Spring Cloud Schema Registryの調査報告

# 参照リンク

* 「Spring Cloud Schema Registry」公式サイト：

<https://docs.spring.io/spring-cloud-schema-registry/docs/1.1.3-SNAPSHOT/reference/html/spring-cloud-schema-registry.html>

* 「Spring Cloud Schema Registry」公式サンプルコード：

<https://github.com/spring-cloud/spring-cloud-stream-samples/tree/main/schema-registry-samples/schema-registry-vanilla-kafka-binder>

* 「Apache Avro」公式サイト：

<http://avro.apache.org/docs/current/gettingstartedjava.html>

# Spring Cloud Schema Registryの紹介

## Spring Cloud Schema Registryのアプリケーションシナリオ

組織にメッセージングベースのpub / subアーキテクチャがあり、複数のプロデューサーとコンシューマーのマイクロサービスが相互に通信する場合、多くの場合、これらすべてのマイクロサービスがスキーマに基づくコントラクトに合意する必要があります。このようなスキーマを進化させて新しいビジネス要件に対応する必要がある場合でも、既存のコンポーネントは引き続き機能する必要があります。このプロジェクトは、前述のスキーマをアプリケーションで登録および使用できるスタンドアロンスキーマレジストリサーバーのサポートを提供します。

## Spring Cloud Schema Registryの概要

Spring Cloud Schema Registry は**スキーマの進化をサポートしているため**、データを時間の経過とともに進化させながら、古いまたは新しいプロデューサーとコンシューマーで、またその逆も可能です。ほとんどの直列化モデル、特に異なるプラットフォームと言語間での移植性を目的とするモデルは、データがバイナリペイロードで直列化される方法を記述するスキーマに依存しています。データを直列化して解釈するには、送信側と受信側の両方がバイナリ形式を記述するスキーマにアクセスできる必要があります。場合によっては、直列化のペイロードタイプから、または逆直列化のターゲットタイプからスキーマを推測できます。

ただし、多くのアプリケーションは、バイナリデータ形式を記述する明示的なスキーマにアクセスできるという利点があります。スキーマレジストリを使用すると、スキーマ情報をテキスト形式（通常は JSON）で保存し、バイナリ形式でデータを送受信する必要のあるさまざまなアプリケーションがその情報にアクセスできるようにします。

## 「Schema Registry Client」について

* Spring Cloud Streamは、独自のスキーマサーバーと対話し、Confluentスキーマレジストリと対話するための「out-of-the-box」な実装を提供します。
* 「Spring Cloud Stream schema registry」のクライアントは、次のように@EnableSchemaRegistryClientを使用して構成できます。

@EnableBinding(Sink.class)

@SpringBootApplication

@EnableSchemaRegistryClient

public static class AvroSinkApplication {

//...

}

* Schema Registry Clientプロパティ：

・「spring.cloud.schemaRegistryClient.endpoint」：スキーマサーバーの場所。これを設定するときは、プロトコル（httpまたはhttps）、ポート、コンテキストパスを含む完全なURLを使用してください。（デフォルト：<http://localhost:8990/>）

・「spring.cloud.schemaRegistryClient.cached」：クライアントがスキーマサーバーのレスポンスをキャッシュする必要があるかどうか。（デフォルト：false）

* 「Spring Cloud Stream」は、「spring-cloud-schema-registry-client」モジュールを介してスキーマベースのメッセージコンバーターのサポートを提供します。 現在、スキーマベースのメッセージコンバーターですぐにサポートされるシリアル化形式は「Apache Avro」のみであり、将来のバージョンでさらに多くの形式が追加される予定です。
* Avro Schema Registry Message Converterプロパティについて：

・「spring.cloud.stream.bindings.output.contentType=application/\*+avro」：「Avro based schema registry client」を有効にします。

・「spring.cloud.schema.avro.dynamicSchemaGenerationEnabled」：コンバーターがリフレクションを使用してPOJOからスキーマを推測する場合に有効にします。（デフォルト：false）

・「spring.cloud.schema.avro.readerSchema」：Avroは、ライタースキーマ（元のペイロード）とリーダースキーマ（アプリケーションペイロード）を調べて、スキーマのバージョンを比較します。（デフォルト：null）

・「spring.cloud.schema.avro.schemaLocations」：このプロパティにリストされている「.avsc」ファイルをスキーマサーバーに登録します。（デフォルト：empty）

・「spring.cloud.schema.avro.prefix」：Content-Typeヘッダーで使用されるプレフィックス。（デフォルト：vnd）

・「spring.cloud.schema.avro.subjectNamingStrategy」：スキーマレジストリにAvroスキーマを登録するために使用されるサブジェクト名を決定します。（デフォルト：org.springframework.cloud.stream.schema.avro.DefaultSubjectNamingStrategy）

# ローカルでSample起動方法

このサンプルでは、異なるペイロードスキーマでイベントを生成する2つのプロデューサーがあると単純に想定します。 両方のペイロードバージョンを消費するコンシューマーは、進化するスキーマに適応するように設計されます。

## アプリケーションを実行

1. 添付ファイルの「source」ディレクトリで次のコマンドを実行して、Dockerコンテナでkafkaを起動します。

> docker-compose up -d

※kafkaのクリーンアップ方法：「CTRL+C」終了し、次のコマンドを実行します。

> docker-compose down

1. 添付ファイルの「jars」ディレクトリで次のコマンドを実行して、「Schema Registry server」を起動します（デフォルトではインメモリH2データベースが使用されます）。

> java -jar spring-cloud-schema-registry-server-1.1.2.jar

1. 添付ファイルの「jars」ディレクトリで次のコマンドを実行して、「consumer」を起動します。

> java -jar schema-registry-vanilla-consumer-kafka-0.0.1-SNAPSHOT.jar

1. 添付ファイルの「jars」ディレクトリで次のコマンドを実行して、「producer1」を起動します。

> java -jar schema-registry-vanilla-producer1-kafka-0.0.1-SNAPSHOT.jar

1. 添付ファイルの「jars」ディレクトリで次のコマンドを実行して、「producer2」を起動します。

> java -jar schema-registry-vanilla-producer2-kafka-0.0.1-SNAPSHOT.jar

## テスト

サンプルの両方のプロデューサーはRESTコントローラーでもあります。 各プロデューサーの「/messages」エンドポイントにアクセスして、サンプルデータをPOSTします。コマンドラインで次のコマンドを順番に実行します。

> curl -X POST http://localhost:9009/messages

> curl -X POST http://localhost:9010/messages

> curl -X POST http://localhost:9009/messages

> curl -X POST http://localhost:9009/messages

> curl -X POST http://localhost:9010/messages

「consumer」は結果をログに記録します。

2021-09-15 14:51:34.428  INFO 47156 --- [container-0-C-1] ication$$EnhancerBySpringCGLIB$$f366a201 : input: {"id": "2f43fa71-d173-410b-a174-2d13bc0604f0-v1", "internalTemperature": 25.40218, "externalTemperature": 0.0, "acceleration": 1.5757453, "velocity": 85.76907}

2021-09-15 14:51:43.116  INFO 47156 --- [container-0-C-1] ication$$EnhancerBySpringCGLIB$$f366a201 : input: {"id": "0ef42ee2-38f1-439c-b594-c2374aa80dc8-v2", "internalTemperature": 4.2042494, "externalTemperature": 36.85029, "acceleration": 3.2295876, "velocity": 48.53946}

2021-09-15 14:51:48.559  INFO 47156 --- [container-0-C-1] ication$$EnhancerBySpringCGLIB$$f366a201 : input: {"id": "f30f84a6-a788-4e5e-bf4e-5280f8268dc4-v1", "internalTemperature": 12.17013, "externalTemperature": 0.0, "acceleration": 7.5441265, "velocity": 14.401126}

2021-09-15 14:51:54.622  INFO 47156 --- [container-0-C-1] ication$$EnhancerBySpringCGLIB$$f366a201 : input: {"id": "6b5f48d0-2529-4e34-8e6c-1b5d611e5d0f-v1", "internalTemperature": 21.54079, "externalTemperature": 0.0, "acceleration": 5.877335, "velocity": 28.805155}

2021-09-15 14:52:00.251  INFO 47156 --- [container-0-C-1] ication$$EnhancerBySpringCGLIB$$f366a201 : input: {"id": "b3b7d473-99bc-4a9f-b88f-8856de3ac9ae-v2", "internalTemperature": 36.687607, "externalTemperature": 42.458054, "acceleration": 1.5461105, "velocity": 31.560547}

※「id」フィールドのペイロードサフィックスを参照してください。それぞれに「-v1」または「-v2」が付加され、それぞれ「producer1」および「producer2」からのものであることを示します。

## サンプル説明

スキーマは「temperature」フィールドで進化します。そのフィールドは、2つの別個のフィールドとして、「internalTemperature」と「externalTemperature」に分割されます。「producer1」は「temperature」のみでペイロードを生成しますが、「producer2」は「internalTemperature」フィールドと「externalTemperature」フィールドを含むペイロードを生成します。

「consumer app」は、「internalTemperature」フィールドと「externalTemperature」フィールドを使用してペイロードを正常に逆シリアル化できます。ただし、「producer1」ペイロード（「temperature」フィールドを含む）が到着すると、スキーマの展開と互換性チェックが自動的に適用されます。各ペイロードには、ヘッダーにペイロードバージョンも含まれているため、スキーマレジストリサーバーと「Avro」の助けを借りた「Spring Cloud Stream」は、スキーマの進化が舞台裏で行われます。エイリアスが新しいスキーマで定義されるフィールドであるため、「internalTemperature」フィールドへの「temperature」の自動マッピングが適用されます。

※理解を容易にするために、各プロジェクトのスキーマ定義ファイル「sensor.avsc」を参照できます。

1. 「consumer」のスキーマ定義ファイル「sensor.avsc」

{

"namespace" : "com.example",

"type" : "record",

"name" : "Sensor",

"fields" : [

{"name":"id","type":"string"},

{"name":"internalTemperature", "type":"float", "default":0.0, "aliases":["temperature"]},

{"name":"externalTemperature", "type":"float", "default":0.0},

{"name":"acceleration", "type":"float","default":0.0},

{"name":"velocity","type":"float","default":0.0}

]

}

1. 「producer1」のスキーマ定義ファイル「sensor.avsc」

{

"namespace" : "com.example",

"type" : "record",

"name" : "Sensor",

"fields" : [

{"name":"id","type":"string"},

{"name":"temperature", "type":"float", "default":0.0},

{"name":"acceleration", "type":"float","default":0.0},

{"name":"velocity","type":"float","default":0.0}

]

}

1. 「producer2」のスキーマ定義ファイル「sensor.avsc」

{

"namespace" : "com.example",

"type" : "record",

"name" : "Sensor",

"fields" : [

{"name":"id","type":"string"},

{"name":"internalTemperature", "type":"float", "default":0.0, "aliases":["temperature"]},

{"name":"externalTemperature", "type":"float", "default":0.0},

{"name":"acceleration", "type":"float","default":0.0},

{"name":"velocity","type":"float","default":0.0},

{"name":"accelerometer","type":[

"null",{

"type":"array",

"items":"float"

}

]},

{"name":"magneticField","type":[

"null",{

"type":"array",

"items":"float"

}

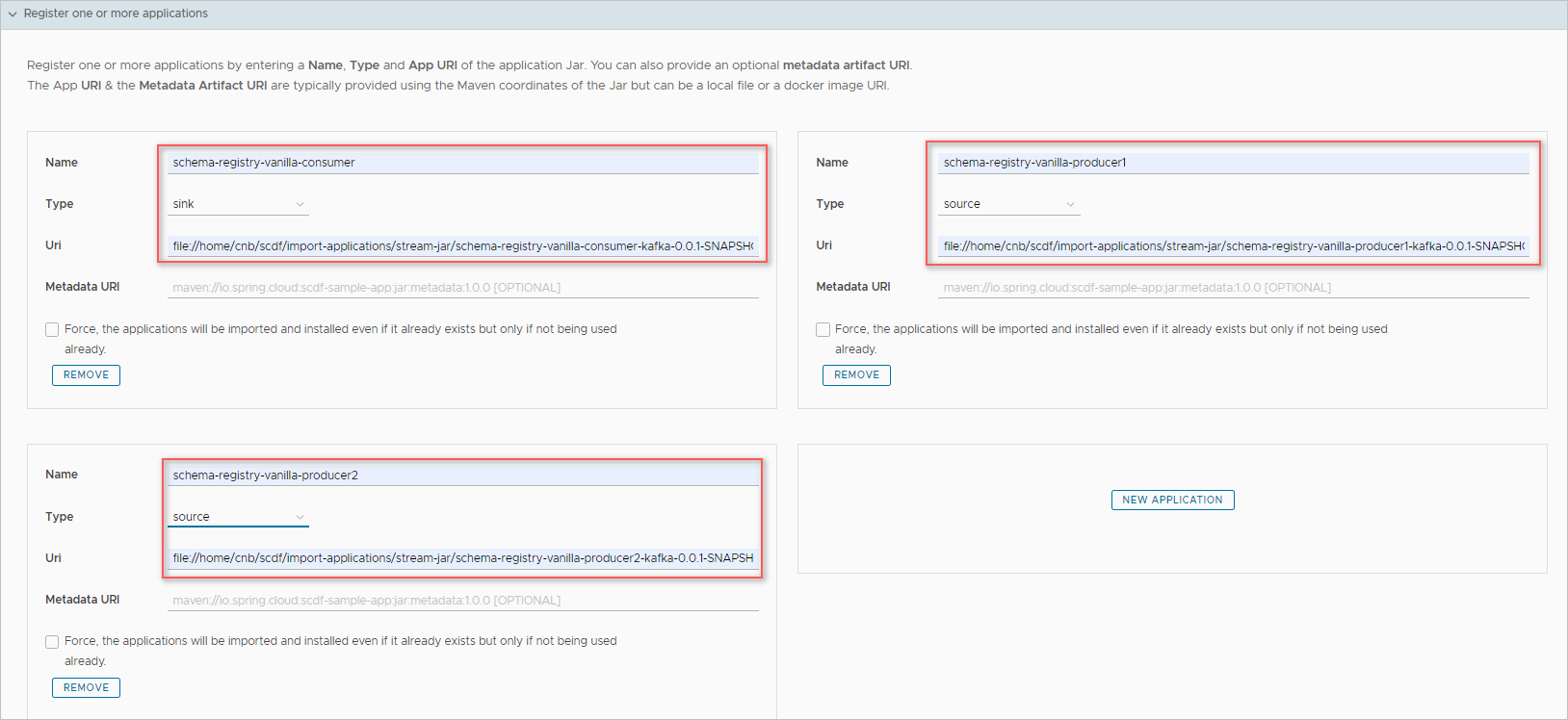
]}

]

}

# SCDFと組み合わせ

「[SCDF] 100aAPP開発環境構築Hand-Shun.xlsx」シート「カスタマイズプリケーションのログイン」を参照し、scdfに「schema-registry-vanilla-consumer-kafka-0.0.1-SNAPSHOT.jar」、「schema-registry-vanilla-producer1-kafka-0.0.1-SNAPSHOT.jar」、「schema-registry-vanilla-producer2-kafka-0.0.1-SNAPSHOT.jar」を登録してください。登録名は「schema-registry-consumer」、「schema-registry-producer1」、「schema-registry-producer2」です。



## ストリームを作成して実行する

1. 添付ファイルの「jars」ディレクトリで次のコマンドを実行して、「Schema Registry server」を起動します（デフォルトではインメモリH2データベースが使用されます）。

> java -jar spring-cloud-schema-registry-server-1.1.2.jar

1. SCDFでそれぞれ次のストリームを作成します。

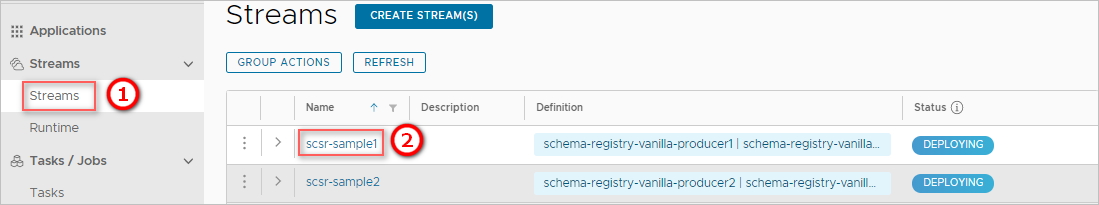
|  |  |
| --- | --- |
| **Name** | **Definition** |
| scsr-sample1 | schema-registry-vanilla-producer1 --spring.cloud.schema-registry-client.endpoint=http://IP:8990 | schema-registry-vanilla-consumer --spring.cloud.schema-registry-client.endpoint=http://IP:8990 |
| scsr-sample2 | schema-registry-vanilla-producer2 --spring.cloud.schema-registry-client.endpoint=http://IP:8990 | schema-registry-vanilla-consumer --spring.cloud.schema-registry-client.endpoint=http://IP:8990 |

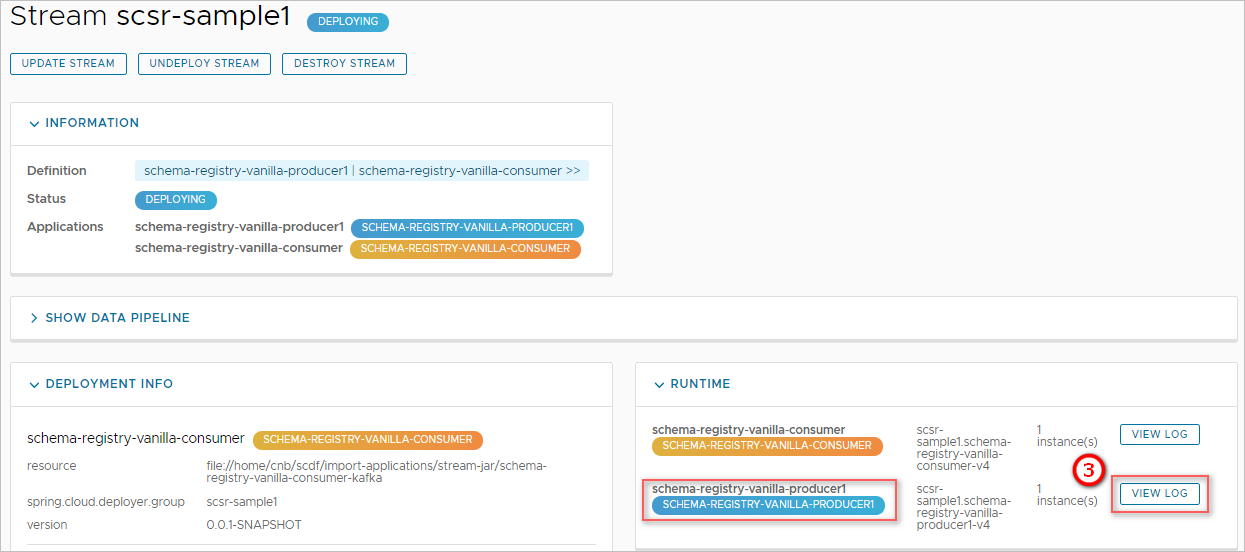
※「IP」を「ローカルIPアドレス」に変更します。

1. ストリーム「scsr-sample1」とストリーム「scsr-sample1」をデプロイします。

## テスト

1. 以下の手順に従って、それぞれ「schema-registry-vanilla-producer1」と「schema-registry-vanilla-producer2」のポートを取得します。







この例では、「schema-registry-vanilla-producer1」のポートは「20083」であり、「schema-registry-vanilla-producer2」のポートは「20060」です。

1. 各プロデューサーの「/messages」エンドポイントにアクセスして、サンプルデータをPOSTします。

例：コマンドラインで次のコマンドを順番に実行します。

> curl -X POST http://localhost:20083/messages

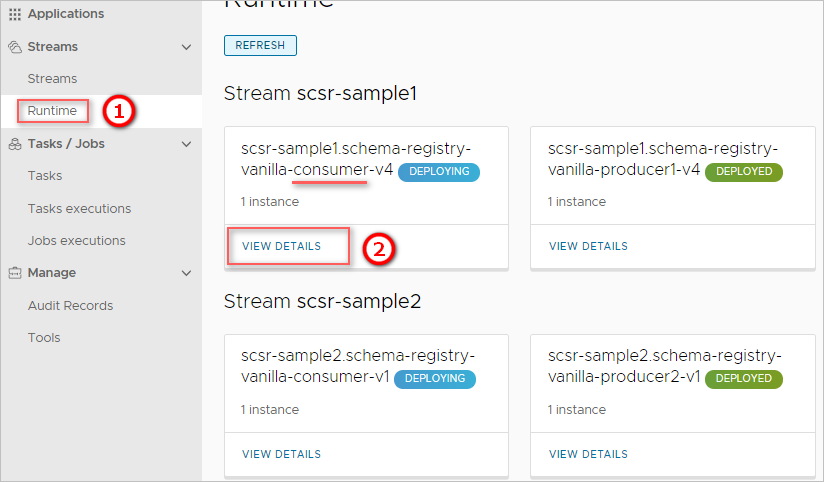
> curl -X POST http://localhost:20060/messages

> curl -X POST http://localhost:20083/messages

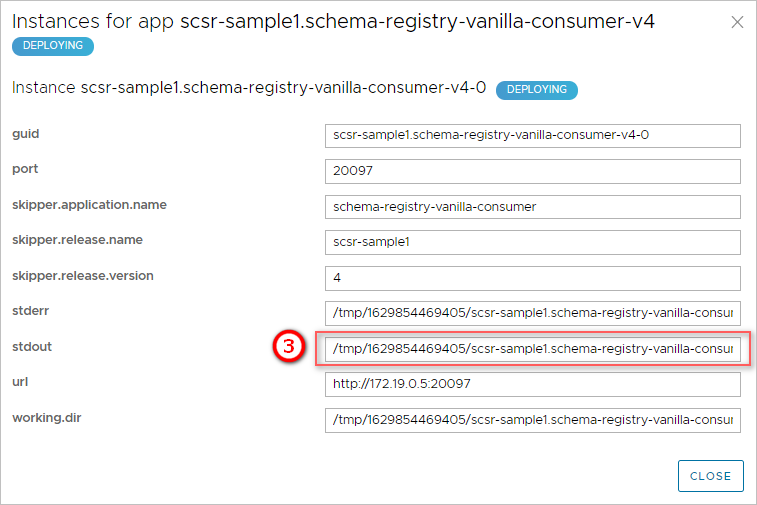
> curl -X POST http://localhost:20083/messages

> curl -X POST http://localhost:20060/messages

1. 以下の手順に従って、「schema-registry-consumer」のログに次のログを表示します。



ダッシュボードの「stdout」テキストボックスにパスをコピーします。



コマンドラインで、次のコマンドを実行します。

※「/path/from/stdout/textbox/in/dashboard」を前の手順でコピーした値に置き換えます。

> docker exec -it skipper tail /path/from/stdout/textbox/in/dashboard

「schema-registry-consumer」のログに次のログを表示します。

