich infolge von Verzögerungen bei der Übersetzung) ist die englische Fassung maßgeblich.