import org.apache.camel.builder.RouteBuilder;

import javax.sql.DataSource;

import java.util.List;

import java.util.Map;

import java.util.Optional;

public class CheckAndProcessRoute extends RouteBuilder {

private final DataSource dataSource;

public CheckAndProcessRoute(DataSource dataSource) {

this.dataSource = dataSource;

}

@Override

public void configure() throws Exception {

from("direct:checkAndProcess")

.routeId("checkAndProcessRoute")

.process(exchange -> {

// Получаем route\_id и data\_hash из заголовков, используя Optional для обработки null

String routeId = Optional.ofNullable(exchange.getIn().getHeader("routeId", String.class)).orElse("");

String dataHash = Optional.ofNullable(exchange.getIn().getHeader("dataHash", String.class)).orElse("");

// Проверяем, что заголовки не пустые

if (routeId.isEmpty() || dataHash.isEmpty()) {

exchange.getIn().setBody("No routeId or dataHash found. Proceeding with route execution.");

exchange.getIn().setHeader("skipCheck", true); // Устанавливаем флаг, чтобы пропустить проверку в базе данных

return;

}

// Формируем SQL-запрос для проверки наличия записи в таблице

String query = String.format(

"SELECT 1 FROM data\_hashes WHERE route\_id = '%s' AND data\_hash = '%s'",

routeId, dataHash

);

// Устанавливаем SQL-запрос в тело сообщения

exchange.getIn().setBody(query);

})

.choice()

.when(header("skipCheck").isEqualTo(true)) // Проверяем, нужно ли пропустить проверку в базе данных

.log("Skipping database check due to missing headers.")

.to("direct:executeCurrentRoute") // Переход к выполнению текущего маршрута

.otherwise()

.to("jdbc:dataSource") // Выполняем запрос к базе данных

.choice()

.when(simple("${body.size()} > 0")) // Проверяем, если запись существует

.log("Route ID and data hash already exist. Skipping current route.")

.stop() // Пропускаем выполнение текущего маршрута

.otherwise()

.log("No matching route ID and data hash found. Executing current route.")

.to("direct:executeCurrentRoute") // Переход к выполнению текущего маршрута

.endChoice()

.end()

.to("direct:nextStep"); // Переход к следующему шагу

}

}

### **Подробное объяснение**

1. **Обработка заголовков с использованием Optional**:
   * Мы используем Optional.ofNullable(...) для безопасного получения значений из заголовков сообщения. Если значение null, Optional вернет пустую строку ("").
   * Если любой из заголовков пустой (то есть данные отсутствуют), устанавливается сообщение, которое говорит, что совпадений не найдено, и добавляется заголовок skipCheck, который используется для пропуска проверки в базе данных.

java  
Копировать код  
String routeId = Optional.ofNullable(exchange.getIn().getHeader("routeId", String.class)).orElse("");

String dataHash = Optional.ofNullable(exchange.getIn().getHeader("dataHash", String.class)).orElse("");

1. **Проверка на пустые значения и установка флага skipCheck**:
   * Если хотя бы одно значение заголовка пустое, устанавливается флаг skipCheck в true, чтобы избежать выполнения SQL-запроса к базе данных.
   * Вместо выброса исключения, маршрут логирует, что данные отсутствуют, и продолжает выполнение текущего маршрута.

java  
Копировать код  
if (routeId.isEmpty() || dataHash.isEmpty()) {

exchange.getIn().setBody("No routeId or dataHash found. Proceeding with route execution.");

exchange.getIn().setHeader("skipCheck", true);

return;

}

1. **Условное ветвление для проверки флага skipCheck**:
   * Сначала проверяется, установлен ли флаг skipCheck. Если да, маршрут пропускает выполнение SQL-запроса и переходит к выполнению текущего маршрута.
   * Если флаг skipCheck не установлен, выполняется SQL-запрос к базе данных.

java  
Копировать код  
.choice()

.when(header("skipCheck").isEqualTo(true))

.log("Skipping database check due to missing headers.")

.to("direct:executeCurrentRoute") // Выполнение текущего маршрута

1. **Выполнение SQL-запроса и последующая проверка**:
   * Если заголовки содержат данные, маршрут выполняет SQL-запрос.
   * Если запрос возвращает результат (найдена пара route\_id и data\_hash), маршрут останавливается.
   * Если пара не найдена, маршрут продолжает выполнение текущего действия.

java  
Копировать код  
.to("jdbc:dataSource")

.choice()

.when(simple("${body.size()} > 0"))

.log("Route ID and data hash already exist. Skipping current route.")

.stop()

.otherwise()

.log("No matching route ID and data hash found. Executing current route.")

.to("direct:executeCurrentRoute")

### **Заключение**

Такой подход позволяет маршруту гибко обрабатывать случаи, когда данные отсутствуют, и продолжать выполнение с логированием и правильной логикой, без выброса исключений. Это обеспечивает надежность работы маршрута и правильное управление выполнением на основе данных в заголовках и таблице.

Для реализации сценария, где один инстанс останавливается на середине выполнения маршрута, а резервный инстанс продолжает выполнение с того же шага, необходимо обеспечить синхронизацию состояния между этими инстансами. Ваша таблица хешей (data-hashes) будет использоваться для отслеживания выполнения шагов маршрута и определения текущего состояния.

### **Основные шаги реализации**

1. **Хранение состояния в таблице хешей:** Каждый шаг маршрута записывает свой идентификатор и хеш данных в таблицу хешей (data-hashes).
2. **Проверка состояния:** Резервный инстанс должен проверять состояние выполненных шагов при старте и продолжать выполнение с того шага, на котором остановился основной инстанс.
3. **Остановка основного инстанса:** Основной инстанс может быть остановлен вручную, через мониторинг, либо автоматически при возникновении ошибки.
4. **Продолжение резервного инстанса:** Резервный инстанс стартует, проверяет, какие шаги уже выполнены, и продолжает выполнение с последнего незавершенного шага.

### **Пример реализации маршрута с проверкой состояния и продолжением выполнения**

#### **Шаг 1: Маршрут основного инстанса, записывающий состояние в таблицу хешей**

Этот маршрут будет выполнять шаги и записывать состояние в таблицу хешей после выполнения каждого шага:

java

Копировать код

import org.apache.camel.builder.RouteBuilder;

import javax.sql.DataSource;

import java.util.UUID;

public class MainInstanceRoute extends RouteBuilder {

private final DataSource dataSource;

public MainInstanceRoute(DataSource dataSource) {

this.dataSource = dataSource;

}

@Override

public void configure() throws Exception {

from("direct:start")

.routeId("mainInstanceRoute")

.process(exchange -> {

// Уникальный идентификатор для каждого шага

String stepId = UUID.randomUUID().toString();

String data = "Some data for step 1"; // Пример данных

String hash = DataHasher.hashData(data);

// Сохраняем шаг и хеш в таблице

String insertQuery = String.format(

"INSERT INTO data\_hashes (route\_id, step\_id, data\_hash) VALUES ('%s', '%s', '%s')",

"mainInstanceRoute", stepId, hash

);

// Выполняем запрос вставки в базу данных

exchange.getIn().setBody(insertQuery);

})

.to("jdbc:dataSource")

.log("Step 1 executed and hash stored.")

// Здесь может быть остановка маршрута в любой момент

.to("direct:nextStep"); // Переход к следующему шагу

}

}

#### **Шаг 2: Резервный маршрут, проверяющий состояние и продолжающий выполнение**

Этот маршрут запускается, проверяет, какие шаги уже выполнены, и продолжает выполнение с последнего незавершенного шага:

java

Копировать код

import org.apache.camel.builder.RouteBuilder;

import javax.sql.DataSource;

import java.util.List;

import java.util.Map;

public class BackupInstanceRoute extends RouteBuilder {

private final DataSource dataSource;

public BackupInstanceRoute(DataSource dataSource) {

this.dataSource = dataSource;

}

@Override

public void configure() throws Exception {

from("direct:startBackup")

.routeId("backupInstanceRoute")

.process(exchange -> {

// Проверяем состояние выполненных шагов

String query = "SELECT step\_id FROM data\_hashes WHERE route\_id = 'mainInstanceRoute' ORDER BY timestamp DESC";

exchange.getIn().setBody(query);

})

.to("jdbc:dataSource")

.process(exchange -> {

List<Map<String, Object>> results = exchange.getIn().getBody(List.class);

// Находим последний выполненный шаг

String lastStepId = results.isEmpty() ? null : (String) results.get(0).get("step\_id");

// Логика определения с какого шага продолжить

if (lastStepId == null) {

exchange.getIn().setHeader("currentStep", 1);

} else {

// Определяем следующий шаг на основе lastStepId

exchange.getIn().setHeader("currentStep", determineNextStep(lastStepId));

}

})

.choice()

.when(header("currentStep").isEqualTo(1))

.log("Starting from Step 1")

.to("direct:executeStep1")

.when(header("currentStep").isEqualTo(2))

.log("Continuing from Step 2")

.to("direct:executeStep2")

// Добавляйте дополнительные шаги по мере необходимости

.otherwise()

.log("Unknown step. Aborting.")

.stop()

.end();

}

// Метод определения следующего шага

private int determineNextStep(String lastStepId) {

// Логика определения следующего шага на основе lastStepId

// Например, возвращаем следующий номер шага

// Реализуйте соответствующую логику на основе ваших шагов

return 2; // Пример: если последний выполненный шаг - Step 1, продолжаем с Step 2

}

}

### **Объяснение логики:**

1. **Запись состояния основным инстансом:**
   * Основной маршрут выполняет шаги и записывает данные о выполненных шагах (идентификатор шага и хеш данных) в таблицу data-hashes.
   * Каждый шаг получает уникальный идентификатор stepId, и его состояние сохраняется в базе данных.
2. **Проверка и продолжение резервным инстансом:**
   * Резервный маршрут сначала проверяет таблицу data-hashes, чтобы определить, какие шаги уже выполнены основным инстансом.
   * На основании последнего выполненного шага определяется следующий шаг для выполнения.
   * Резервный маршрут продолжает выполнение с того шага, на котором остановился основной маршрут, используя заголовок currentStep для выбора следующего шага.
3. **Определение следующего шага:**
   * Метод determineNextStep определяет, какой шаг должен быть выполнен следующим, на основе идентификатора последнего выполненного шага (lastStepId).
   * Вы можете настроить этот метод, чтобы он соответствовал вашей бизнес-логике и порядку шагов.

### **Заключение**

Эта реализация позволяет резервному инстансу продолжить выполнение с того же шага, на котором остановился основной инстанс, используя данные из таблицы хешей для отслеживания состояния. Этот подход обеспечивает надежность и согласованность выполнения процессов в распределенной системе, где один инстанс может подхватить выполнение маршрута с того же места, где его прервал другой инстанс.

Для реализации сценария, в котором пользователь определяет, какие шаги и в какой последовательности должны быть выполнены, можно использовать подход с динамическим построением маршрутов на основе пользовательского ввода. Этот ввод может быть получен, например, через REST API, конфигурационный файл, базу данных или любое другое средство ввода данных.

### **Основные шаги реализации**

1. **Получение пользовательской информации о шагах:**
   * Пользователь может предоставить список шагов и их последовательность через REST API или другой интерфейс.
   * Эти данные можно хранить в виде JSON, XML, или другого удобного формата.
2. **Построение маршрута на основе пользовательского ввода:**
   * На основе полученных данных маршрут динамически строится и выполняется в нужной последовательности.
   * Apache Camel поддерживает динамическое построение маршрутов через использование компонентов Direct, Dynamic Router и Recipient List.

### **Пример реализации**

#### **1. Определение REST API для получения пользовательского ввода**

Используем REST API для получения списка шагов и их последовательности от пользователя.

java

Копировать код

import org.apache.camel.builder.RouteBuilder;

import org.apache.camel.model.rest.RestBindingMode;

public class RestApiRoute extends RouteBuilder {

@Override

public void configure() throws Exception {

restConfiguration()

.component("restlet")

.bindingMode(RestBindingMode.json);

rest("/api")

.post("/execute-steps")

.consumes("application/json")

.produces("application/json")

.type(StepRequest.class) // Объект, содержащий информацию о шагах

.route()

.routeId("dynamicStepExecution")

.process(exchange -> {

// Получаем информацию о шагах от пользователя

StepRequest stepRequest = exchange.getIn().getBody(StepRequest.class);

// Устанавливаем шаги в заголовок для использования в маршруте

exchange.getIn().setHeader("steps", stepRequest.getSteps());

})

.to("direct:executeDynamicSteps"); // Переход к маршруту выполнения шагов

}

}

#### **2. Класс для представления пользовательского ввода**

Создаем класс StepRequest для хранения информации о шагах, которую предоставляет пользователь.

java

Копировать код

import java.util.List;

public class StepRequest {

private List<String> steps; // Список шагов, например, ["step1", "step2", "step3"]

// Геттеры и сеттеры

public List<String> getSteps() {

return steps;

}

public void setSteps(List<String> steps) {

this.steps = steps;

}

}

#### **3. Маршрут для динамического выполнения шагов**

Этот маршрут выполняет шаги в последовательности, заданной пользователем.

java

Копировать код

import org.apache.camel.builder.RouteBuilder;

import java.util.List;

public class ExecuteStepsRoute extends RouteBuilder {

@Override

public void configure() throws Exception {

from("direct:executeDynamicSteps")

.routeId("executeStepsRoute")

.process(exchange -> {

// Получаем список шагов из заголовка

List<String> steps = exchange.getIn().getHeader("steps", List.class);

// Строим маршрут динамически на основе шагов пользователя

for (String step : steps) {

// Логика обработки каждого шага

// Переход к нужному шагу в зависимости от его имени

switch (step) {

case "step1":

exchange.getContext().createProducerTemplate().sendBody("direct:executeStep1", exchange.getIn().getBody());

break;

case "step2":

exchange.getContext().createProducerTemplate().sendBody("direct:executeStep2", exchange.getIn().getBody());

break;

case "step3":

exchange.getContext().createProducerTemplate().sendBody("direct:executeStep3", exchange.getIn().getBody());

break;

default:

throw new IllegalArgumentException("Unknown step: " + step);

}

}

})

.log("All steps executed according to user instructions.");

}

}

#### **4. Примеры реализации шагов**

Определите шаги как отдельные маршруты или методы, к которым будет обращаться динамический маршрут:

java

Копировать код

public class StepsDefinition extends RouteBuilder {

@Override

public void configure() throws Exception {

from("direct:executeStep1")

.log("Executing Step 1")

.process(exchange -> {

// Логика для шага 1

});

from("direct:executeStep2")

.log("Executing Step 2")

.process(exchange -> {

// Логика для шага 2

});

from("direct:executeStep3")

.log("Executing Step 3")

.process(exchange -> {

// Логика для шага 3

});

}

}

### **Объяснение**

1. **Получение данных от пользователя**:
   * REST API принимает POST-запрос с данными о шагах и их последовательности.
   * Данные сохраняются в виде объекта StepRequest, который передается в основной маршрут.
2. **Построение и выполнение маршрута**:
   * Основной маршрут использует информацию о шагах из заголовков сообщения для динамического выполнения маршрутов.
   * В зависимости от имени шага (step1, step2, step3), маршрут выполняет соответствующий шаг, используя ProducerTemplate для синхронного вызова маршрутов direct:executeStep1, direct:executeStep2, и так далее.
3. **Обработка ошибок**:
   * Если пользователь указывает неизвестный шаг, маршрут выбрасывает исключение, уведомляя об ошибке.

### **Пример запроса от пользователя**

Пример JSON-запроса от пользователя, который определяет последовательность шагов:

json

Копировать код

{

"steps": ["step1", "step3", "step2"]

}

### **Заключение**

Этот подход позволяет динамически управлять выполнением шагов маршрута в Apache Camel на основе пользовательского ввода. Пользователь предоставляет последовательность шагов через REST API, а маршрут Camel выполняет их в указанной последовательности. Такой гибкий подход позволяет адаптировать маршруты к потребностям бизнеса и требованиям пользователей без необходимости изменения кода маршрутов.

4o