



コンポーネントSDK 4.2 開発ガイド

DataSpider Servista Component SDK 4.2 Getting Started

ドキュメント・リリース日: 2019 年 7 月 3 日 (第 1 版)



目次

1. はじめに	1
1.1. 注意事項	1
1.1.1. お客様へのお願い	1
1.1.2. 商標について	1
1.2. 表記について	2
1.3. マークについて	2
2. 開発環境の設定	3
2.1. アプリケーションの準備	3
2.2. 環境変数の設定	3
2.3. DATASPIDER_HOME の設定	4
3. Servista Component 概要	5
3.1. Servista Component の概要	5
3.2. アダプタを構成する要素	5
3.2.1. DataProcessingModuleComponent	5
3.2.2. OperationFactory	5
3.2.3. Operation	5
4. オペレーションのライフサイクルについて	7
5. アダプタの作成	8
5.1. Hello World Adapter の作成	8
5.1.1. 概要	8
5.1.2. アダプタの作成	9
5.1.3. アダプタのビルド	13
5.1.4. アダプタのインストール	15
5.1.5. アダプタの動作確認	15
5.2. MessageXML Adapter の作成	17
5.2.1. 概要	17
5.2.2. アダプタの作成	18
5.2.3. アダプタのビルド	24
5.2.4. アダプタのインストール	25
5.2.5. アダプタの動作確認	27
5.3. Print Adapterの作成	28
5.3.1. 概要	28
5.3.2. アダプタの作成	28
5.3.3. アダプタのビルド	37

5.3.4. アダプタのインストール	38
5.3.5. アダプタの動作確認	39
6. トラブルシューティング	40
7. グローバルリソースについて	42
7.1. グローバルリソースのクラス構成	42
7.2. グローバルリソースの実装方法	42
7.3. グローバルリソースを利用するアダプタの作成	43
7.4. アダプタのビルド	51
7.5. アダプタのインストール	53
7.6. アダプタの動作確認	53
8. XML フレームワークのデータ型について	55
8.1. XML フレームワークのデータ型	55
8.1.1. テーブルモデル型	55
8.1.2. XML データ型	56
9. 大容量データ処理について	57
9.1. 大容量データ処理のプロパティ実装	57
9.2. プロパティで設定された値の取得	57
9.3. 大容量データ処理に対応した読み取り処理	57
9.3.1. テーブルモデル型の読み取り処理	58
9.3.2. XML 型の大容量データ処理対応読み取り処理	59
9.4. 大容量データ処理対応の書き込み処理	59
9.4.1. テーブルモデル型の大容量データ処理対応書き込み処理	59
9.4.2. XML型の大容量データ処理対応書き込み処理	60
10. トランザクションについて	61
10.1. コンポーネントでサポートするトランザクション	61
11. 高度なプロパティ設定	62
11.1. テーブルプロパティ	62
11.1.1. テーブルプロパティの実装	63
11.1.2. テーブルプロパティからの値の取得	64
11.2. 接続テストプロパティ	64
11.2.1. アクションボタンの実装	64
11.2.2. リスナーの定義	65
11.2.3. アクションの確認	67
11.3. カスタムプロパティ制約によるプロパティ制約の拡張	67
11.3.1. カスタムプロパティ制約の作成	68
11.3.2. カスタムプロパティ制約専用エディタの作成	69

12. プロパティ制約について	74
12.1. Fillin	74
12.2. FileInputFillin	75
12.3. NumberFillin	75
12.4. MultipleLineFillin	76
12.5. PasswordFillin	76
12.6. Multi	76
12.7. FillinMulti	77
12.8. CharsetNameFillinMulti	77
12.9. CheckBox	77
12.10. InformationConstraint	77
13. 入出力制約について	79
13.1. 入出力制約の定義方法	79
13.2. XML 型の入出力制約を定義する際のスキーマについて	80
14. エラーコンポーネント変数の設定	81
14.1. エラーコンポーネント変数とは	81
14.2. エラーコンポーネント変数の実装	82
14.3. 独自のエラーコンポーネント変数の実装	83
15. メッセージコードについて	85
16. エラーチェックについて	87
16.1. プロパティの設定が不正な場合	87
17. 外部ライブラリの利用について	88
18. オペレーションとグローバルリソースのアイコンファイルについて	89
19. パスワード取得時の注意点	90
20. ログ出力指針	91
21. 推奨されていない API について	92
22. 言語切り替え対応について	93
22.1. アダプタの言語切り替え対応	93
22.2. ヘルプドキュメントの言語切り替え対応	93
23. 2.3.x から 2.4.0 の変更点	94
24. 2.4.x から 3.0 の変更点	95
24.1. パッケージの変更	95
25. 3.0 から 3.1 の変更点	96
26. 3.1 から 3.2 の変更点	97
27. 3.2 から 4.0 の変更点	98
27.1. 言語切り替え機能の追加	98

27.2. 非推奨メソッドの追加	98
28. 4.0 から 4.0SP1 の変更点	99
28.1. LoggingContext のメソッド追加	99
29. 4.0SP1 から 4.1 の変更点	100
30. 4.1 から 4.2 の変更点	101
31. ヘルプドキュメントの作成	102
31.1. アダプタのヘルプドキュメントの記述ルール	102
31.2. アダプタのヘルプドキュメントを追加する	102
31.3. 追加したヘルプドキュメントを登録する	103
31.3.1. ヘルプセットファイルの編集	103
31.3.2. マップファイルの編集	104
31.3.3. TOC ファイルの編集	105
31.4. ヘルプインデックスを再構築する	106
31.5. 英語環境用ヘルプドキュメントを作成する	106
31.5.1. 英語環境用ヘルプドキュメント編集	107
32. クラス図	108

1. はじめに

本ドキュメントでは、DataSpider Servista で提供されているAPI (Application Programming Interface) を用いて、スクリプトで処理を行う Component (アダプタ)を作成するために必要な情報を提供していきます。

本ドキュメントで使用されている\$SDK_HOMEとは、SDK をインストールしたディレクトリを表しています。

1.1. 注意事項

1.1.1. お客様へのお願い

- 本ソフトウェアの著作権は株式会社セゾン情報システムズまたはそのライセンサーが所有しています。
- 本ソフトウェアおよび本ドキュメントを無断で複製、転載することを禁止します。
- 本ドキュメントは万全を期して作成されていますが、万一不明な点や誤り、記載もれなど、お気づきの点がございましたら弊社までご連絡ください。
- 本ソフトウェアは使用者の責任でご使用ください。ご使用の結果、万トラブルおよび訴訟などが発生し、または、あらゆる直接、または間接の損害および損失につきまして、弊社は一切責任を負わないものとします。あらかじめご了承ください。
- 本ソフトウェアの仕様や本ドキュメントに記載されている内容は、改善のため予告なしに変更されることがあります。
- 本ソフトウェアの使用には、ソフトウェアライセンス契約が必要で、株式会社セゾン情報システムズまたはそのライセンサーの重要な業務機密と独自の情報が含まれており、日本国政府の著作権法で保護されています。株式会社セゾン情報システムズまたはそのライセンサーのソフトウェアと本ドキュメントの無断使用は、損害賠償、刑事訴訟の対象となります。

1.1.2. 商標について

- DataSpider、DataSpider ロゴ、DataSpider Servista、その他の関連製品名、サービス名などは、株式会社セゾン情報システムズの登録商標または商標です。
- その他記載されている会社名・商品名・サービス名などは、各社の商標および登録商標です。
- 個々のページに表示・記載されたこれら商標などの複製・転用を禁止致します。




1.2. 表記について

本ドキュメント内の表記は、次の規則に沿って行われています。

- DataSpider Servista の画面に表示されるメニュー名・タブ名・プロパティ項目名および値・ボタン名は [] で囲んで太字で表します。また、それ以外の機能名や画面のタイトル、名称のないものは「」で囲んで前者と区別しています。
- 「\$DATASPIDER_HOME」は DataSpider Servista をインストールしたディレクトリを表します。
デフォルトでは、Windows 版の場合は「C:\Program Files\DataSpiderServista」、UNIX/Linux 版の場合は「<ユーザのホームディレクトリ>/DataSpiderServista」となります。
- x86 版とは、32bit OS を表します。
x64 版とは、64bit(Intel 64/AMD64) OS を表します。
- <と> で囲まれた名称は、可変であることを表します。
例:\$DATASPIDER_HOME/server/logs/<日付ディレクトリ>
- 「Studio」とは「DataSpider Studio」を、「Studio for Web」とは「DataSpider Studio for Web」を指します。
- DataSpiderServer についての記述は Windows 版・UNIX/Linux 版共通になっています。
オペレーティングシステムに依存する内容 (パス区切り文字など) は適宜読み替えてご使用ください。
- 「DSS-」で始まる番号は、各問題の管理用の一意な ID となります。

1.3. マークについて

本ドキュメント内で使用しているマークについての説明は以下の通りです。

マーク	説明
	操作や設定に関するヒントであることを表します。
	操作や設定に関する注意事項や制限事項であることを表します。
	詳細な説明が別の項目に記載されていることを表します。

2. 開発環境の設定

2.1. アプリケーションの準備

アダプタを作成するためには、以下のアプリケーションが準備されている必要があります。

- Java Platform, Standard Edition 8 Development Kit
- Apache ant 1.7.0 以降
- DataSpider Servista 4.2

2.2. 環境変数の設定

以下の環境変数が設定されていることを確認してください。

- JAVA_HOME (JDK がインストールされているディレクトリ)
- ANT_HOME (ANT がインストールされているディレクトリ)

上記の環境変数が設定できたら、以下のディレクトリにパスを通します。(PATH 環境変数に追加します)

- %JAVA_HOME%\bin (UNIX の場合は \$JAVA_HOME/bin)
- %ANT_HOME%\bin (UNIX の場合は \$ANT_HOME/bin)

上記の準備が正しく設定されているかを確認します。コマンドプロンプトから以下のコマンドが実行できることを確認してください。

- java -version
- ant -version

それぞれのコマンドを実行して、適切なバージョンが表示されていれば、アダプタの開発環境の設定は完了です。

バージョンが表示されない場合は、環境変数の設定が正しく行われていない可能性があります。

JAVA_HOME、ANT_HOME、PATH それぞれの環境変数が正しく設定されているか確認し、適切なバージョンが表示されるように修正してください。

2.3. DATASPIDER_HOME の設定

\$SDK_HOME/dev にある、build.properties ファイルの dataspider.home プロパティに、DataSpider Servista をインストールしたディレクトリパスを設定します。

3. Servista Component 概要

この章では、Servista Component の概要と、構成する要素について説明します。

3.1. Servista Component の概要

Servista Component とは、スクリプトで使用可能な処理や設定群を包括したモジュールを指します。

以降、Servista Component をアダプタまたはコンポーネントと呼称します。

アダプタは、オペレーションあるいはグローバルリソースで構成され、スクリプトからオペレーションで実装した処理を実行します。

また、オペレーションでは再利用可能なグローバルリソースを利用できます。グローバルリソースについては「[7. グローバルリソースについて](#)」をご参照ください。

3.2. アダプタを構成する要素

3.2.1. DataProcessingModuleComponent

- 別名：モジュールコンポーネント
- コンポーネント（アダプタ）を表すクラス
- オペレーションおよびリソースとは多対 1 の関係

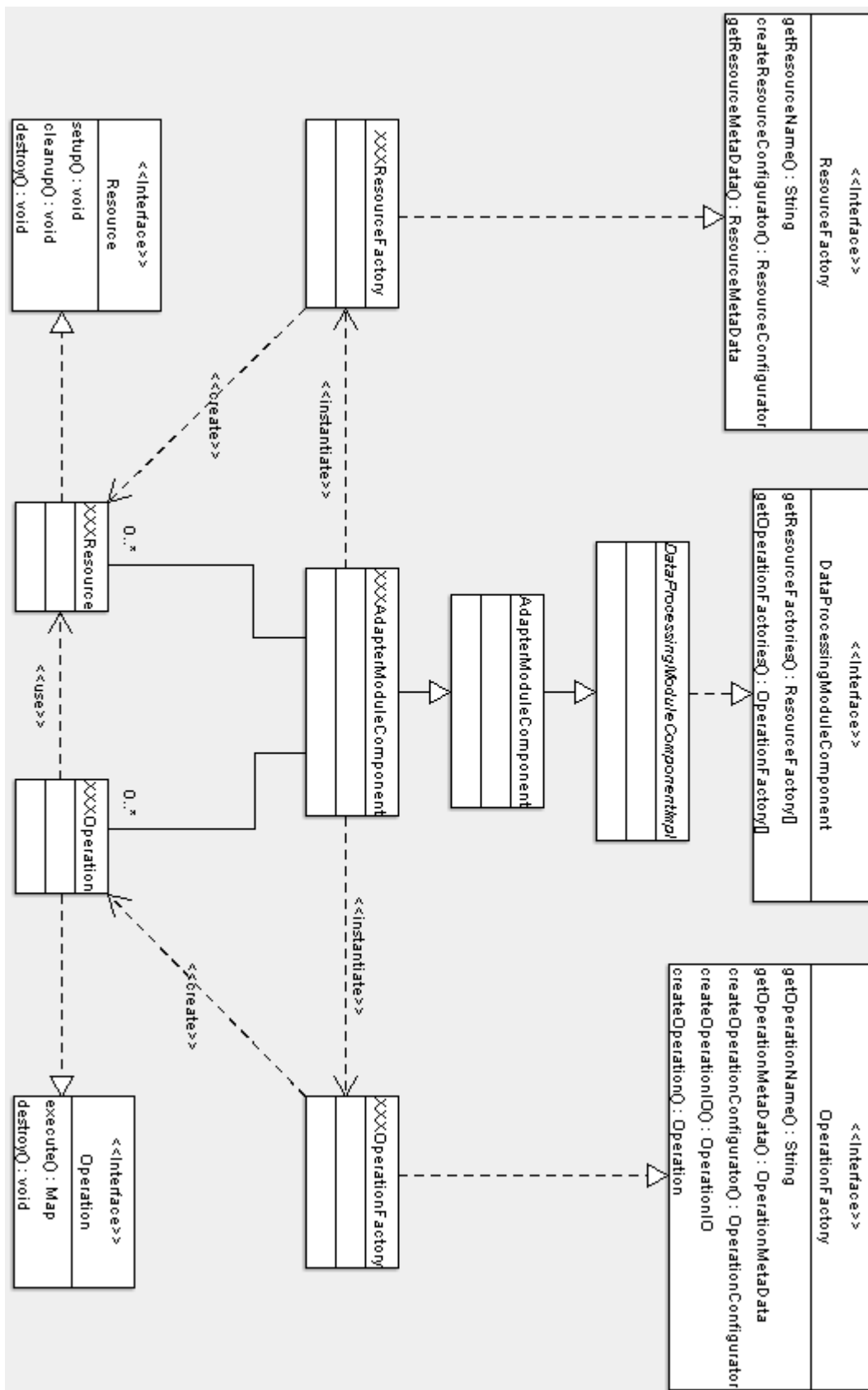
3.2.2. OperationFactory

- 別名：オペレーションファクトリ
- アダプタのプロパティを定義
- スクリプト上でやりとりする入出力データの型を定義
- オペレーションの生成

3.2.3. Operation

- 別名：オペレーション
- アダプタの主要機能となるデータの読み取りや変換、書き込みなどの個々の処理を提供

各クラスの関連は、当ページの「コンポーネントクラス図」をご参照ください。



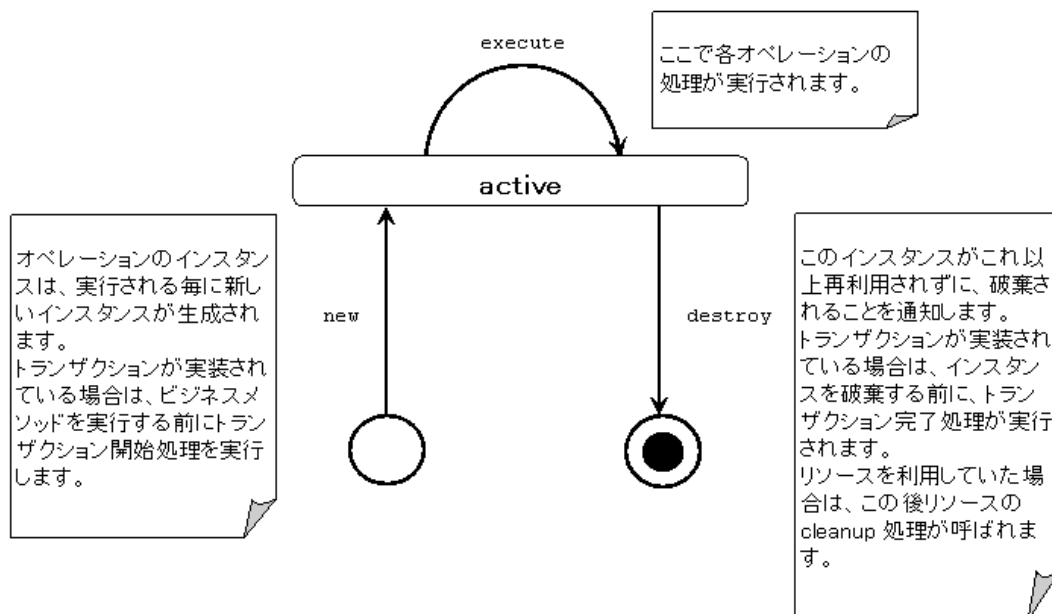
1 コンポーネントクラス図

4. オペレーションのライフサイクルについて

この章では、スクリプト実行時に生成されるオペレーションインスタンスのライフサイクルについて説明します。

オペレーションは、処理が実行されるたびに必ず新しいインスタンスを生成します。オペレーションインスタンスが生成されると、そのオペレーションで実装されている各処理が実行されます。当該処理が終了するとインスタンスは破棄されます。

オペレーションがトランザクションを実装している場合は、オペレーションを実行する前にトランザクション開始処理が実行され、destroy が呼ばれる前にトランザクション終了処理が実行されます。トランザクションの実装に関しては、「[10. トランザクションについて](#)」をご参照ください。



2 コンポーネントライフサイクル

5. アダプタの作成

5.1. Hello World Adapter の作成

5.1.1. 概要

基本情報

アダプタ名	Hello World Adapter
モジュール名	hello_world_adapter
パッケージ名	com.appresso.ds.dp.modules.adapter.helloworld
クラス構成	HelloWorldAdapterOperation HelloWorldAdapterOperationFactory HelloWorldAdapterModuleComponent

オペレーション情報

読み取りオペレーション	Hello World を出力
書き込みオペレーション	なし
読み取りプロパティ	なし
書き込みプロパティ	なし

このアダプタを作成することで、以下の項目に関して学習することができます。

- アダプタを定義するクラス構成
- 実装対象となるオペレーションの種類
- ログの種類と出力方法
- アダプタのビルド、デプロイ方法
- アダプタの利用方法

なお、このアダプタのサンプルは以下のディレクトリにあります。

- \$SDK_HOME/samples/hello_world_adapter

5.1.2. アダプタの作成

1. モジュールディレクトリの作成

\$SDK_HOME/dev 直下に hello_world_adapter という名前のディレクトリを作成します。

- \$SDK_HOME/dev/hello_world_adapter

2. build.xml、config.properties のコピー

\$SDK_HOME/dev/conf ディレクトリにある、build.xml ファイル と config.properties ファイルをモジュールディレクトリにコピーします。

3. config.properties ファイルの編集

以下のプロパティを設定します。

Implementation-Vendor	APPRESSO K.K.
module.label	HelloWorld
module.category	SAMPLE
display.name	Hello World Adapter

4. ソースディレクトリの作成

\$SDK_HOME/dev/hello_world_adapter に src ディレクトリを作成します。

- \$SDK_HOME/dev/hello_world_adapter/src

5. ソースファイルの作成

ソースディレクトリに以下のファイル名でソースファイルを作成します。

- HelloWorldAdapterOperation.java
- HelloWorldAdapterOperationFactory.java
- HelloWorldAdapterModuleComponent.java

6. HelloWorldAdapterOperation の実装

HelloWorldAdapterOperation は DataSpider Servista で提供している Operation インターフェースを実装したクラスです。

HelloWorldAdapterOperation では execute メソッドを実装して、本機能である Hello World を出力する部分を記述します。

execute メソッドは、Map オブジェクトに何らかのオブジェクトを put して戻り値として返した場合、ほかのアダプタの入力データとしてそのオブジェクト利用することができます。Hello World Adapter では、特に取得されるものはないので、戻り値の Map オブジェクトには何も put せずに返します。

メッセージの出力には以下のクラスを使います。

- クラス: `LoggingContext`
- パッケージ: `com.appresso.ds.common.fw`
- インスタンスの取得: `OperationContext` クラスの `getLoggingContext` メソッド
- メソッド:
 - `info(String)` 結果ログを出力します。
 - `finfo(String)` 細かい結果ログを出力します。
 - `warn(String)` 警告ログを出力します。
 - `fwarn(String)` 細かい警告ログを出力します。
 - `finest(String)` 詳細なログを出力します。
 - `debug(String)` デバッグ情報を出力します。

`HelloWorldAdapterOperation` は、`import` 文で以下のパッケージ名を指定します。(※)

- `com.appresso.ds.common.fw.*`
- `com.appresso.ds.dp.spi.*`
- `java.util.*`

※ サンプルでは FQN で指定しています。

`LoggingContext` クラスを用いてメッセージを出力すると、その内容がクライアントにコールバックされます。これらをもとに、`execute` メソッドを以下のように実装します。

HelloWorldAdapterOperation.java

```

1 package com.appresso.ds.dp.modules.adapter.helloworld;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 import com.appresso.ds.common.fw.LoggingContext;
7 import com.appresso.ds.dp.spi.Operation;
8 import com.appresso.ds.dp.spi.OperationContext;
9
10 public class HelloWorldAdapterOperation implements Operation {
11     private final OperationContext context;
12
13     public HelloWorldAdapterOperation(OperationContext context) {
14         this.context = context;
15     }
16
17     @Override
18     public Map execute(Map inputData) throws Exception {
19         LoggingContext log = context.log();
20         log.finest("*****");
21         log.info("** Hello World ! **");
22         log.warn("** execute() ! **");
23         log.finest("*****");
24
25         return new HashMap();
26     }
27
28     @Override
29     public void destroy() {
30     }
31 }

```

7. HelloWorldAdapterOperationFactory の実装

HelloWorldAdapterOperationFactory は、アダプタのプロパティを保持するオブジェクトを返したり、オペレーションオブジェクトを返したりする Factory クラスです。

クラスの関係は、「[コンポーネントクラス図](#)」をご参照ください。

HelloWorldAdapterOperationFactory では、プロパティの設定はなく、またデータの入出力機能も実装しないため、以下のメソッドの戻り値は null を返します。

- createOperationIO
- createOperationConfigurator

HelloWorldAdapterOperationFactory は、import 文で以下のパッケージ名を指定します。(※)

- com.appresso.ds.dp.spi.*

※ サンプルでは FQN で指定しています。

HelloWorldAdapterOperationFactory の実装は以下の通りです。

HelloWorldAdapterOperationFactory.java

```


1 package com.appresso.ds.dp.modules.adapter.helloworld;
2
3 import com.appresso.ds.dp.spi.GetDataOperationFactory;
4 import com.appresso.ds.dp.spi.Operation;
5 import com.appresso.ds.dp.spi.OperationConfiguration;
6 import com.appresso.ds.dp.spi.OperationConfigurator;
7 import com.appresso.ds.dp.spi.OperationContext;
8 import com.appresso.ds.dp.spi.OperationIO;
9
10 public class HelloWorldAdapterOperationFactory extends GetDataOperationFactory {
11     @Override
12     public OperationConfigurator createOperationConfigurator(
13         OperationConfiguration conf, OperationContext context) throws Exception {
14
15         // プロパティは実装しないので、null を返します。
16         return null;
17     }
18
19     @Override
20     public OperationIO createOperationIO(
21         OperationConfiguration conf, OperationContext context) throws Exception {
22
23         // データの入出力はしないので、null を返します。
24         return null;
25     }
26
27     @Override
28     public Operation createOperation(
29         OperationConfiguration conf, OperationContext context) throws Exception {
30
31         // オペレーションを返します。
32         return new HelloWorldAdapterOperation(context);
33     }
34 }

```

8. HelloWorldAdapterModuleComponent の実装

HelloWorldAdapterModuleComponent は AdapterModuleComponent クラスを継承しており、コンポーネントが実装している OperationFactory の派生クラスおよび、ResourceFactory の派生クラスを配列として返すメソッドを実装します。

HelloWorld Adapter は ResourceFactory の派生クラスを実装していないので、OperationFactory の派生クラスである HelloWorldAdapterOperationFactory を配列として返すメソッドを実装します。HelloWorldAdapterModuleComponent のクラス図は「[コンポーネントクラス図](#)」をご参照ください。

 **AdapterModuleComponent を継承しているクラスは、必ず<アダプタ名> ModuleComponent.java というファイル名で作成してください。**

HelloWorldAdapterModuleComponent は、import 文で以下のパッケージ名を指定します。(※)

- com.appresso.ds.dp.spi.*

※ サンプルでは FQN で指定しています。

HelloWorldAdapterModuleComponent の実装は以下のようになります。

HelloWorldAdapterModuleComponent.java

```
1 package com.appresso.ds.dp.modules.adapter.helloworld;
2
3 import com.appresso.ds.dp.spi.AdapterModuleComponent;
4 import com.appresso.ds.dp.spi.OperationFactory;
5
6 public class HelloWorldAdapterModuleComponent extends AdapterModuleComponent {
7     @Override
8     public OperationFactory[] getOperationFactories() throws Exception {
9         return new OperationFactory[] { new HelloWorldAdapterOperationFactory() };
10    }
11 }
```

5.1.3. アダプタのビルド

1. アダプタのビルド

アダプタのビルドは、\$SDK_HOME/dev/hello_world_adapter ディレクトリで以下のコマンドを実行します。

- コマンド: ant

コマンド実行後、\$SDK_HOME/dev/hello_world_adapter/build ディレクトリに、hello_world_adapter.jar と

いう jar ファイルが生成されていれば、ビルドは成功です。

以前のビルドで生成されたjarファイルを削除する場合は、ant コマンドの第一引数に distclean ターゲットを指定して実行します。

- コマンド: ant distclean


2. アイコンファイルのコピー

アダプタのビルドに成功すると、\$SDK_HOME/dev/hello_world_adapter/META-INF ディレクトリが生成されます。そのディレクトリに \$SDK_HOME/dev/conf ディレクトリにある、以下のファイルをコピーしてください。

- operation.get_data.icon

3. module.properties ファイルの確認

\$SDK_HOME/dev/hello_world_adapter/META-INF/module.properties ファイルが正常に作成されたことを確認します。このプロパティファイルは、\$SDK_HOME/dev/hello_world_adapter/config.properties ファイルで定義されているプロパティをもとに作成されています。それぞれのプロパティの内容は以下の通りです。

module.category	<p>アダプタがどのカテゴリに属するかを指定します</p> <ul style="list-style-type: none"> • file : ファイル系アダプタ • database : データベース系アダプタ • directoryservice : ディレクトリサービス系アダプタ • network : ネットワーク系アダプタ • application : アプリケーション系アダプタ • mapper : 変換系アダプタ • cloud : クラウド系アダプタ • bigdata : ビッグデータ系アダプタ <p> 任意のカテゴリ名も定義できます。(2 バイト文字は不可)</p>
component.class	<p>AdapterModuleComponent の派生クラスを完全修飾名 (※) で指定します ※パッケージ名を含むクラス名</p>
module.label	<p>アダプタのラベル名を指定します</p>
display.name	<p>アダプタの表示名を指定します</p>

5.1.4. アダプタのインストール

アダプタのインストールは \$SDK_HOME/dev/hello_world_adapter から ant コマンドの第一引数に install ターゲットを指定して実行します。

- コマンド: ant install

ant install を実行後、\$DATASPIDER_HOME/server/plugin/data_processing/modules ディレクトリ以下にアダプタがインストールされます。インストールしたアダプタを有効にするには、DataSpiderServer の再起動が必要です。

5.1.5. アダプタの動作確認

アダプタのインストールが完了したら、アダプタの動作確認を行います。DataSpiderServer を再起動したあと、DataSpider Studio から、サーバへの再接続を行ってください。そのあとの手順は以下の通りです。

1. デザイナから任意のプロジェクトを開きます。
2. 新規スクリプトを作成します。
3. ツールパレットに、module.properties ファイルの module.category プロパティで指定したカテゴリ (SAMPLE) に入っていることを確認します。
4. カテゴリ内に、module.properties ファイルの module.label プロパティで指定したラベル名で存在していることを確認します。
5. 「読み取り」というオペレーションを、選択し、スクリプトキャンバス上にドラッグ & ドロップします。
6. プロパティを設定するダイアログが開きます。Hello World Adapter はプロパティを持っていないので設定はしません。名前を入力し「完了」を押下すると、アダプタが、スクリプトキャンバスに配置されます。
7. [start] - [HelloWorldAdapter] - [end] の順にプロセスフローを引きます。プロセスフローはフロー元のアイコンを選択して、フロー先のアイコンにドラッグ & ドロップすることで引くことができます。
8. デザイナのメニューバーから、[ツール] - [オプション]でログレベルが「FINEST」となっていない場合、「FINEST」へ変更します。
9. デザイナのメニューバーから、[テスト実行] - [デバッグ実行の開始/再開]または「F5」でスクリプトを実行します。
10. 以下のログがデザイナの実行ログで確認できれば成功です。

```
*****  
** Hello World ! **  
** execute() ! **  
*****
```

5.2. MessageXML Adapter の作成

5.2.1. 概要

基本情報

アダプタ名	MessageXML Adapter
モジュール名	message_xml_adapter
パッケージ名	com.appresso.ds.dp.modules.adapter.messagexml
クラス構成	MessageXMLAdapterOperation MessageXMLAdapterOperationFactory MessageXMLAdapterModuleComponent

オペレーション情報

読み取りオペレーション	プロパティで設定したメッセージを CDATA として持つ XML 文書を表すオブジェクトを生成し、それを Map オブジェクトに put して戻り値として返す
書き込みオペレーション	なし
読み取りプロパティ	出力されるメッセージ
書き込みプロパティ	なし

このアダプタを作成することで、以下の項目に関して学習することができます。

- 読み取りオペレーションの実装方法
- プロパティの作成
- オペレーションの実装方法
- 結果データの作成・出力方法

なお、このアダプタのサンプルは以下のディレクトリにあります。

- \$SDK_HOME/samples/message_xml_adapter

5.2.2. アダプタの作成

1. モジュールディレクトリの作成

\$SDK_HOME/dev 直下に message_xml_adapter という名前のディレクトリを作成します。

- \$SDK_HOME/dev/message_xml_adapter

2. build.xml、config.properties のコピー

\$SDK_HOME/dev/conf ディレクトリにある、build.xml ファイルと config.properties ファイルをモジュールディレクトリにコピーします。

3. config.properties ファイルの編集

以下のプロパティを設定します。

Implementation-Vendor	APPRESSO K.K.
module.label	MessageXML
module.category	SAMPLE
display.name	Message XML Adapter

4. ソースディレクトリの作成

\$SDK_HOME/dev/message_xml_adapter に src ディレクトリを作成します。

- \$SDK_HOME/dev/message_xml_adapter/src

5. ソースファイルの作成

ソースディレクトリに以下のファイル名でソースファイルを作成します。

- MessageXMLAdapterOperation.java
- MessageXMLAdapterOperationFactory.java
- MessageXMLAdapterModuleComponent.java

6. MessageXMLAdapterOperationFactory の実装

MessageXMLAdapterOperationFactory は、GetDataOperationFactory の派生クラスであり、アダプタのプロパティを保持するオブジェクトを返したり、オペレーションオブジェクトを返したりする Factory クラスです。以下の実装を用意する必要があります。

- プロパティを作成する
- 出力データが XML であることを定義する

c. オペレーションを作成する

import 文で以下のパッケージ名を指定する必要があります。(※)

- com.appresso.ds.common.spi.param.*
- com.appresso.ds.common.spi.constraint.*

※ サンプルでは FQN で指定しています。

a. プロパティを作成する

MessageXML Adapter では、メッセージ (文字列) を入力するプロパティを実装する必要があります。

プロパティ制約には以下のクラスを利用します。

- クラス: Fillin
- パッケージ: com.appresso.ds.common.spi.constraint

 プロパティ制約に関する詳細は「[12. プロパティ制約について](#)」をご参照ください。

実装は下記ようになります。

MessageXMLAdapterOperationFactory.java

```
14  static final String KEY_MESSAGE = "MESSAGE";

49  // プロパティを表すパラメータの作成
50  private SimpleParameter createMessageParameter() {
51
52      // パラメータに自由入力型のプロパティ制約を定義します。
53      Fillin fillin = new Fillin();
54      fillin.setLabel("メッセージ");
55      fillin.setRequired(true);
56      fillin.setShortcut("M");
57      fillin.setDescription("出力メッセージを指定");
58
59      return new SimpleParameter(KEY_MESSAGE, fillin);
60  }
```

createMessageParameter メソッドではまず Fillin オブジェクトを生成します。Fillin オブジェクトには、プロパティの名前、そのプロパティが必須かどうか、プロパティの説明、プロパティのショートカットなど

が設定できます。

戻り値として SimpleParameter オブジェクトを返しますが SimpleParameter をインスタンス化する際に、プロパティ制約のオブジェクト (この場合 Fillin オブジェクト) とそのプロパティ制約のキーとなる文字列を渡します。こうすることで、プロパティに入力された値 (MessageXMLAdapter の場合はメッセージ) はキーを指定することで取得することが可能になります。

次に、上記で作成した SimpleParameter を Configurator クラスの派生クラスである OperationConfigurator に追加する必要があります。Configurator クラスは主に、パラメータ (プロパティ) に関する設定方法を表しています。

OperationConfigurator オブジェクトの生成およびそれを返すメソッドの実装は以下のようになります。

MessageXMLAdapterOperationFactory.java

```
18 public OperationConfigurator createOperationConfigurator(  
19     OperationConfiguration conf, OperationContext context) throws Exception {  
20  
21     // オペレーションの設定方法 (プロパティ) を表すクラスです。  
22     OperationConfigurator configurator = new OperationConfigurator(conf, context);  
23  
24     // パラメータ (プロパティ) を追加します。  
25     configurator.addSimpleParameter(createMessageParameter());  
26  
27     return configurator;  
28 }
```

createOperationConfigurator メソッドは、OperationFactory インターフェースで定義されているメソッドであり、OperationFactory インターフェースの派生クラスである MessageXMLAdapterOperationFactory はこのメソッドを実装する必要があります。

このメソッド内で、OperationConfigurator を生成し、createMessageParameter メソッドの戻り値である SimpleParameter オブジェクトを add します。

b. 出力データが XML であることを定義する

MessageXML Adapter は、結果として XML 型のオブジェクトを格納した Map オブジェクトを返す仕様になっています。アダプタ側では、事前にどの型のデータを返すかを定義する必要があります。この項目ではその設定について説明致します。

アダプタで使用されるデータ入出力の型の定義は、OperationIO クラスに追加するクラスの型によって決まります。

実装は以下のようになります。

MessageXMLAdapterOperationFactory.java

```

15  static final String KEY_RESULT = "RESULT";

31  public OperationIO createOperationIO(
32      OperationConfiguration conf, OperationContext context) throws Exception {
33
34      OperationIO io = new OperationIO();
35      // 出力データが XML 型であることを定義します。
36      XMLOutput output = new XMLOutput(KEY_RESULT);
37      io.addOutput(output);
38
39      return io;
40  }

```

createOperationIO メソッドは、OperationFactory インターフェースで定義されているメソッドであり、OperationFactory インターフェースの派生クラスである MessageXMLAdapterOperationFactory はこのメソッドを実装する必要があります。XML 型の出力制約を定義するには、まず XMLOutput クラスのインスタンスを生成します。インスタンスを生成する際には、キーとなる文字列を指定します。そのインスタンスを OperationIO オブジェクトに add して、このメソッドの戻り値として返すことで、出力されるデータの型は、XML 型で定義されたことになります。

c. オペレーションを作成する

OperationFactory インターフェースの派生クラスである MessageXMLAdapterOperationFactory でもう一つ実装しなければならないメソッドが createOperation メソッドです。

このメソッドの戻り値は Operation インターフェースを実装した MessageXMLAdapterOperation クラスのインスタンスです。

MessageXMLAdapterOperationFactory.java

```

43  public Operation createOperation(
44      OperationConfiguration conf, OperationContext context) throws Exception {
45
46      return new MessageXMLAdapterOperation(conf, context);
47  }

```

7. MessageXMLAdapterOperationの実装

MessageXMLAdapterOperation クラスは、コンストラクタの引数に OperationConfiguration 型と OperationContext 型の2つが定義されています。コンストラクタに渡されたオブジェクトは、private なフィー

ルドに代入し、ほかのクラスからアクセスできないようにします。

import 文で以下のパッケージ名を指定する必要があります。(※)

- java.util.*;
- com.appresso.ds.common.xmlfw.xml.*;
- com.appresso.ds.xmlfw.*;

※ サンプルでは FQN で指定しています。

実装は以下のようになります。

MessageXMLAdapterOperation.java

```
13 private final OperationConfiguration conf;  
14 private final OperationContext context;  
15  
16 public MessageXMLAdapterOperation(  
17     OperationConfiguration conf, OperationContext context) throws Exception {  
18     this.conf = conf;  
19     this.context = context;  
20 }
```

MessageXMLAdapterOperation は Operation インターフェースを実装するため、以下の2つのメソッドを実装しなければなりません。

- execute メソッド
- destroy メソッド

この2つのうち、アダプタで行う処理を記述するのは execute メソッドになります。

以下に execute メソッドの実装例を示します。

MessageXMLAdapterOperation.java

```

23  public Map execute(Map inputData) throws Exception {
24      String message = conf.getValue(MessageXMLAdapterOperationFactory.KEY_MESSAGE).toString();
25
26      // XmlBuilder の生成
27      XmlBuilder builder = DataBuilderFactory.newMemoryXmlBuilder();
28
29      builder.startDocument();
30      builder.startElement("message");
31      builder.cdata(message);
32      builder.endElement("message");
33      builder.endDocument();
34
35      Map ret = new HashMap();
36      // 生成された XML を結果データとして Map に格納
37      // キーは出力型制約(XMLOutput)を定義したキーを指定します。
38      ret.put(MessageXMLAdapterOperationFactory.KEY_RESULT, builder.getResult());
39      return ret;
40  }

```

上記の例では、はじめにプロパティで設定されたメッセージを取得しています。

取得の方法は、OperationConfiguration オブジェクトの getValue メソッドの引数にプロパティのキーとなる文字列を渡すことで取得することができます。

MessageXMLAdapter では、MessageAdapterOperationFactory の、createOperationConfigurator メソッド内で、パラメータをaddする際に指定したキーがプロパティのキーになります。

次に、XmlBuilder インターフェースを実装しているクラスのインスタンスを生成しています。

XmlBuilderはDataSpider Servista で提供しているインターフェースで、処理結果の XML 文書を構築するメソッドが提供されています。

DataBuilderFactory の static メソッドである newMemoryXmlBuilder メソッドで、XmlBuilder インターフェースを実装しているクラスのインスタンスを取得し、そのインスタンスで XML 文書を構築します。


XML 文書の構築が終わったら、XmlBuilder インターフェースで定義されている getResult メソッドで、コンポーネント間で受け渡しする、構造化されたデータを表すオブジェクトを取得します。

構造化されたデータは、XML 文書として表現できるため、getResult メソッドで返されるオブジェクトを Map オブジェクトに格納し、その Map オブジェクトを execute メソッドの戻り値として返すことで、コンポーネント間のデータの受け渡しを実現します。

その際、キーとして XML 型の出力制約を定義したときのキーを指定します。このキーを指定することで、キーに関連付けられたデータを XML 文書で出力させることができます。

正しいキーを指定しないと、ほかのコンポーネントに対してデータを渡すことができません。

8. MessageXMLAdapterModuleComponent の実装

 **AdapterModuleComponent** を継承しているクラスは必ず、**<アダプタ名>ModuleComponent.java** というファイル名で作成してください。

import 文で以下のパッケージ名を指定する必要があります。(※)

- com.appresso.ds.dp.spi.*

※ サンプルでは FQN で指定しています。

MessageXMLAdapterModuleComponent の実装は、以下のようになります。

MessageXMLAdapterModuleComponent.java

```
1 package com.appresso.ds.dp.modules.adapter.messagexml;
2
3 import com.appresso.ds.dp.spi.AdapterModuleComponent;
4 import com.appresso.ds.dp.spi.GetDataOperationFactory;
5 import com.appresso.ds.dp.spi.OperationFactory;
6
7 public class MessageXMLAdapterModuleComponent extends AdapterModuleComponent {
8     @Override
9     public OperationFactory[] getOperationFactories() throws Exception {
10         return new GetDataOperationFactory[] { new MessageXMLAdapterOperationFactory() };
11     }
12 }
```

5.2.3. アダプタのビルド

1. アダプタのビルド

アダプタのビルドは、\$SDK_HOME/dev/message_xml_adapter ディレクトリで以下のコマンドを実行します。

- コマンド: ant

コマンド実行後、\$SDK_HOME/dev/message_xml_adapter/build ディレクトリ

に、message_xml_adapter.jar という jar ファイルが生成されていれば、ビルドは成功です。

以前のビルドで生成された jar ファイルを削除する場合は、ant コマンドの第一引数に distclean ターゲットを指定して実行します。

- コマンド: ant distclean

2. アイコンファイルのコピー

アダプタのビルドに成功すると、\$SDK_HOME/dev/message_xml_adapter/META-INF ディレクトリが生成されます。

そのディレクトリに\$SDK_HOME/dev/conf ディレクトリにある、以下のファイルをコピーしてください。

- operation.get_data.icon

3. module.properties ファイルの確認

\$SDK_HOME/dev/message_xml_adapter/META-INF/module.properties ファイルが正常に作成されたことを確認します。このプロパティファイルは、\$SDK_HOME/dev/message_xml_adapter/config.properties ファイルで定義されているプロパティをもとに作成されています。

それぞれのプロパティの内容は以下の通りです。

module.category	<p>アダプタがどのカテゴリに属するかを指定します</p> <ul style="list-style-type: none"> • file : ファイル系アダプタ • database : データベース系アダプタ • directoryservice : ディレクトリサービス系アダプタ • network : ネットワーク系アダプタ • application : アプリケーション系アダプタ • mapper : 変換系アダプタ • cloud : クラウド系アダプタ • bigdata : ビッグデータ系アダプタ <p> 任意のカテゴリ名も定義できます。(2 バイト文字は不可)</p>
component.class	<p>AdapterModuleComponent の派生クラスを完全修飾名 (※) で指定します ※パッケージ名を含むクラス名</p>
module.label	<p>アダプタのラベル名を指定します</p>
display.name	<p>アダプタの表示名を指定します</p>

5.2.4. アダプタのインストール

アダプタのインストールは \$SDK_HOME/dev/message_xml_adapter から ant コマンドの第一引数に install ターゲ

ットを指定して実行します。

- コマンド: `ant install`

`ant install` を実行後、`$DATASPIDER_HOME/server/plugin/data_processing/modules` ディレクトリ以下にアダプタがインストールされます。インストールしたアダプタを有効にするには、DataSpiderServer の再起動が必要です。

5.2.5. アダプタの動作確認

DataSpiderServer を再起動したあと、DataSpider Studio から、サーバへの再接続を行ってください。そのあとの手順は以下の通りです。

1. デザイナから任意のプロジェクトを開きます。
2. 新規スクリプトを作成します。
3. ツールパレットに、module.properties ファイルの module.category プロパティで指定したカテゴリ (SAMPLE) に入っていることを確認します。
4. カテゴリ内に、module.properties ファイルの module.label プロパティで指定したラベル名 (MessageXML) で存在していることを確認します。
5. 「読み取り」というオペレーションを、選択し、スクリプトキャンバス上にドラッグ & ドロップします。
6. プロパティを設定するダイアログが開きます。必須設定タブの「メッセージ」プロパティに任意のメッセージを入力します。名前を入力し「完了」を押下すると、アダプタが、スクリプトキャンバスに配置されます。
7. [ファイル] カテゴリのXMLから「XMLファイル書き込み処理」オペレーションをスクリプトキャンバスにドラッグ & ドロップします。
8. 入力データプロパティに[MessageXMLAdapter]で設定した名前プロパティを選択し、必須設定タブの「ファイル」プロパティに任意の書き込み先ファイルパスを入力します。
9. [start] - [MessageXMLAdapter] - [XML ファイル書き込み処理] - [end] の順にプロセスフローを引きます。
[XML ファイル書き込み処理] は入力データに [MessageXMLAdapter] を指定します。これで、
[MessageXMLAdapter] と [XML ファイル書き込み処理] の間にはデータフローが引かれます。
10. デザイナのメニューバーから、[テスト実行] - [デバッグ実行の開始 / 再開] または [実行] または「 F5 」キーでスクリプトを実行します。
11. [XML ファイル書き込み処理] で指定した、書き込み先ファイルに以下のような内容のデータが書き込まれていれば MessageXML Adapter の処理は成功です。

```
<?xml version="1.0" encoding="Shift_JIS"?>
<message><![CDATA[メッセージ]]></message>
```


5.3. Print Adapterの作成

5.3.1. 概要

基本情報

アダプタ名	Print Adapter
モジュール名	print_adapter
パッケージ名	com.appresso.ds.dp.modules.adapter.print
クラス名	PrintAdapterOperation PrintAdapterOperationFactory PrintAdapterModuleComponent

オペレーション情報

読み取りオペレーション	なし
書き込みオペレーション	入力元のXMLをファイルに出力
読み取りプロパティ	なし
書き込みプロパティ	出力先のファイルを指定

このアダプタを作成することで、以下の項目に関して学習することができます。

- 書き込みアダプタの作成方法
- ファイルを指定するプロパティの作成方法
- 入力データの扱い方
- ファイル操作の方法

なお、このアダプタのサンプルは以下のディレクトリにあります。

- \$SDK_HOME/samples/print_adapter

5.3.2. アダプタの作成

1. モジュールディレクトリの作成

\$SDK_HOME/dev 直下に print_adapter という名前のディレクトリを作成します。

- \$SDK_HOME/dev/print_adapter

2. build.xml、config.properties のコピー

\$SDK_HOME/dev/conf ディレクトリにある、build.xml ファイルと config.properties ファイルをモジュールディレクトリにコピーします。

3. config.properties ファイルの編集

以下のプロパティを設定します。

Implementation-Vendor	APPRESSO K.K.
module.label	Print
module.category	SAMPLE
display.name	Print Adapter

4. ソースディレクトリの作成

\$SDK_HOME/dev/print_adapter に src ディレクトリを作成します。

- \$SDK_HOME/dev/print_adapter/src

5. ソースファイルの作成

ソースディレクトリに以下のファイル名でソースファイルを作成します。

- PrintAdapterOperation.java
- PrintAdapterOperationFactory.java
- PrintAdapterModuleComponent.java

6. PrintAdapterOperationFactory の実装

PrintAdapterOperationFactory は、PutDataOperationFactory の派生クラスであり、アダプタのプロパティを保持するオブジェクトを返したり、オペレーションオブジェクトを返したりする Factory クラスです。

以下の実装を用意する必要があります。

- 書き込みプロパティ制約を作成する
- 入力データがXMLであることを定義する
- オペレーションを作成する

import文で以下のパッケージ名を指定する必要があります。(※)

- com.appresso.ds.dp.spi.*

- com.appresso.ds.common.spi.param.*
- com.appresso.ds.common.spi.constraint.*
- com.appresso.ds.dp.share.adapter.common.*
- com.appresso.ds.dp.share.adapter.file.*

※ サンプルでは FQN で指定しています。

a. 書き込みプロパティ制約を作成する

Print Adapter では、以下のプロパティを実装する必要があります。

- I. 書き込み先のファイルを指定するプロパティ
- II. 文字セットを選択するプロパティ
- III. インデントの有無を選択するプロパティ

 プロパティ制約に関する詳細は、「[12. プロパティ制約について](#)」ご参照ください。

I. 書き込み先のファイルを指定するプロパティ

- クラス: FileInputFillin
- パッケージ: com.appresso.ds.common.spi.constraint

実装は下記のようになります。

PrintAdapterFactory.java

```

20  static final String KEY_FILE = "FILE";

72  private SimpleParameter createFileInputParameter() {
73      // ファイルを指定するプロパティ制約を定義
74      FileInputFillin constraint = new FileInputFillin();
75      constraint.setLabel("ファイル");
76      constraint.setRequired(true);
77      constraint.setShortcut("F");
78      constraint.setDescription("書き込み先のファイルを指定します");
79
80      return new SimpleParameter(KEY_FILE, constraint);
81  }
```

II. 文字セットを選択するプロパティ

- クラス: CharsetNameFillinMulti
- パッケージ: com.appresso.ds.common.spi.constraint

実装は下記ようになります。

PrintAdapterFactory.java

```

22  static final String KEY_CHARSET = "CHARSET";

102 private SimpleParameter createCharsetParameter() {
103     // 文字セットを選択できるプロパティ制約を定義します。
104     CharsetNameFillinMulti constraint = new CharsetNameFillinMulti();
105     constraint.setLabel("文字セット");
106     constraint.setRequired(true);
107     constraint.setShortcut("C");
108     constraint.setDescription("書き込み先の文字セットを指定します");
109
110     return new SimpleParameter(KEY_CHARSET, constraint);
111 }
```

III. インデントの有無を選択するプロパティ

- クラス: Multi
- パッケージ: com.appresso.ds.common.spi.constraint

実装は下記ようになります。

PrintAdapterFactory.java

```

21  static final String KEY_INDENT = "INDENT";
```

```

83  private SimpleParameter createIndentParameter() {
84      // 選択形式のプロパティ制約を定義
85      Multi constraint = new Multi();
86      constraint.setLabel("インデント");
87      constraint.setRequired(true);
88      constraint.setShortcut("I");
89      constraint.setDescription("インデントを入れるかどうか指定します");
90      // 選択肢を定義
91      constraint.setItems(new Item[] {
92          new Item("true"),
93          new Item("false")
94      });
95
96      // GUI の表示形式をコンボボックスに設定
97      constraint.setStyle(Multi.STYLE_COMBOBOX);
98
99      return new SimpleParameter(KEY_INDENT, constraint);
100 }

```

上記 3 つのパラメータ (プロパティ) を返すメソッドを用意したあ

と、createOperationConfigurator メソッドの戻り値として返す OperationConfigurator オブジェクトに add します。そうすることで、プロパティ画面に、ファイル指定プロパティ、インデント指定プロパティ、文字セット指定プロパティというそれぞれの制約が設定された3つのプロパティを表示させることができます。

b. 入力データが XML 型であることを定義する

PrintAdapter は、入力データとして XML 型のオブジェクトを受け取る仕様になっています。アダプタ側では、事前にどの型のデータを受け取るかを定義する必要があります。この項目ではその設定について説明します。

アダプタで使用するデータ入出力の型の定義は、OperationIO クラスに追加するクラスの型によって決まります。実装は以下ようになります。

PrintAdapterOperationFactory.java

```

19  static final String KEY_XML_INPUT = "XML_INPUT";

```

```

39 public OperationIO createOperationIO(
40     OperationConfiguration conf,
41     OperationContext context) throws Exception {
42
43     OperationIO io = new OperationIO();
44     // 入力データが XML 型であることを定義
45     XMLInput input = new XMLInput(KEY_XML_INPUT);
46     io.addInput(input);
47
48     return io;
49 }

```

createOperationIO メソッドは、OperationFactory インターフェースで定義されているメソッドであり、OperationFactory インターフェースの派生クラスである PrintAdapterOperationFactory はこのメソッドを実装する必要があります。このメソッド内で、OperationIO インスタンスを生成します。次に、XMLInput クラスのインスタンスを生成しますが、その際に、入力データのキーとなる文字列を渡します。XMLInput インスタンスを OperationIO オブジェクトに add して、このメソッドの戻り値として返すことで、入力データの型は、XML 型で定義されたことになります。

c. オペレーションを作成する

OperationFactory インターフェースの派生クラスである PrintAdapterOperationFactory でもう一つ実装しなければならないメソッドが createOperation メソッドです。このメソッドの戻り値は Operation インターフェースを実装した PrintAdapterOperation クラスのインスタンスです。

PrintAdapterOperationFactory.java

```

52  public Operation createOperation(
53      OperationConfiguration conf,
54      OperationContext context) throws Exception {
55
56      // プロパティで設定された値を取得します。
57      String filePath = conf.getValue(KEY_FILE).toString();
58      String indent = conf.getValue(KEY_INDENT).toString();
59      String charset = conf.getValue(KEY_CHARSET).toString();
60
61      // FileConnection を生成
62      FileConnection con = FileAdapterUtil.getConnection(filePath, context, conf);
63      PrintAdapterOperation op = new PrintAdapterOperation(con, context);
64
65      // インデント、文字セットをオペレーションに設定
66      op.setIndent(indent);
67      op.setCharset(charset);
68
69      return op;
70  }

```

createOperation メソッドでは、はじめに、それぞれのプロパティのキーを指定して、プロパティに設定されている値を取得しています。その中で、書き込み先のファイルパスを文字列として、filePath 変数に格納します。変数 filePath は DataSpider Servista で提供されている、FileAdapterUtil の static メソッド getConnection の引数として渡すことで、FileConnection クラスのインスタンスを生成することができます。

FileConnection クラスは、トランザクション管理、ファイル IO の管理を行っているクラスで、そのインスタンスを FileAdapterOperation の派生クラスである、PrintAdapterOperation のコンストラクタに渡すことで、ユーザは、ファイルのトランザクション管理を意識することなく、ファイル IO の実装を簡略化することができます。

7. PrintAdapterOperation の実装

import 文で以下のパッケージ名を指定する必要があります。(※)

- java.util.*;
- java.io.*;
- com.appresso.ds.common.xmlfw.xml.*;
- com.appresso.ds.dp.share.adapter.common.*;

- `com.appresso.ds.dp.share.adapter.file.*;`
- `com.appresso.ds.dp.spi.*;`
- `com.appresso.ds.xmlfw.*;`

※ サンプルでは FQN で指定しています。

`PrintAdapterOperation`は`com.appresso.ds.dp.share.adapter.file` パッケージの `FileAdapterOperation` クラスを継承しています。

そのため、`PrintAdapterOperation` のコンストラクタ内で、スーパークラスのコンストラクタを呼び出す必要があります。

`PrintAdapterOperation.java`

```
20  public PrintAdapterOperation(  
21      FileConnection con, OperationContext context) {  
22  
23      super(con, context);  
24  }
```

次にオペレーションの実装です。

`PrintAdapterOperation.java`

```
17  private String indent;  
18  private String charset;
```



```

27 public Map execute(Map inputData) throws Exception {
28     try (OutputStream out = new BufferedOutputStream(getOutputStream())) {
29         // XML ハンドラを生成
30         XmlHandler printHandler = PrintingHandlerFactory.createXmlPrintHandler(out, charset, isIndent());
31
32         // XML パーサを生成
33         // 引数の Map オブジェクトから入力データを取得
34         // キーは入力制約 (XMLInput) を定義したキーを指定
35         XmlParser parser = DataParserFactory.newXmlParser(
36             inputData.get(PrintAdapterOperationFactory.KEY_XML_INPUT));
37
38         // パース開始 (ファイルにデータが出力されます)
39         parser.parse(printHandler);
40
41         // 結果データは空の Map オブジェクトを返します。
42         return new HashMap();
43     }
44 }

```


execute メソッド内では、指定したファイルの OutputStream を取得して、そのストリームに対して、入力データの XML 文書を書き込む処理を行っています。

まず、PrintHandlerFactory の static メソッド createXmlPrintHandler で XML 文書出力するためのハンドラである XmlHandler インスタンスを生成します。

次に入力データを取得します。引数として渡された、Map オブジェクトには、この場合、XML 文書が格納されています。そのデータを、入力制約を定義する際に指定したキーを指定して入力データを取得します。そして、取得した入力データをもとに XML 文書を解析するパーサを生成します。パースが開始されると、入力データである XML 文書を解析しながら、ハンドラにイベントが通知され、書き込み先のファイルに XML 文書が出力されていきます。

Print Adapter は結果データを返さないなので、戻り値は空の Map オブジェクトを返します。

8. PrintAdapterManagerComponent の実装

 **AdapterManagerComponent** を継承しているクラスは必ず<アダプタ名>ModuleComponent.java というファイル名で作成してください。

import 文で以下のパッケージ名を指定する必要があります。(※)

- com.appresso.ds.dp.spi.*

※ サンプルでは FQN で指定しています。

PrintAdapterManagerComponent の実装は、以下のようになります。

PrintAdapterManagerComponent.java

```

6 public class PrintAdapterManagerComponent extends AdapterModuleComponent {
7     @Override
8     public OperationFactory[] getOperationFactories() throws Exception {
9         return new OperationFactory[] { new PrintAdapterManagerOperationFactory() };
10    }
11 }

```

5.3.3. アダプタのビルド

1. アダプタのビルド

アダプタのビルドは、\$SDK_HOME/dev/print_adapter ディレクトリで以下のコマンドを実行します。

- コマンド: ant

コマンド実行後、\$SDK_HOME/dev/print_adapter/build ディレクトリに、print_adapter.jar という jar ファイルが生成されていれば、ビルドは成功です。

以前のビルドで生成された jar ファイルを削除する場合は、ant コマンドの第一引数に distclean ターゲットを指定して実行します。

- コマンド: ant distclean

2. アイコンファイルのコピー

アダプタのビルドに成功すると、\$SDK_HOME/dev/print_adapter/META-INF ディレクトリが生成されます。そのディレクトリに \$SDK_HOME/dev/conf ディレクトリにある、以下のファイルをコピーしてください。

- operation.put_data.icon

3. module.properties ファイルの確認

\$SDK_HOME/dev/print_adapter/META-INF/module.properties ファイルが正常に作成されたことを確認します。このプロパティファイルは、\$SDK_HOME/dev/print_adapter/config.properties ファイルで定義されているプロパティをもとに作成されています。それぞれのプロパティの内容は以下の通りです。

module.category	<p>アダプタがどのカテゴリに属するかを指定します</p> <ul style="list-style-type: none"> • file : ファイル系アダプタ • database : データベース系アダプタ • directoryservice : ディレクトリサービス系アダプタ • network : ネットワーク系アダプタ • application : アプリケーション系アダプタ • mapper : 変換系アダプタ • cloud : クラウド系アダプタ • bigdata : ビッグデータ系アダプタ <p> 任意のカテゴリ名も定義できます。(2 バイト文字は不可)</p>
component.class	<p>AdapterModuleComponentAdapterModuleComponent の派生クラスを完全修飾名 (※) で指定します ※パッケージ名を含むクラス名</p>
module.label	アダプタのラベル名を指定します
display.name	アダプタの表示名を指定します

5.3.4. アダプタのインストール

アダプタのインストールは \$SDK_HOME/dev/print_adapter から ant コマンドの第一引数に install ターゲットを指定して実行します。

- コマンド: ant install

ant install を実行後、\$DATASPIDER_HOME/server/plugin/data_processing/modules ディレクトリ以下にアダプタがインストールされます。インストールしたアダプタを有効にするには、DataSpiderServer の再起動が必要です。

5.3.5. アダプタの動作確認

DataSpiderServer を再起動したあと、DataSpider Studio から、サーバへの再接続を行ってください。そのあとの手順は以下の通りです。

1. デザイナから任意のプロジェクトを開きます。
2. 新規スクリプトを作成します。
3. ツールパレットに、module.properties ファイルの module.category プロパティで指定したカテゴリ (SAMPLE) に入っていることを確認します。
4. カテゴリ内に、module.properties ファイルの module.label プロパティで指定したラベル名 (Print) で存在していることを確認します。
5. まず、XML ファイルの読み取り処理を行うため、[ファイル] カテゴリの XML から、「XML ファイル読み取り」オペレーションをスクリプトキャンバスにドラッグ & ドロップします。
6. 「ファイル」プロパティに読み取り対象となる XML ファイルを DataSpider ファイルシステムのパスで指定します。
7. 次に、Print Adapter の「書き込み」オペレーションをスクリプトキャンバス上にドラッグ & ドロップします。
8. Print Adapter の必須設定タブには、「ファイル」、「文字セット」、「インデント」プロパティが存在しています。「ファイル」プロパティには、DataSpider ファイルシステムのパスで書き込み先のファイルを指定します。また、入力データは [XML ファイル読み取り処理] を指定します。プロパティおよび、入力データの設定が完了後、名前を入力し「完了」を押下して、アダプタをスクリプトキャンバスに配置します。
9. [start] - [XML ファイル読み取り処理] - [PrintAdapter] - [end] の順にプロセスフローを引きます。
10. デザイナのメニューバーから、[テスト実行] - [デバッグ実行の開始 / 再開] または [実行] または「F5」キーでスクリプトを実行します。
11. [PrintAdapter] で指定した、書き込み先ファイルに [XML ファイル読み取り処理] のデータが書き込まれていれば Print Adapter の処理は成功です。

6. トラブルシューティング

ant コマンドを実行すると、「Property:dataspider.home must be set Please make sure build.properties is correctly configured. 」というエラーが発生してビルドに失敗します。

\$SDK_HOME/dev ディレクトリにある build.properties ファイルに DATASPIDER_HOME プロパティが存在していないか、コメントアウトされています。

ant コマンドを実行すると、「\$DATASPIDER_HOME/server/system/common/lib not found. 」というエラーが発生してコンパイルに失敗します。

DataSpider Servista のインストールディレクトリが参照できません。DATASPIDER_HOME プロパティで指定したパスを確認してください。

ant コマンドを実行すると、「シンボルを見つけられません」というエラーが発生してビルドに失敗します。

ソースコードで参照しているクラスのパッケージがインポートされていない可能性があります。Import 文の設定を確認してください。

サーバを起動すると、アダプタの初期化に失敗し、ClassNotFoundException というエラーが発生して、アダプタのロードができません。

\$SDK_HOME/dev/XXX_adapter/META-INF ディレクトリにある、module.properties ファイルの component.class プロパティで指定した値が正しくありません。以下の手順で再度アダプタのインストールを行ったあと、DataSpiderServer を再起動してください。

- a. \$SDK_HOME/dev/XXX_adapter/config.properties ファイルの component.class プロパティの値を修正
- b. \$SDK_HOME/dev/XXX_adapter ディレクトリで ant コマンドを実行
- c. \$SDK_HOME/dev/XXX_adapter ディレクトリで ant uninstall コマンドを実行
- d. \$SDK_HOME/dev/XXX_world_adapter ディレクトリで ant install コマンドを実行

Hello World Adapter のスクリプトを実行すると、「 Hello World !**」と「** execute() !**」しかログが出力されません。**

ログレベルの設定がデフォルトの「INFO」になっています。デザイナのメニューバーの [テスト実行] - [実行オプション] から、ログレベルを [FINEST] に設定してください。

スクリプトを実行してもログが出力されません。

デザイナのメニューバーの [テスト実行] - [実行] で、スクリプトを実行している可能性があります。[テスト

実行] - [デバッグ実行の開始 / 再開] または、「 F5 」でスクリプトを実行してください。

MessageXML Adapter のメッセージを XML Adapter の書き込み処理で確認しようとしたところ、XML Adapter の書き込み処理で、`InputDataNotFoundException` という例外が発生してメッセージが確認できません。

以下のことを確認してください。

- MessageXMLAdapterOperationFactory クラスの createOperationIO メソッドで、XML 型の出力制約 (XMLOutput) を定義しているか
- MessageXMLAdapterOperation クラスの execute メソッドで返す Map オブジェクトに put する際のキーが、出力制約を定義したときのキーになっているか

Print Adapter の処理中に `InputDataNotFoundException` という例外が発生してアダプタの処理に失敗します。

PrintAdapterOperationのexecute メソッドで、引数の Map オブジェクトから入力データを取得する際に指定するキーが、XML 型の入力制約 (XMLInput) を定義したときのキーになっているか確認してください。

7. グローバルリソースについて

この章では、グローバルリソースについて説明します。

グローバルリソースとは、すべてのプロジェクトから利用できる共有リソースです。データベースやメールサーバ、FTP サーバなど外部システムへの接続はグローバルリソースとなります。グローバルリソースはシステムで非常に重要なリソースとなるため、管理者と作成者のみ変更を行うことができます。

7.1. グローバルリソースのクラス構成

グローバルリソースを実現するためには、`com.appresso.ds.dp.spi.ResourceFactory` インターフェースの実装クラス（リソースファクトリ）と `com.appresso.ds.dp.spi.Resource` インターフェースの実装クラス（リソース）を作成する必要があります。それぞれの役割は以下の通りです。

- リソースファクトリ

- リソースのプロパティの設定
- リソースの一意な名前の提供
- リソースのメタ情報の提供
- リソースの生成

- リソース

- グローバルリソースの情報を保持

7.2. グローバルリソースの実装方法

サンプルコード「Sample Resource Adapter」をもとに説明します。

なお、ここでの説明はアダプタでグローバルリソースを設定、参照するために必要な情報のみとします。

基本情報

アダプタ名	Sample Resource Adapter
モジュール名	sample_resource_adapter
パッケージ名	com.appresso.ds.dp.modules.adapter.resource

クラス名	SampleOperation SampleOperationFactory SampleResourceAdapterModuleComponent SampleResource SampleResourceFactory
------	--

オペレーション情報

読み取りオペレーション	グローバルリソースで設定したメッセージをログに出力
書き込みオペレーション	なし
読み取りプロパティ	グローバルリソースの設定
書き込みプロパティ	なし

リソース情報

プロパティ	スクリプト実行時にログに出力するメッセージを設定
-------	--------------------------

このアダプタを作成することで、以下の項目に関して学習することができます。

- グローバルリソースの作成方法
- オペレーションからグローバルリソースを利用する方法

なお、このアダプタのサンプルは以下のディレクトリにあります。

- `$SDK_HOME/samples/sample_resource_adapter`

7.3. グローバルリソースを利用するアダプタの作成

1. モジュールディレクトリの作成

`$SDK_HOME/dev`直下に`sample_resource_adapter` という名前のディレクトリを作成します。

- `$SDK_HOME/dev/sample_resource_adapter`

2. build.xml、config.properties のコピー

`$SDK_HOME/dev/conf` ディレクトリにある、`build.xml` ファイルと `config.properties` ファイルをモジュールディレクトリにコピーします。

3. config.properties ファイルの編集

以下のプロパティを設定します。

Implementation-Vendor	APPRESSO K.K.
module.label	Resource
module.category	SAMPLE
display.name	Sample Resource Adapter

4. ソースディレクトリの作成

\$SDK_HOME/dev/sample_resource_adapter に src ディレクトリを作成します。

- \$SDK_HOME/dev/sample_resource_adapter/src

5. ソースファイルの作成

ソースディレクトリに以下のファイル名でソースファイルを作成します。

- SampleOperation.java
- SampleOperationFactory.java
- SampleResourceAdapterModuleComponent.java
- SampleResource.java
- SampleResourceFactory.java

6. SampleOperationFactory の実装

SampleOperationFactory は、OperationFactory インターフェースを実装した具象クラスです。

SampleOperationFactory.java

```

33  public OperationConfigurator createOperationConfigurator(
34      OperationConfiguration conf,
35      OperationContext context) throws Exception {
36
37      // OperationConfigurator を生成
38      OperationConfigurator configurator = new OperationConfigurator(conf, context);
39
40      // ResourceConstraint を生成
41      ResourceConstraint constraint = new ResourceConstraint();
42      // ラベルを設定
43      constraint.setLabel("Sample Resource");
44      // SampleResource 固有の名前を設定
45      constraint.setResourceNameRegex(SampleResourceFactory.RESOURCE_NAME);
46      // リソースパラメータを生成
47      ResourceParameter param = new ResourceParameter(KEY_SAMPLE_RESOURCE, constraint);
48
49      // リソースパラメータを追加
50      configurator.addResourceParameter(param);
51      return configurator;
52  }

```

オペレーションファクトリでは、オペレーションのプロパティでグローバルリソースの設定を行うためのプロパティを実装します。上記コード 41 行目で、ResourceConstraint というプロパティ制約を定義しています。ResourceConstraint は、グローバルリソースのためのプロパティ制約です。45 行目では、ResourceConstraint クラスの setResourceNameRegex メソッドで SampleResource 固有のリソース名を設定します。47 行目で、プロパティで表示されているリソース名を取得するときに指定するキーと、プロパティ制約のインスタンスを渡して、パラメータインスタンスを生成し、50 行目で configurator インスタンスに追加しています。

7. SampleResourceFactory の実装

SampleResourceFactory クラスは、ResourceFactory インターフェースを実装した具象クラスです。

SampleResourceFactory クラスでは、ResourceFactory クラスで提供されている4つのメソッドを実装する必要があります。以下にそれぞれのメソッドについての説明を記述します。

◦ getResourceName メソッド

```

15  public static final String RESOURCE_NAME = "sample";

```

```

18 // SampleResource 固有の文字列 (キー) を返します。
19 public String getResourceName() {
20     return RESOURCE_NAME;
21 }

```

オペレーションが利用するリソースを識別するための名前を返します。ここで返す文字列は、ResourceConstraint インスタンスの setResourceNameRegex メソッドで設定する文字列と正規表現でマッチする文字列でなければなりません。Sample Resource Adapter では、ResourceConstraint の setResourceNameRegex メソッドで設定した文字列は、SampleResourceFactory の getResourceName メソッドで返す値を参照しているので、問題ありません。

◦ getResourceMetadata メソッド

```

24 public ResourceMetadata getResourceMetadata(
25     ResourceContext context) throws Exception {
26
27     // ResourceMetadata を生成
28     ResourceMetadata meta = new ResourceMetadata(this, context);
29     // リソースのアイコン名を設定
30     meta.setLabel("Sample Resource");
31     meta.setDefaultType(ResourceInfo.TYPE_GLOBAL_RESOURCE);
32     // プールのサポートを有効に設定
33     meta.setPoolSupported(true);
34     meta.setPoolDisabled(false);
35     return meta;
36 }

```

プールの設定、リソースアイコンのラベル名など、このクラスで生成するグローバルリソースのメタ情報を設定します。上記コードでは、30,33,34 行目でグローバルリソースのリソースアイコンで使用するラベル名の設定と、プールを有効にしています。

◦ createResourceConfigurator メソッド

```

39 public ResourceConfigurator createResourceConfigurator(
40     ResourceConfiguration conf,
41     ResourceContext context) throws Exception {
42
43     // ResourceConfigurator を生成
44     ResourceConfigurator configurator = new ResourceConfigurator(this, conf, context);
45
46     // ResourceConfigurator の初期化
47     configurator.initialize();
48
49     return configurator;
50 }

```

グローバルリソースとして設定するプロパティを定義します。プロパティの定義は、OperationFactory クラスの createResourceConfigurator メソッドで定義されています。

ラスの createOperationConfigurator

メソッドで実装する方法と同じなので、ここでの説明は省略させていただきます。

- **createResource メソッド**

```
55 public Resource createResource(  
56     ResourceConfiguration conf,  
57     ResourceContext context) throws Exception {  
58  
59     context.log().info("*** Create new resource ***");  
60     // リソースのインスタンスを返します。  
61     return new SampleResource(conf, context);  
62 }
```

グローバルリソースのインスタンスを返します。上記コードでは 61 行目で SampleResource クラスのインスタンスを返しています。

8. SampleResource の実装

SampleResource クラスは、Resource インターフェースを実装した具象クラスです。

SampleResource.java

```

8 public class SampleResource implements Resource {
9     private final String message;
10    private final LoggingContext log;
11
12    public SampleResource(
13        ResourceConfiguration conf,
14        ResourceContext context) {
15
16        this.message = conf.getValue(SampleResourceFactory.KEY_MESSAGE).toString();
17        this.log = context.log();
18    }
19
20    @Override
21    public void setup() throws Exception {
22        log.info("*** Resource SetUp ***");
23    }
24
25    @Override
26    public void cleanup() throws Exception {
27        log.info("*** Resource CleanUp ***");
28    }
29
30    @Override
31    public void destroy() throws Exception {
32        // server.log に出力されます。
33        log.info("*** Resource Destroy ***");
34    }
35
36    public String getMessage() {
37        return message;
38    }
39 }

```

SampleResource クラスでは、Resource クラスで提供されている 3 つのメソッドを実装する必要があります。

- **new(インスタンス化)**

- プロパティで設定されたリソース名をもとにリソースのインスタンスが生成されます。データベース系アダプタでは、データベースとのコネクションを生成します。

- **setup メソッド**

- リソースインスタンスが生成されたまたは、プールされている状態から利用される状態に遷移した直後に呼ばれます。データベース系アダプタでは、データベースのコネクションに対する接続状態を確認します。

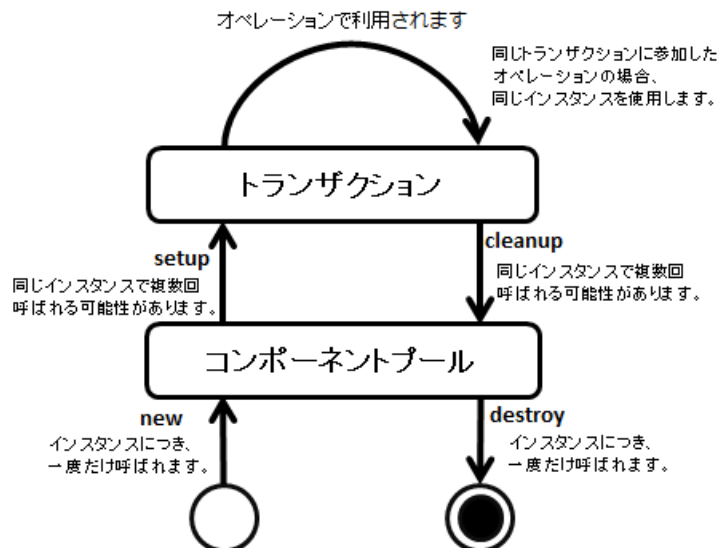
◦ cleanup メソッド

- トランザクションの終了時に呼ばれ、これからプールされることを通知します。この操作に失敗した場合、グローバルリソースのインスタンスは破棄されます。このメソッドは通常、プールされる前にグローバルリソースの状態が再利用可能かどうかをチェックします。また、プールされない場合は、インスタンスを破棄するコードを実装します。

◦ destroy メソッド

- グローバルリソースのインスタンスがこれ以上再利用されずに破棄されることを通知します。たとえば、プールの生存タイムアウトを超過した場合などで呼ばれます。データベース系アダプタでは、データベースとの接続を Close します。

◦ インスタンスと各メソッド呼び出しの関係



◦ destroy の呼ばれるタイミングについて

- プールを利用している場合

DataSpider Servista のプール GC に依存します。プール GC とは、プールされているグローバルリソースを定期的にチェックして、個々のグローバルリソースで設定されている生存時間を超えていれば、そのグローバルリソースを破棄する機構です。プール GC に破棄されるタイミングで destroy が呼ばれます。なお、プール GC の実行される間隔は180secです。(変更はできません)

- プールを利用していない場合

処理がトランザクションに参加している場合は、トランザクションの終了処理が完了したあとに呼ばれます。トランザクションに参加していない場合は、処理が完了したあとに呼ばれます。

9. SampleOperation の実装

SampleOperation クラスは Operation インターフェースを実装した具象クラスです。

SampleOperation.java

```

17 public class SampleOperation implements Operation, Transactional {
18     private final OperationConfiguration conf;
19     private final OperationContext context;

32     public Map execute(Map inputData) throws Exception {
33         // プロパティで設定されているリソース名を取得します。
34         String referenceName = conf.getValue(
35             SampleOperationFactory.KEY_SAMPLE_RESOURCE).toString();
36
37         // リソース名をもとにリソースインスタンスを取得します。
38         SampleResource resource = (SampleResource) context.getResource(referenceName);
39
40         // リソースが null だった場合、リソース名を引数として ResourceNotFoundException をスローします。
41         if (resource == null) {
42             throw new ResourceNotFoundException(referenceName);
43         }
44
45         // リソースで設定されている値を取得します。
46         String message = resource.getMessage();
47
48         log.info("### " + message + " ###");
49         return new HashMap();
50     }

```

SampleOperation クラスは、execute メソッドを実装します。Sample Resource Adapter ではこのメソッドでグローバルリソースを取得しています。上記コードで説明すると、34 行目のコードではプロパティで設定されているグローバルリソース名を取得しています。次にその名前からグローバルリソースのインスタンスを取得しています (38 行目)。resource オブジェクトが null の場合、プロパティで設定されたグローバルリソースが見つからないということになります。その場合、com.appresso.ds.common.dp.ResourceNotFoundException をスローしてください (42 行目)。グローバルリソースが取得できたら、そのグローバルリソースに設定されている値をログとして出力しています。

10. SampleResourceAdapterModuleComponent の実装

SampleResourceAdapterModuleComponent クラスは抽象クラスの AdapterModuleComponent クラスを継承します。

SampleResourceAdapterModuleComponent.java

```

1 package com.appresso.ds.dp.modules.adapter.resource;
2
3 import com.appresso.ds.dp.spi.AdapterModuleComponent;
4 import com.appresso.ds.dp.spi.OperationFactory;
5 import com.appresso.ds.dp.spi.ResourceFactory;
6
7 public class SampleResourceAdapterModuleComponent extends AdapterModuleComponent {
8
9     public SampleResourceAdapterModuleComponent() {
10    }
11
12    @Override
13    // リソースファクトリのインスタンスを配列で返します。
14    public ResourceFactory[] getResourceFactories() throws Exception {
15        return new ResourceFactory[] { new SampleResourceFactory() };
16    }
17
18    @Override
19    public OperationFactory[] getOperationFactories() throws Exception {
20        return new OperationFactory[] { new SampleOperationFactory() };
21    }
22 }

```

上記コードでは、getResourceFactoriesメソッドで、SampleResourceFactory のインスタンスを配列として返しています。こうすることで、Sample Resource Adapter はグローバルリソースを利用することができるようになります。

7.4. アダプタのビルド

1. アダプタのビルド

アダプタのビルドは、\$SDK_HOME/dev/sample_resource_adapter ディレクトリで以下のコマンドを実行します。

- コマンド: ant

コマンド実行後、\$SDK_HOME/dev/sample_resource_adapter/build ディレクトリ

に、sample_resource_adapter.jar という jar ファイルが生成されていれば、ビルドは成功です。

以前のビルドで生成された jar ファイルを削除する場合は、ant コマンドの第一引数に distclean ターゲットを指定して実行します。

- コマンド: ant distclean

2. アイコンファイルのコピー


アダプタのビルドに成功すると、\$SDK_HOME/dev/sample_resource_adapter/META-INF ディレクトリが生成されます。そのディレクトリに \$SDK_HOME/dev/conf ディレクトリにある、以下のファイルをコピーしてください。

- operation.get_data.icon
- resource.sample.icon

3. module.properties ファイルの確認

\$SDK_HOME/dev/sample_resource_adapter/META-INF/module.properties ファイルが正常に作成されたことを確認します。

このプロパティファイルは、\$SDK_HOME/dev/sample_resource_adapter/config.properties ファイルで定義されているプロパティをもとに作成されています。それぞれのプロパティの内容は以下の通りです。

module.category	<p>アダプタがどのカテゴリに属するかを指定します</p> <ul style="list-style-type: none"> • file : ファイル系アダプタ • database : データベース系アダプタ • directoryservice : ディレクトリサービス系アダプタ • network : ネットワーク系アダプタ • application : アプリケーション系アダプタ • mapper : 変換系アダプタ • cloud : クラウド系アダプタ • bigdata : ビッグデータ系アダプタ <p> 任意のカテゴリ名も定義できます。(2 バイト文字は不可)</p>
component.class	<p>AdapterModuleComponent の派生クラスを完全修飾名 (※) で指定します ※パッケージ名を含むクラス名</p>
module.label	<p>アダプタのラベル名を指定します</p>
display.name	<p>アダプタの表示名を指定します</p>

7.5. アダプタのインストール

アダプタのインストールは `$SDK_HOME/dev/sample_resource_adapter` から `ant` コマンドの第一引数に `install` ターゲットを指定して実行します。

- コマンド: `ant install`

`ant install` を実行後、`$DATASPIDER_HOME/server/plugin/data_processing/modules` ディレクトリ以下にアダプタがインストールされます。インストールしたアダプタを有効にするには、DataSpiderServer の再起動が必要です。

7.6. アダプタの動作確認

DataSpiderServer を再起動したあと、DataSpider Studio から、サーバへの再接続を行ってください。そのあとの手順は以下の通りです。

1. デザイナから任意のプロジェクトを開きます。
2. 新規スクリプトを作成します。
3. ツールパレットに、`module.properties` ファイルの `module.category` プロパティで指定したカテゴリ (SAMPLE) に入っていることを確認します。
4. カテゴリ内に、`module.properties` ファイルの `module.label` プロパティで指定したラベル名 (Resource) で存在していることを確認します。
5. Sample Resource Adapter の「Sample Resource」オペレーションをスクリプトキャンバス上にドラッグ & ドロップします。
6. Resourceプロパティで「追加」を押下して、リソース設定ダイアログを起動させます。リソース設定ダイアログの「Message」プロパティで任意の文字列を入力し、[完了]を押下します。Resource プロパティで、Sample Resource という名前のリソースが選択されていることを確認して、[完了]を押下してアダプタをスクリプトキャンバス上へ配置します。
7. [Start] – [Sample Resource] – [End] という順にプロセスフローを引き、スクリプトをデバッグ実行します。処理が正常に終了し以下のログが出力されていれば成功です。

```
### Create new resource ###
### resource set up ###
### < 任意のメッセージ > ###
#### resource clean up ###
```

8. 再度スクリプトを実行し、「### Create new resource ###」が出力されなければ、新たにグローバルリソースのインスタンスが生成されずにプールから利用されたことが確認できます。

8. XML フレームワークのデータ型について

この章では、コンポーネントで扱うデータ型について説明します。

8.1. XML フレームワークのデータ型

XML フレームワークのデータ型には、以下の 2 種類があります。

- テーブルモデル型
- XML 型

それぞれの特徴は以下の通りです。

- テーブルモデル型

列と行で表すことのできる表形式のデータを扱うデータモデルで、内部のデータ保持形式は java の二次元配列です。

- パフォーマンス: アダプタおよび Mapper の処理において高速な処理を実現
- 大容量処理: アダプタおよび Mapper の処理において対応

- XML 型

繰り返しポイントが複数あるような階層構造のデータを扱うデータモデルで、内部のデータ保持形式は独自の XML データ形式です。

- パフォーマンス: XML 解析が必要なため、テーブルモデル型のほうが高速
- 大容量処理: アダプタは対応しているが、Mapper が未対応

8.1.1. テーブルモデル型

テーブルモデル型とは、行と列の二次元配列からなるデータ型のことです。Servista のコンポーネントでは、データベース系アダプタや CSV アダプタなど、表形式のデータを扱うアダプタで採用しています。テーブルモデル型を採用する基準は以下の 2 点が挙げられます。

1. データを「行」「列」の表形式で表すことができる。
2. フォーマットが固定される

1 の理由として、テーブルモデル型のデータは java

の二次元配列で保持されます。そのため、リレーショナルデータベースのテーブルや、Excel のスプレッドシートなど表形式でデータが格納されているデータソースとのやりとりは、テーブルモデル型を採用することができます。

2 の理由としては、テーブルモデルのフォーマットは処理の前に定義するため、処理ごとに動的にフォーマットが変わる (列数が増える) データはテーブルモデル型で扱うことはできません。逆にフォーマットが固定されるデータであれば、表形式ではなくてもテーブルモデル型を採用することができます。

【例】 ServistaのMail Adapter のフォーマット

```
<?xml version="1.0" encoding="UTF-8"?>
<table>
  <row>
    <column>a.メッセージID</column>
    <column>b.送信日付(受信日付)</column>
    <column>c.差出人</column>
    <column>d.宛先</column>
    <column>e.CC</column>
    <column>f.BCC</column>
    <column>g.返信先</column>
    <column>h.件名</column>
    <column>i.Contetnt-Type</column>
    <column>j.本文</column>
    <column>k.添付ファイルパス</column>
  </row>
</table>
```

8.1.2. XML データ型

XML データ型は、XML データ情報をバイト列として保持します。パフォーマンスおよび、メモリ耐性に関してテーブルモデル型と比較すると、テーブルモデル型のほうがパフォーマンス、メモリ耐性ともに優れています。

9. 大容量データ処理について

DataSpider Servista では、大容量データ処理機構が用意されています。この章では大容量データ処理機構を利用するために必要な実装を説明します。

9.1. 大容量データ処理のプロパティ実装

XXXOperationFactory.java

```
1 import com.appresso.ds.xmlfw.XmlFrameworkConstraints;
2 public class XXXOperationFactory implements OperationFactory {
3     public OperationConfigurator createOperationConfigurator(
4         OperationConfiguration conf,
5         OperationContext context) {
6
7         OperationConfigurator configurator =
8             new OperationConfigurator(conf, context);
9         // これで大容量タブが表示されます。
10        XmlFrameworkConstraints.addTableDataParameter(configurator);
11    }
12 }
```

上記コードでは、1 行目に `import com.appresso.ds.xmlfw.XmlFrameworkConstraints` クラスをインポートしています。このクラスは DataSpider Servista の XML Framework で、大容量データ処理のプロパティを実装するために用意されているクラスです。実際には 10 行目でこのクラスの `addTableDataParameter` メソッドを利用することで大容量データ処理のプロパティを実装することができます。

9.2. プロパティで設定された値の取得

```
boolean largeData=XmlFrameworkConstraints.isLargeDataSelected(conf, context);
```

プロパティで設定された値の取得方法も、DataSpider Servista の XML Framework で用意されています。上記コードのように、`OperationContext`と`OperationConfiguration` のインスタンスの取得が可能な場合は、大容量データ処理で使用するプロパティの値を取得することが可能です。

9.3. 大容量データ処理に対応した読み取り処理

上記項目では、大容量データ処理のプロパティ設定および値の取得方法について説明しました。この項目では、その値を利用した読み取り処理を、テーブルモデル型と XML 型それぞれについて説明します。

9.3.1. テーブルモデル型の読み取り処理

XXXOperation

```

1 import com.appresso.ds.common.xmlfw.table.TableDataBuilder;
2 import com.appresso.ds.common.xmlfw.table.TableDataType;
3 import com.appresso.ds.xmlfw.DataParserFactory;
4 ...
5 private String[] columnNames;
6 private boolean largeData;
7 ...
8 public Map execute(Map inputData) throws Exception {
9     // プロパティで設定された型
10    int[] columnTypes = new int[columnNames.length];
11    Arrays.fill(columnTypes, TableDataType.TYPE_STRING);
12    // TableDataBuilder の生成
13    // largeDataは「11.2.プロパティで設定された値の取得」を参照
14    TableDataBuilder builder = DataBuilderFactory.newTableDataBuilder(columnTypes, largeData);
15
16    // データ作成開始のマーク
17    builder.startConstruction();
18    // 行の開始のマーク
19    builder.startRow();
20    // データの書き込み
21    builder.column(value);
22    // 行の終了のマーク
23    builder.endRow();
24    // データ作成終了のマーク
25    builder.endConstruction();
26    // 結果の取得
27    builder.getResult();
28    ...
29 }
```

上記コードは Operation インターフェースを実装した具象クラスの execute メソッド内部です。10 行目では、columnNames.length とありますが、これはスクリプト実行前にプロパティで設定された列の名前です。定義されたカラムの数と同数の int 型の配列を生成して、11 行目で各配列に TableDataType.TYPE_STRING という定数を代入しています。これは、定義されたすべてのカラムの型は「文字列」であるということを設定しています。TableDataType で定義されている型は文字列以外にもあります (基本的には java の型をサポート)。列の型を設定したあと、その情報と、大容量データ処理のプロパティから取得した boolean 変数 largeData を引数としてビルダーのインスタンスを生成します。(14 行目)

このビルダーを利用してテーブルモデル型のデータを構築し (17-25 行目)、構築されたデータのインスタンスを取得します。(27 行目) 取得されたインスタンスは結果データとしてスクリプトの後続の処理に渡すために、execute メ

ソッドの戻り値である Map オブジェクトに格納します。

9.3.2. XML 型の大容量データ処理対応読み取り処理

XXXOperation

```

1 import com.appresso.ds.common.xmlfw.xml.XmlBuilder;
2 import com.appresso.ds.xmlfw.DataBuilderFactory;
3 ...
4 private boolean largeData;
5 ...
6 public Map execute(Map inputData) throws Exception {
7     // XmlBuilder の生成
8     // largeData は「11.2. プロパティで設定された値の取得」を参照
9     XmlBuilder builder = DataBuilderFactory.newXmlBuilder(largeData);
10
11     // ドキュメントの開始
12     builder.startDocument();
13     // タグの開始
14     builder.startElement ("tag", null);
15     // CDATA の生成
16     builder.cdata("data");
17     // タグの終了
18     builder.endElement ("tag");
19     // ドキュメントの終了
20     builder.endDocument();
21     // 結果の取得
22     builder.getResult();
23     ...
24 }
```

上記コードは Operation インターフェースを実装した具象クラスの execute メソッド内部です。9 行目で、大容量データ処理のプロパティから取得した boolean 値 largeData を引数としてビルダーのインスタンスを生成します。生成されたビルダーを利用して XML データを構築し (11-20 行目)、構築されたデータのインスタンスを取得します。(22 行目) 取得されたインスタンスは結果データとしてスクリプトの後続の処理に渡すために、execute メソッドの戻り値である Map オブジェクトに格納します。

9.4. 大容量データ処理対応の書き込み処理

9.4.1. テーブルモデル型の大容量データ処理対応書き込み処理

XXXOperation

```

1 import com.appresso.ds.common.xmlfw.table.Column;
2 import com.appresso.ds.common.xmlfw.table.Row;
3 import com.appresso.ds.common.xmlfw.table.TableRowIterator;
4 import com.appresso.ds.xmlfw.DataParserFactory;
5 ...
6 public Map execute(Map inputData) throws Exception {
7     // 入力データを取得
8     Object data = inputData.get(ComponentIOKeys.DEFAULT_RESULT_KEY);
9     try (TableRowIterator iterator = DataParserFactory.newTableRowIterator(data)) {
10         // TableRowIterator の生成
11         // レコードの数分だけループ
12         while (iterator.hasNext()) {
13             // レコードを取得
14             Row record = iterator.next();
15             // カラムの数を取得し、その数分ループ
16             int columnCount = record.getColumnCount();
17             for (int i = 0; i < columnCount; i++) {
18                 // カラムを取得
19                 Column column = record.getColumn(i);
20                 // カラムのデータを文字列で取得
21                 String stringData = column.getStringValue();
22             }
23         }
24     }
25 }
26 ...

```

上記コードは Operation インターフェースを実装した具象クラスの execute メソッド内部です。まず、8 行目で入力データを取得しています。後続の処理では、結果データをテーブルモデル型として扱うために XML Framework で用意されている TableRowIterator を利用しています。TableRowIterator クラスは、テーブルモデル型に対応したイテレータです。このイテレータの next メソッドで行単位のループをし、そこで取得した Row インスタンスをさらにループさせてカラムを取得しています。21 行目で Column クラスの getStringValue メソッドを実行していますが、これは、結果データのカラムが文字列として定義されている場合に利用します。値の取得については各型について用意されています。この処理は必ず try - with - resources 文または try - finally 句を使用して TableRowIterator インスタンスをクローズする必要があります。

9.4.2. XML型の大容量データ処理対応書き込み処理

「[5.3. Print Adapterの作成](#)」をご参照ください。

10. トランザクションについて

この章では、コンポーネントにおけるトランザクションの実装について説明します。

10.1. コンポーネントでサポートするトランザクション

すべてのコンポーネントは、トランザクションをサポートすることができます。サポートすることが可能なトランザクションのタイプとしては、トランザクションをサポートしない場合を含めて、以下の3通りがあります。

- **XATransaction**: 2 フェーズコミットプロトコルをサポートするタイプのトランザクションです
- **LocalTransaction**: 通常の BEGIN - COMMIT 型のトランザクションをサポートするタイプのトランザクションです
- **NoTransaction**: トランザクションがサポートされない場合です

どのトランザクションをサポートするかは、下記に示すインターフェースを実装することで設定することができます。インターフェースを実装しない場合は、トランザクションをサポートしないことを意味します。通常は、グローバルリソースを利用しているアダプタは Resource インターフェースの実装クラスに、そうでない場合は Operation インターフェースの実装クラスで下記トランザクションインターフェースを実装します。なお、両方のインターフェースを実装した場合、処理がグローバルトランザクションに参加している場合は XATransaction、そうでない場合は LocalTransaction が適用されます。

- **com.appresso.ds.dp.spi.XATransactional**
 - 2 フェーズコミットプロトコルをサポートするタイプのトランザクションを実装することができます。このインターフェースを実装した場合、javax.transaction.xa.XAResource インターフェースの実装クラスのインスタンスを返す、initXAResource メソッドを実装する必要があります。通常は接続先のデータソースで提供されている javax.transaction.xa.XAResource インターフェースの実装クラスを利用します。独自で javax.transaction.xa.XAResource インターフェースを実装することはまれなケースです。
- **com.appresso.ds.dp.spi.TransactionResource**
 - 通常の BEGIN - COMMIT 型のトランザクションをサポートするタイプのトランザクションを実装することができます。このインターフェースを実装した場合、com.appresso.ds.dp.spi.TransactionResource インターフェースの実装クラスのインスタンスを返す、initTransactionResource メソッドを実装する必要があります。

11. 高度なプロパティ設定

11.1. テーブルプロパティ



列名
a
b
c

上へ(U)

下へ(D)

削除(R)

追加(A)

テーブル形式のプロパティの実装方法および、テーブルに入力された値の取得方法について説明します。

11.1.1. テーブルプロパティの実装

XXXOperationFactory.java

```

1 import com.appresso.ds.common.dp.constraint.TableConstraint;
2 import com.appresso.ds.common.spi.param.TableColumn;
3 import com.appresso.ds.common.spi.param.TableParameter;
4 ...
5 public OperationConfigurator createOperationConfigurator(
6     OperationConfiguration conf,
7     OperationContext context) throws Exception {
8
9     OperationConfigurator configurator = new OperationConfigurator(conf, context);
10
11     // テーブル用のプロパティ制約を生成
12     TableConstraint constraint = new TableConstraint();
13     constraint.setLabel("列一覧");
14
15     // テーブル用のパラメータを生成
16     TableParameter parameter = new TableParameter(KEY_TABLE, constraint);
17
18     // テーブルのカラムを生成
19     TableColumn column = new TableColumn(KEY_COLUMN, new Fillin());
20     column.getConstraint().setLabel("列名");
21
22     // カラムを追加
23     parameter.addColumn(column);
24
25     configurator.addTableParameter(parameter);
26     return configurator;
27 }
28 ...

```

上記コードは、OperationFactory インターフェースを実装した具象クラスの、createOperationConfigurator メソッドです。(ResourceFactory インターフェースを実装したクラスの場合は、createResourceConfigurator メソッドに実装します) 12 行目では、com.appresso.ds.common.spi.constraint.TableConstraint クラスのインスタンスを生成します。TableConstraint クラスは、テーブル形式のプロパティ固有のプロパティ制約です。生成されたインスタンスにテーブルのラベル名を設定します。

16 行目では、com.appresso.ds.common.spi.param.TableParameter クラスのインスタンスを生成しています。このときコンストラクタの第一引数に、このテーブルに関連付けするための文字列 (キー) を指定し、第二引数には、TableConstraint インスタンスを指定します。

次に、テーブルのカラムを定義します。カラムにもカラムごとにプロパティ制約を指定することができ、上記コードの 19 行目では、カラムを表す TableColumn インスタンスを生成する際に、第二引数に、new Fillin() というオブジェク

トを指定しています。これは、定義したカラムは単純なテキストが入力可能なフィールドを表しています。TableColumn に指定できるプロパティ制約には、Fillin のほかに、Multi、FillinMulti が指定可能です。

23 行目の addColumn メソッドでは、定義したカラムをテーブルに追加しています。そして最後に、configurator インスタンスに TableParameter インスタンスを追加すれば、テーブルプロパティの定義は完了です。

11.1.2. テーブルプロパティからの値の取得

```

1  ...
2  int rowCount = conf.getTable(KEY_TABLE).getRowCount();
3  String[] columns = new String[rowCount];
4  for (int i = 0, length = columns.length; i < length; i++) {
5      columns[i] = conf.getTable(KEY_TABLE).getValue(i, 0).toString();
6  }
7  ...

```

テーブルプロパティそのものを表すインスタンスの取得は、OperationConfiguration インスタンスからテーブルのキーを指定して取得します。上記コードでは、2 行目で、OperationConfiguration インスタンスである conf からテーブルを取得し、さらにそのテーブルで設定された行数を取得しています。

4-6 行目では、行数だけループさせて、5 行目でテーブルインスタンスから列の値を文字列として取得しています。このときテーブルインスタンスの getValue メソッドで指定している第一引数は「行番号」、第二引数は「列番号」を表しそれぞれ「0」を開始番号とします。(今回の例では、列は一つしかないので「0」で固定です。)

11.2. 接続テストプロパティ

[接続テスト...](#)

接続テストはリンクボタンで実装されています。この項目では、アクションボタンとそのボタンに関連付けられるアクションの実装について説明します。

11.2.1. アクションボタンの実装

XXXResourceFactory.java

```

1 import com.appresso.ds.common.dp.constraint.ActionConstraint;
2 import com.appresso.ds.dp.spi.ParameterActionHandler;
3 ...
4 public ResourceConfigurator createResourceConfigurator(
5     ResourceConfiguration conf, ResourceContext context) throws Exception {
6     ResourceConfigurator configurator = new ResourceConfigurator(conf, context);
7
8     // アクション用のプロパティ制約を生成
9     ActionConstraint constraint = new ActionConstraint();
10    constraint.setLabel("接続テスト(T)...");
11    constraint.setShortcut("T");
12
13    // アクション用のパラメータを生成
14    ActionParameter actionParam = new ActionParameter(KEY_CONNECT_TEST, constraint);
15
16    configurator.addActionParameter(actionParam);
17
18    // アクションを監視するリスナーをセット
19    configurator.setActionHandler(KEY_CONNECT_TEST, new ConnectTestActionHandler(conf));
20
21    return configurator;
22 }
23 ...

```

上記コードは、ResourceFactory インターフェースを実装した具象クラスの、createResourceConfigurator メソッドです。(OperationFactory インターフェースを実装したクラスの場合は、createOperationConfigurator メソッドに実装します)

9 行目で、アクションボタンのために提供されている com.appresso.ds.common.spi.constraint.ActionConstraint クラスのインスタンスを生成します。10 行目では、ボタンのラベルを設定しています。

次に、アクションボタンのフィールドを表す com.appresso.ds.common.spi.param.ActionParameter クラスのインスタンスを生成します。ActionParameter のコンストラクタの第一引数には、文字列 (キー) を指定します。このキーは、アクションが発火されたときに通知されるリスナーを関連付けするために利用され、上記コードでは、19 行目でキーとリスナーを関連付けしています。

11.2.2. リスナーの定義

前述では、アクションとリスナーをキーによって関連付けしていると説明しました。ここでは、そのリスナーにあたるクラスの作成について説明します。

```

1 import com.appresso.ds.common.dp.action.DialogResponse;
2 import com.appresso.ds.dp.spi.Configurator;
3 import com.appresso.ds.dp.spi.ParameterActionHandler;
4 import com.appresso.ds.dp.spi.ParameterActionResponse;
5 import com.appresso.ds.dp.spi.ParameterObject;
6
7 public class ConnectTestActionHandler implements ParameterActionHandler {
8     private final ResourceConfiguration conf;
9
10    public ConnectTestActionHandler(ResourceConfiguration conf) {
11        this.conf = conf;
12    }
13
14    public ParameterActionResponse actionPerformed(
15        Object arg, ParameterObject parameter, Configurator configurator) throws Exception {
16        // ParameterActionResponse を生成
17        ParameterActionResponse response = new ParameterActionResponse();
18        DialogResponse dialogResponse = null;
19        try {
20            // 接続テスト
21            ...
22            // ダイアログを表示するインスタンスを生成
23            dialogResponse = new DialogResponse(DialogResponse.TYPE_OK);
24            // タイトル、メッセージの設定
25            dialogResponse.setTitle("接続テスト");
26            dialogResponse.setMessage("***** 成功 *****");
27        } catch (Exception e) {
28            // 接続テスト失敗
29            // ダイアログを表示するインスタンスを生成
30            dialogResponse = new DialogResponse(DialogResponse.TYPE_ERROR);
31            // タイトル、メッセージの設定
32            dialogResponse.setTitle("接続テスト");
33            dialogResponse.setMessage(e.getMessage());
34            // 例外オブジェクトをセット
35            dialogResponse.setThrowable(e);
36        } finally {
37            // 接続のクローズ処理
38            ...
39        }
40        response.setActionResponse(dialogResponse);
41        return response;
42    }
43 }

```

上記コードは、com.appresso.ds.common.dp.param.ParameterActionHandler インターフェースを実装した具象クラスです。ParameterActionHandler インターフェースでは、ParameterActionResponse インスタンスを返す actionPerformed メソッドが提供されています。

actionPerformed メソッドでは、17 行目で ParameterActionResponse のインスタンスを生成しています。

19 行目からの try 句では、接続テストのコードが入ります。接続テストに成功した場合に

は、ParameterActionResponse インスタンスに、成功ダイアログを表示させるインスタンスを設定します。(23-26 行目)

何らかの理由で失敗した場合、例外をキャッチして、失敗した状態を表すダイアログを表示させるインスタンスを設定します。(30-35 行目)

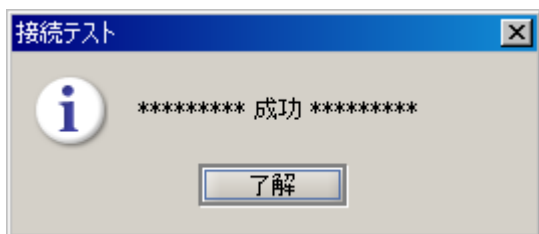
finally 句では通常、接続テストで使用したインスタンスをクローズする処理を行います。(38 行目)

最後に ParameterActionResponse のインスタンスに DialogResponse のインスタンスをセットして返します。(40、41 行目)

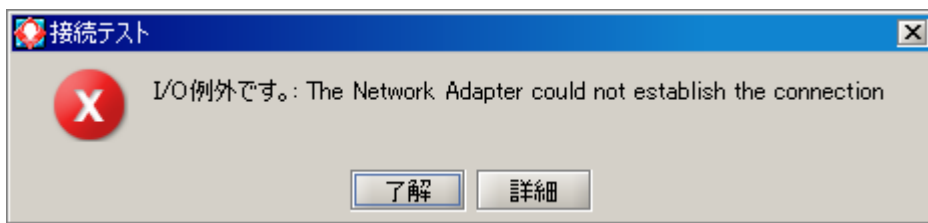
11.2.3. アクションの確認

プロパティに「接続テスト (T) …」というリンクボタンが表示されます。そのボタンを押下して成功時、失敗時、それぞれ以下のダイアログが表示されれば成功です。

[成功時]



[失敗時]



11.3. カスタムプロパティ制約によるプロパティ制約の拡張

DataSpider Servista のプロパティ制約は拡張可能な設計になっており、アダプタ固有のプロパティ制約の作成や、独

自に作成したプロパティに専用のエディタを用意することができます。DataSpider Servista では、独自に拡張したプロパティをカスタムプロパティと呼びます。

カスタムプロパティには二段階のレベルがあります。

一段階目はカスタムプロパティ制約を作成し、エディタは DataSpider Servista で標準提供されるものを使用する、という拡張です。この拡張は、エラーチェックやラベル表記の統一などを実現する際に行われるもので、通常の java の継承の機構を使ってプロパティ制約の拡張を実現しているため、比較的小さい手間で独自にプロパティ制約を開発することができます。

二段階目の拡張は、カスタムプロパティ制約に対応した専用のエディタを作成し、アダプタで使用されるプロパティ制約の編集を、より詳細に制御する、という拡張です。この拡張方法は、一段階目の拡張と比べると開発に必要な作業が多くなりますが、より柔軟にプロパティ制約を拡張することが可能となるため、アダプタ開発の際に、DB 系のアダプタで使用されている SQL ウィザードのような複雑な GUI を作成したい場合には有効です。

11.3.1. カスタムプロパティ制約の作成

DataSpider Servista 標準で提供されるプロパティ制約クラスの多くは、継承可能なクラスとなっています。このため、通常の java の継承の機構を用いてプロパティ制約を継承することで、カスタムプロパティ制約を作成することができます。

たとえば文字列を自由入力するためのプロパティ制約 Fillin を拡張し、“A-”から始まる文字列以外を入力できないようにするためのプロパティ制約 ProductCodeFillin を作成するには、次のようなプロパティ制約をアダプタのソースコード中に定義します。

```
1 package <アダプタのパッケージ>;
2 import com.appresso.ds.common.dp.constraint.Fillin;
3 import com.appresso.ds.common.dp.constraint.VerificationException;
4
5 public class ProductCodeFillin extends Fillin {
6     public void verify(String value) throws Exception {
7         super.verify(value);
8         if (value == null || value.length() == 0)
9             throw new VerificationException("製品コードが入力されていません。");
10        if (!value.startsWith("A-"))
11            throw new VerificationException("製品コードはA-から始まる必要があります。");
12    }
13 }
```

このようなカスタムプロパティ制約を作成することで、アダプタ中の複数箇所で使用されるプロパティ制約のロジック

をまとめて管理することが可能となります。

アダプタの内部だけでなく、複数のアダプタでカスタムプロパティ制約を再利用するには、DataSpiderServer インストールディレクトリ以下の、次のディレクトリにjarファイルを配置することで、全アダプタから利用可能なカスタムプロパティを定義することができます。

- \$DATASPIDER_HOME/server/plugin/data_processing/share/lib/

カスタムプロパティ制約に限らず、上記ディレクトリに配置されたjarファイルに含まれるクラスは、すべてのアダプタから参照可能になります。

11.3.2. カスタムプロパティ制約専用エディタの作成

アダプタごとに、より高度なカスタマイズを行いたい場合には、カスタムプロパティに対応した専用エディタを開発することも可能です。

カスタムプロパティ制約に対応した専用エディタ開発の手順は次のようになります。

1. カスタムプロパティ制約の作成
2. カスタムプロパティ制約に対応したエディタの作成
3. カスタムプロパティ制約とエディタの関連付け

ここでは、専用エディタの例として、色を設定するためのダイアログを表示するエディタを作成します。

1. カスタムプロパティ制約の作成

カスタムプロパティ制約 ColorConstraint を、次のように定義します。

```
1 package アダプタの共通モジュールのパッケージ;
2 public class ColorConstraint
3     extends com.appresso.ds.common.dp.constraint.ActionConstraint {
4 }
```

カスタムプロパティ制約専用エディタを開発する場合、カスタムプロパティ制約を定義するクラスは、サーバ上で稼動するアダプタと、クライアント上で稼動する専用エディタの双方から参照可能な場所に配置する必要があります。

カスタムプロパティ制約をアダプタとクライアントの双方から参照可能な箇所に配置するには、カスタムプロパティ制約を定義したクラスを含むjarファイルを、下記の場所に配置します。

- \$DATASPIDER_HOME/server/system/common/lib

このフォルダに配置されたjarファイルは、クライアント起動時にサーバから自動的にダウンロードされ、クライアントのモジュールからも参照可能になります。

2. カスタムプロパティ制約に対応したエディタの作成

次に、カスタムプロパティ制約に対応したエディタを生成するファクトリクラスを定義します。プロパティ制約のエディタを表すインターフェース `FieldConstraintEditorFactory` には、設定ダイアログで表示するエディタコンポーネントを定義するためのメソッド (`createEditor`) と、プロパティインスペクタで表示するエディタコンポーネントを定義するためのメソッド (`createCellEditor/createCellRenderer`) が定義されています。

```

1 package <アダプタのエディタ用のパッケージ>;
2 import java.awt.*;
3 import java.awt.event.*;
4 import javax.swing.*;
5 import javax.swing.table.*;
6 import com.appresso.ds.gui.core.*;
7 import com.appresso.ds.studio.pe.*;
8
9 public class ColorConstraintEditorFactory implements FieldConstraintEditorFactory {
10     public JComponent createEditor(Services services, PropertyValueHolder valueHolder) throws Exception {
11         return new ColorConstraintEditor(valueHolder);
12     }
13
14     public TableCellEditor createCellEditor(
15         Services services, PropertyValueHolder valueHolder) throws Exception {
16         return null;
17     }
18
19     public TableCellRenderer createCellRenderer(
20         Services services, PropertyValueHolder valueHolder) throws Exception {
21         return null;
22     }
23
24     private class ColorConstraintEditor extends ActionFieldEditor {
25         ColorConstraintEditor(PropertyValueHolder valueHolder) throws Exception {
26             super(valueHolder);
27         }
28
29         protected ActionListener createButtonActionListener()
30             throws Exception {
31             return new ActionListener() {
32                 public void actionPerformed(ActionEvent e) {
33                     try { // 色を選択するダイアログを表示
34                         Color selectedColor = JColorChooser.showDialog(ColorConstraintEditor.this, "色の選択", null);
35                         // 選択された色の情報をシリアルライズしてサーバに送信

```

```

36         getValueHolder().actionPerformed(selectedColor);
37     } catch (Exception ex) {
38         // エラーメッセージ表示などの例外処理
39     }
40 }
41 };
42 }
43
44 protected PropertyEventListener createPropertyEventListener()
45     throws Exception {
46     return null;
47 }
48 }
49 }

```

この例では、ダイアログで表示するためのエディタコンポーネントのみを定義し、プロパティインスペクタに関連するメソッドでは戻り値 `null` を返しています。`null` が返却された場合、プロパティインスペクタは該当するプロパティの編集領域を非表示にします。プロパティインスペクタに独自のエディタを追加する場合には、通常のSwingのJTable を扱うときと同じ要領でプロパティインスペクタに関連するメソッド (`createCellEditor/createCellRenderer`) を実装します。

36 行目の `valueHolder.actionPerformed(selectedColor)` では、引数に受け渡されたオブジェクトをシリアルライズし、サーバ側のアダプタモジュールに転送します。サーバに送られてきたオブジェクトをアダプタ内でハンドリングするには、`OperationConfigurator#setActionHandler` メソッドを用いて、該当する `ActionParameter` に対してハンドラを設定します。`ActionParameter` のハンドラを表す `ParameterActionHandler` の実装方法については、「[11.2. 接続テストプロパティ](#)」をご参照ください。

また、カスタムプロパティ制約に対応したエディタを初期化する際、アダプタ側から設定情報などのデータをエディタに渡したい場合には、プロパティ制約にシリアルライズ可能なフィールドを用意し、下記のようにエディタ起動前にプロパティ制約のインスタンスからフィールドの値を取得することで、エディタ起動の際に受け渡すことができます。

```

1  protected ActionListener createButtonActionListener()throws Exception {
2      return new ActionListener() {
3          public void actionPerformed(ActionEvent e) {
4              try {
5                  // プロパティ制約のインスタンスを取得
6                  ColorConstraint constraint = getConstraint();
7                  // プロパティ制約のインスタンスを取得
8                  Object editorParam = constraint.getXXX();
9                  ...
10             } catch (Exception ex) {
11                 ...
12             }
13         }
14     };
15 }

```

3. カスタムプロパティ制約とエディタの関連付け

ここまでの作業で、カスタムプロパティ制約の作成と、作成したカスタムプロパティ制約に対応したエディタの作成が完了したので、最後にカスタムプロパティ制約とエディタとを関連付ける設定を下記ファイルに追加します。

- \$DATASPIDER_HOME/client/system/conf/fieldconstraints.properties

変更前の ActionConstraint の設定

```

# ActionConstraint

actionconstraint.count=2
actionconstraint.0.type=com.appresso.ds.common.dp.constraint.internal.FileSelectionAction
actionconstraint.0.editor=com.appresso.ds.studio.pe.editor.FileSelectionActionEditorFactory
actionconstraint.1.type=com.appresso.ds.common.dp.constraint.ActionConstraint
actionconstraint.1.editor=com.appresso.ds.studio.pe.editor.ActionConstraintEditorFactory

```

変更後の ActionConstraint の設定（下線部が変更箇所）

```
# ActionConstraint

actionconstraint.count=3
actionconstraint.0.type=com.appresso.ds.common.dp.constraint.ColorConstraint
actionconstraint.0.editor=com.appresso.ds.studio.pe.editor.ColorConstraintEditorFactory
actionconstraint.1.type=com.appresso.ds.common.dp.constraint.internal.FileSelectionAction
actionconstraint.1.editor=com.appresso.ds.studio.pe.editor.FileSelectionActionEditorFactory
actionconstraint.2.type=com.appresso.ds.common.dp.constraint.ActionConstraint
actionconstraint.2.editor=com.appresso.ds.studio.pe.editor.ActionConstraintEditorFactory
```

プロパティ制約のエディタは、fieldconstraints.properties に定義されているインデックスの順に読み込まれるため、継承関係で上位にあるクラスより先に定義が読み込まれるようにします。

12. プロパティ制約について

この章では、DataSpider Servista で提供している種々のプロパティ制約について説明します。

プロパティ制約を表すクラスは `com.appresso.ds.common.spi.constraint` パッケージに入っています。

すべてのプロパティ制約を表すクラスは、`com.appresso.ds.common.spi.constraint.FieldConstraint` クラスの派生クラスです。

各プロパティ制約共通メソッド

メソッド名	機能
<code>setLabel(String)</code>	領域の表示名を設定します
<code>setShortcut(String)</code>	領域のショートカットを設定します
<code>setDescription(String)</code>	領域の説明を設定します
<code>setRequired(boolean)</code>	この領域に値を設定することが必須かどうかを設定します
<code>setEnabled(boolean)</code>	この領域が設定不可かどうかを設定します
<code>setVisible(boolean)</code>	この領域が表示されるかどうかを設定します
<code>setEditable(boolean)</code>	この領域が編集可能かどうかを設定します
<code>setSelective(Boolean)</code>	この領域が選択肢を持つかどうかを設定します
<code>setItems(Item[])</code>	この領域の静的な選択肢を設定します
<code>setInitialValue(String)</code>	この属性値の初期値を設定します

なお、各プロパティ制約に関連付けられている、GUI は、以下のサンプルアダプタをインストールすることで確認することができます。

- `$SDK_HOME/samples/property_sample_adapter`

 `property_sample_adapter` の実行はできません。

12.1. Fillin

自由入力型のプロパティ制約です。

スーパークラス: `com.appresso.ds.common.spi.constraint.PropertyConstraint`

固有メソッド

メソッド名	機能
setLength(int)	入力可能な最大値を設定します

12.2. FileInputFillin

「参照」ボタンを持つプロパティ制約です。

スーパークラス: com.appresso.ds.common.spi.constraint.Fillin

固有メソッド

メソッド名	機能
setCurrentDirectory(String)	カレントディレクトリを設定します
setFileExtensions(String[])	対象となるファイルの拡張子を設定します
setFileDescription(String)	対象となるファイルの説明を設定します
setFileSelectionMode(int)	対象となるファイルのモードを設定します <ul style="list-style-type: none"> • DIRECTORIES_ONLY: 0 ディレクトリのみを表示するモードです • FILES_AND_DIRECTORIES: 1 ファイルもディレクトリも両方表示するモードです
setLocalFile(boolean)	サーバローカルファイルシステムを使用するオプションです。

12.3. NumberFillin

数字自由入力型のプロパティ制約です。

スーパークラス: com.appresso.ds.common.spi.constraint.Fillin

固有メソッド

メソッド名	機能
setAllowMax(boolean)	最大値を許容するかどうかを設定します
setMaxValue(double)	最大値を設定します
setAllowMin(boolean)	最小値を許容するかどうかを設定します
setMinValue(double)	最小値を設定します
setAllowDouble(boolean)	浮動小数点を許容するかどうかを設定します

12.4. MultipleLineFillin

複数行自由入力型のプロパティ制約です。

スーパークラス: com.appresso.ds.common.spi.constraint.Fillin

固有メソッド

メソッド名	機能
setRows(int)	行数を設定します

12.5. PasswordFillin

文字列が HIDDEN_STRINGで 隠されるプロパティ制約です。

スーパークラス: com.appresso.ds.common.spi.constraint.Fillin

12.6. Multi

選択型のプロパティ制約です。

選択肢は、setItem(Item[]) で設定します。

スーパークラス: com.appresso.ds.common.spi.constraint.PropertyConstraint

固有メソッド

メソッド名	機能
setStyle(int)	<p>選択形式を設定します。</p> <ul style="list-style-type: none"> STYLE_COMBOBOX: 0 コンボボックス形式の選択欄を表します STYLE_RADIOBUTTON: 1 ラジオボタン形式の選択欄を表します STYLE_LIST: 2 リスト形式の選択欄を表します

12.7. FillinMulti

自由入力可能な選択型のプロパティ制約です。

スーパークラス: `com.appresso.ds.common.spi.constraint.Multi`

12.8. CharsetNameFillinMulti

自由入力可能な文字セット選択型のプロパティ制約です。

スーパークラス: `com.appresso.ds.common.spi.constraint.FillinMulti`

選択できる文字セットは DataSpider Servista がインストールされている OS により異なり、Windows の場合、デフォルトで Shift_JIS、UNIX の場合、デフォルトで EUC-JP がセットされます。

12.9. CheckBox

二値選択型のプロパティ制約です。

スーパークラス: `com.appresso.ds.common.spi.constraint.PropertyConstraint`

固有メソッド

メソッド名	機能
<code>setChecked(Boolean)</code>	初期状態でチェックされているかどうかを設定します

12.10. InformationConstraint

プロパティに情報を付加できるプロパティ制約です。

スーパークラス: `com.appresso.ds.common.dp.constraint.FieldConstraint`

固有メソッド

メソッド名	機能
<code>setTitle(String)</code>	情報のタイトルを設定します
<code>setMessage(String)</code>	情報のメッセージを設定します

InformationConstraint を使用する場合は、タイトルと、メッセージは、コンストラクタの引数として設定します。InformationConstraint 専用のパラメータとして、`com.appresso.ds.common.spi.param.InformationParameter` が追加されました。

【使用例】

```
1 InformationConstraint constraint = new InformationConstraint("タイトル", "メッセージ");  
2 InformationParameter parameter = new InformationParameter("KEY_INFO", constraint);
```

13. 入出力制約について

この章では、アダプタの実行時に扱う入力および出力データに対する制約について説明します。

デザイナーで定義されたスクリプト上で実行されるアダプタは、処理を行う際、データの受け取りや結果を出力する機能を実装しています。アダプタを作成する際にあらかじめ、入出力データに対して型の制約を定義します。DataSpider Servista では、現在以下の型を定義することができます。

- String(文字列) 型
- int(10 進数整数値) 型
- boolean(真偽値) 型
- XML 型

13.1. 入出力制約の定義方法

入出力制約は、OperationFactory インターフェースを実装した派生クラスで、createOperationIO メソッドを実装して定義します。

[例] 文字列型の出力制約を定義する場合

```

1  ...
2  private static final String KEY_STRING_OUTPUT = "STRING_OUTPUT";
3
4  public OperationIO createOperationIO(
5      OperationConfiguration conf, OperationContext context) throws Exception {
6      OperationIO io = new OperationIO();
7      StringOutput output = new StringOutput(KEY_STRING_OUTPUT);
8      io.addOutput(output);
9      return io;
10 }
11 ...

```

上記の例では、String(文字列) 型の出力制約を定義しています。この定義によって、アダプタは、スクリプト実行時の処理で、出力制約を定義したときの出力制約キー (上記例の場合、KEY_STRING_OUTPUT) をキーとして、execute メソッドの戻り値である Map オブジェクトに String 型のデータを格納することで、String 型のデータを出力することができます。この定義をしなかった場合、アダプタの処理内で String 型のデータを出力することはできません。

13.2. XML 型の入出力制約を定義する際のスキーマについて

入力または出力データがXML型で、あらかじめその XML 文書のスキーマ構造が固定されている場合には、入出力データに対して XMLSchema を定義することができます。設定方法は、createOperationIO メソッドで、XMLOutput および XMLInput クラスで定義されている setSchema(String) を利用することで設定することができます。

14. エラーコンポーネント変数の設定

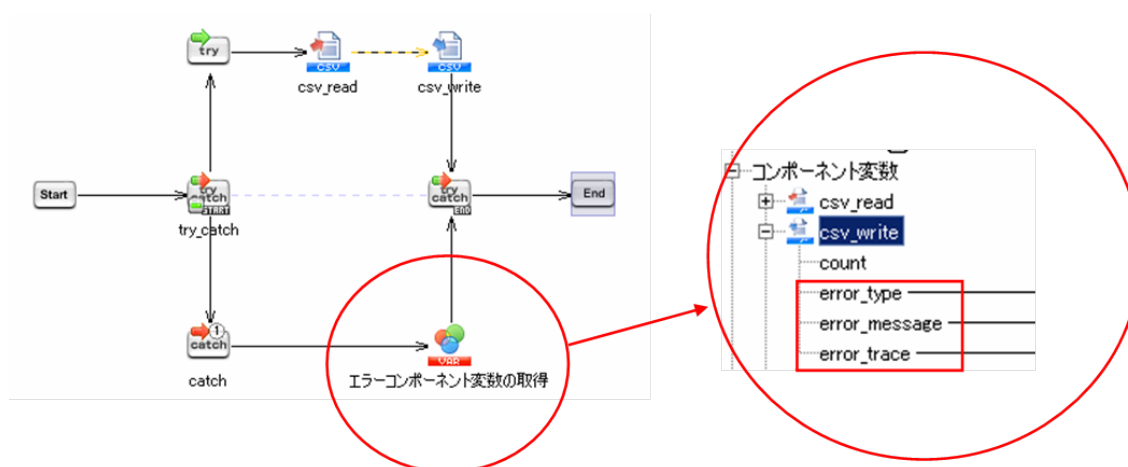
この章では、スクリプト上でエラーの型やエラーメッセージ、スタックトレースを取得するためのコンポーネント変数の設定方法について説明します。

14.1. エラーコンポーネント変数とは

エラーコンポーネント変数とは、コンポーネントの処理が失敗した場合、その処理のエラー情報が格納されたコンポーネント変数のことです。

エラーコンポーネント変数を設定すると、設定された処理には、以下のコンポーネント変数が定義されます。スクリプト実行時にそのコンポーネントの処理が失敗した場合、例外監視処理と組み合わせるとエラー情報をエラーコンポーネント変数から取得することができます。

- error_type
- error_message
- error_trace
- message_category (DataSpider Servista 3.0 以降)
- message_code (DataSpider Servista 3.0 以降)
- message_level (DataSpider Servista 3.0 以降)



エラーコンポーネント変数で取得できる情報は下記の通りです。

種別	コンポーネント変数名	変数の型	説明
エラータイプ	error_type	文字列	エラーが発生した場合、そのエラー (java.lang.Throwable 型のクラス) の FQN が代入されます。
エラーメッセージ	error_message	文字列	エラーが発生した場合、そのエラー (java.lang.Throwable 型のクラス) で実装されている getMessage メソッドで取得したメッセージが代入されます。
エラーのトレース情報	error_trace	文字列	エラーが発生した場合、そのエラー (java.lang.Throwable 型のクラス) のトレース情報が代入されます。
メッセージカテゴリ	message_category	文字列	エラー発生時、発生したエラーに対応するメッセージコードのカテゴリが格納されます。
メッセージコード	message_code	文字列	エラー発生時、発生したエラーに対応するメッセージコードのコードが格納されます。
メッセージレベル	message_level	文字列	エラー発生時、発生したエラーに対応するメッセージコードの重要度が格納されます。

14.2. エラーコンポーネント変数の実装

エラーコンポーネント変数の実装について説明します。エラーコンポーネント変数は、OperationFactory インターフェースを実装したクラスで実装されている createOperationIO メソッド内で定義します。

XXXOperationFactory.java

```

1  ...
2  public OperationIO createOperationIO(
3      OperationConfiguration conf, OperationContext context) throws Exception {
4      OperationIO io = new OperationIO();
5      // エラーコンポーネント変数を定義します。
6      IOConstraint.addErrorConstraint(io);
7      return io;
8  }
9  ...

```

上記 6 行目のコードを追加するだけで、この OperationFactory インターフェースの実装クラスが生成するオペレーションに、エラーコンポーネント変数を設定することができます。

14.3. 独自のエラーコンポーネント変数の実装

本項では、標準のエラーコンポーネント変数以外に独自にエラーコンポーネント変数を定義する方法について説明します。独自にエラーコンポーネント変数を定義する必要性として、データベースとのアクセスエラー時に発生する `java.sql.SQLException` のエラーコードをスクリプト上で取得したい場合などが考えられます。以下、`java.sql.SQLException` のエラーコードを、独自のエラーコンポーネント変数として実装する方法を説明します。

- エラーコンポーネント変数の定義

XXXOperationFactory

```

1  ...
2  public static final String OUTPUT_KEY_ERROR_CODE = "error_code";
3  ...
4  public OperationIO createOperationIO(
5      OperationConfiguration conf, OperationContext context) throws Exception {
6
7      OperationIO io = new OperationIO();
8      // 数値型の出力エラーコンポーネント変数「error_code」を定義します。
9      io.addOutput(new IntegerOutput(OUTPUT_KEY_ERROR_CODE));
10     return io;
11 }
12 ...

```

「error_code」という名前で、数値型出力コンポーネント変数を定義します。

- エラーコンポーネント変数の取得および代入

XXXOperation.java

```

1 import com.appresso.ds.common.dp.OperationException;
2 ...
3 public Map execute(Map inputData) throws Exception {
4     // エラーコードを格納する Map オブジェクトを生成します。
5     Map ret = new HashMap();
6     try {
7         // データベースとアクセスする処理
8         ...
9     } catch (SQLException e) {
10        // エラーコードを取得します。
11        int errorCode = e.getErrorCode();
12        // エラーコードを Map オブジェクトに格納します。
13        ret.put(XXXOperationFactory.OUTPUT_KEY_ERROR_CODE, errorCode);
14        // OperationException をスローします。
15        throw new OperationException(e, ret);
16    }
17 }
18 ...

```

Operation インターフェースの実装クラスで実装している execute メソッド内で、データベースにアクセスする処理を try-catch 句でくくり、try 句で SQLException が発生した場合、SQLException をキャッチしエラーコードを取得します (11 行目)。取得したエラーコードを Map オブジェクトに格納し (格納するときのキーはコンポーネント変数を定義したときに指定した名前)、SQLException とエラーコンポーネント変数を格納した Map オブジェクトを引数として、OperationException を生成しスローします (15 行目)。

スローされた OperationException は、DataSpiderServer でキャッチされます。DataSpiderServer ではキャッチした OperationException が保持している Map オブジェクトに格納されているエラーコンポーネント変数の値を、スクリーン上で扱えるように処理します。

15. メッセージコードについて

DataSpider Servista3.0 から、スクリプト実行時に発生したエラーに任意のコードを割り当て、エラー発生時には、そのコードをもとにエラーの内容・原因・対応策などを確認することができる「メッセージコード」機能が追加されました。メッセージコードに関する詳細は、DataSpider Servista のヘルプを参照してください。

ここでは、SDK で開発したアダプタでメッセージコードに対応するための手順について説明します。

1. カテゴリの決定

アダプタごとに一意な文字列となるカテゴリを決めます (使用できる文字列は半角大文字の英字のみです)。

[例] Excel アダプタの場合

- EXCEL

2. プロパティファイルの定義

アダプタの META-INF ディレクトリ以下に、messagecode.properties という名前のファイルを作成します。このファイルでは、メッセージのカテゴリ、および例外とコードの対応付けや重み付け (重要度) を定義します。

[記述方法]

```
category= < カテゴリ名 ( 半角大文字の英字 )>
```

```
<キー>.class=< 対象の Exception の FQN>
```

```
<キー>.code=< コード ( 数値 4 桁 )>
```

```
<キー>.level=< 重要度 ( 現在は「 E 」のみ )>
```

💡 < キー > には messagecode.properties ファイル内で一意な文字列 (半角英数字) を指定します。

⚠️ コードで指定できる数値は「 0001 」からになります。(「 0000 」は指定できません)。

💡 複数の例外 (またはエラー) を、1つのコードに割り当てることも可能です。

[例] Excel アダプタの場合

```
category=EXCEL
```

```
0.class=jxl.read.biff.PasswordException
```

```
0.code=0001
```

```
0.level=E
```

```
1.class=jxl.read.biff.BiffException
```

```
1.code=0002
```

```
1.level=E
```

```
2.class=com.apresso.ds.dp.modules.adapter.excel.IllegalFormatException
```

```
2.code=0003
```

```
2.level=E
```

16. エラーチェックについて

16.1. プロパティの設定が不正な場合

必須となるプロパティが設定されていないなど、プロパティの設定が不正な値の場合には、`InvalidPropertyConfigurationException` をスローします。

```
1 import com.appresso.ds.common.dp
2
3 public class XXXFactory implements OperationFactory {
4     ...
5     public Operation createOperation(OperationConfiguration conf, OperationContext context) throws Exception {
6         String name = conf.getValue("NAME").toString();
7         if (name == null || name.equals("")) {
8             throw new InvalidPropertyConfigurationException("NAMEが指定されていません。");
9         }
10        ...
11        return new XXXOperation(conf, context);
12    } ...
13 }
```

17. 外部ライブラリの利用について

DataSpider Servista のライブラリ以外を参照している場合は、そのライブラリを下記のディレクトリに配置してビルドを実行してください。

- `$SDK.HOME/dev/XXX_adapter`

上記ディレクトリに置かれたライブラリは、`ant install` コマンドを実行する

と、`$DATASPIDER_HOME/server/plugin/data_processing/modules/XXX_adapter` ディレクトリにコピーされます。

18. オペレーションとグローバルリソースのアイコンファイルについて

アダプタのオペレーションおよびグローバルリソースには、一意を表すアイコンファイルが必要です。オペレーションのアイコンファイルの場合、デザイナのツールパレットやスクリプトキャンバスなどに表示され、グローバルリソースのアイコンファイルの場合、コントロールパネルのグローバルリソースの設定などに表示されます。アイコンファイルの仕様は以下の通りです。

- アイコンファイルの画像形式
 - PNG
- アイコンファイルのサイズ
 - 32 × 32 ピクセル
- オペレーションのアイコンファイルの命名規則
 - operation.<OperationFactoryのgetOperationName メソッドで返す文字列 >.icon

[例] getOperationName メソッドで返す文字列が "execute_query" の場合

アイコン名: operation.execute_query.icon

- グローバルリソースのアイコンファイルの命名規則
 - resource.<ResourceFactoryのgetResourceName メソッドで返す文字列 >.icon

[例] getResourceName メソッドで返す文字列が "connection" の場合

アイコン名: resource.connection.icon

💡 \$SDK_HOME/dev/XXX_adapter/META-INF ディレクトリにアイコンファイルを配置することで、ant install コマンド実行時にインストール先にコピーされます。

19. パスワード取得時の注意点

DataSpider Servista 2.3.0 から、PasswordFillin を制約にしたパラメータに保存される値は、暗号化して保存されます。

保存されている文字列を取得する場合は通常、Value オブジェクトの toString メソッドで取得しますが、この方法だと復号せずに文字列を取得します。

したがって、PasswordFillin の場合は暗号が取得されます。復号した平文を取得するには、Value オブジェクトの decryptPassword メソッドを使用してください。

20. ログ出力指針

出力するログの内容とその内容を出力するログレベルの指針は以下の一覧を参照してください。

ログレベル名	メソッド名	説明	出力色
ERROR	error()	運用でのエラー発生時、内部状態を出力する場合に使用してください。 【例】 コネクションの生成に失敗した場合に、その内容をエラーログレベルで出力します。	濃い赤 (255.0.0)
WARN	warn()	運用時に警告として出力する場合に使用してください。	赤 (128.0.0)
INFO	info()	運用時の推奨ログレベルです。処理件数があるものは、件数を出力します。それ以外は出力しません。 【例】 CSVアダプタ [1] 行のデータを抽出しました。	濃い黒 (0.0.0)
FINEWARN	fwarn()	開発時に警告として出力する場合に使用してください。	薄茶色 (128.96.0)
FINEINFO	finfo()	開発時の推奨ログレベルです。アダプタの処理で大きなステップの開始や終了をログ出力します。 また、大きなステップで必要とされるプロパティ値も出力できます。 【例】 CSV アダプタ CSV ファイル [/data/in.csv] からデータの抽出を開始します。	黒 (64.64.64)
FINEST	finest()	処理の詳細な情報を出力する場合に使用してください。また、プロパティのダンプも FINEST で出力します。 【例】 FTP アダプタ FTP サーバとのやりとり。	グレー (128.128.128)
DEBUG	debug()	内部状態のデータを出力する場合に使用してください。	グレー (168.168.168)

21. 推奨されていない API について

推奨されていない API の詳細については、Javadoc の非推奨 API リストを参照してください。

- `$SDK_HOME/doc/api/deprecated-list.html`



推奨されていない API は予告なく削除される場合があります。

22. 言語切り替え対応について

DataSpider Servista 4.0 から、表示言語を選択できるようになりました。

本項では、選択された表示言語によってログメッセージや表示ラベルを変更するための実装例を説明します。

22.1. アダプタの言語切り替え対応

- 選択された表示言語の取得

次のクラス・メソッドで表示言語の `java.util.Locale` を取得できます。

```
com.appresso.ds.common.locale.ThreadLocaleManager#get()
```

- 実装例

表示言語によってラベルなどを変更する場合、`ResourceBundle` の利用を推奨します。

次の例では、各パッケージに配置された `messages.properties` ファイルから表示言語のリソースを取得します。

- `ResourceBundle` を使用した言語切り替え対応例

`com.appresso.ds.dp.modules.adapter.simple_csv` パッケージに `messages.properties` を配置している場合

```
ResourceBundle.getBundle("com.appresso.ds.dp.modules.adapter.simple_csv.messages", ThreadLocaleManager.get());
```

上記の例では、表示言語が日本語であれば `messages_ja.properties` を、英語であれば `messages_en.properties` を取得します。

22.2. ヘルプドキュメントの言語切り替え対応

選択された表示言語のヘルプドキュメントを表示するには「[31.5. 英語環境用ヘルプドキュメントを作成する](#)」を参照してください。

英語環境用ヘルプドキュメントを作成しない場合、表示言語を英語にすると作成したアダプタのヘルプは表示されません。

23. 2.3.x から 2.4.0 の変更点

DataSpider Servista 2.3.x から 2.4.0 のバージョンアップに伴う変更はありません。

24. 2.4.x から 3.0 の変更点

DataSpider Servista 2.4.x から 3.0 のバージョンアップに伴う変更は以下の通りです。

24.1. パッケージの変更

DataSpider Servista 3.0 では、プロパティ制約およびパラメータ関連のクラスのパッケージが変更されました。以下にパッケージが変更されたクラスを記述します。作成したアダプタで以下のクラスを変更前パッケージ名で import している場合、変更後のパッケージ名で import 文を修正する必要があります。

変更前パッケージ名	変更後パッケージ名	クラス
com.appresso.ds.common.dp.constraint	com.appresso.ds.common.spi.constraint	すべてのクラス
com.appresso.ds.dp.spi	com.appresso.ds.common.spi.param	ActionParameter ArrayItemLoader Configurator Configuration Configuration.NodeIterator Configuration.Table Configuration.Tree Configuration.Value InformationParameter ResourceParameter ResourceValueHandler ParameterActionHandler ParameterActionResponse ParameterObject ParameterObserver SimpleParameter TableColumn TableColumnValueLoader TableParameter TableValueInitialLoader TreeNode TreeParameter TreeValueInitialLoader Verifier VerifierChain

25. 3.0 から 3.1 の変更点

DataSpider Servista 3.0 から 3.1 のバージョンアップに伴う変更はありません。

26. 3.1 から 3.2 の変更点

DataSpider Servista 3.1 から 3.2 のバージョンアップに伴う変更はありません。

27. 3.2 から 4.0 の変更点

DataSpider Servista 3.2 から 4.0 のバージョンアップに伴う変更は以下の通りです。

27.1. 言語切り替え機能の追加

DataSpider Servista 4.0 では、ログイン時に表示言語を選択できるようになりました。

オペレーションのラベルやログなどを表示言語に合わせて変更する方法については、「[22. 言語切り替え対応について](#)」を参照してください。

27.2. 非推奨メソッドの追加

以下のメソッドが非推奨に追加されました。

- `com.appresso.ds.common.xmlfw.table.TableRowIterator#reset()`

本メソッドは DataSpider Servista 4.1 で削除されました。

28. 4.0 から 4.0SP1 の変更点

DataSpider Servista 4.0 から 4.0SP1 のバージョンアップに伴う変更は以下の通りです。

28.1. LoggingContext のメソッド追加

DataSpider Servista 4.0 SP1 では、LoggingContext のログ出力メソッドに `Supplier<String>` を引数とするメソッドが追加されました。下記にこれまでのコード例と `Supplier` を利用したコード例を記載します。どちらも結果は等価となります。

これまでのログ出力

```
1 if (context.log().isDebugEnabled()) {  
2     context.log().debug("デバッグレベルのログ");  
3 }
```

`Supplier` によるログ出力

```
1 context.log().debug(() -> "デバッグレベルのログ");
```


29. 4.0SP1 から 4.1 の変更点

DataSpider Servista 4.0SP1 から 4.1 のバージョンアップに伴う変更はありません。

30. 4.1 から 4.2 の変更点

DataSpider Servista 4.1から 4.2 のバージョンアップに伴う変更はありません。

31. ヘルプドキュメントの作成

DataSpider Servista のヘルプの機構にヘルプドキュメントを追加する手順は大きく分けて 3 つあります。

- アダプタのヘルプドキュメント (*.html) を追加する
- 追加したヘルプドキュメントを登録する
- ヘルプインデックスを再構築する

ヘルプドキュメントのサンプルは以下のディレクトリにあります。

- `$SDK_HOME/samples/simple_csv_adapter/help`

本項では、上記サンプルを参考にしながら説明します。

31.1. アダプタのヘルプドキュメントの記述ルール

DataSpider Servista のヘルプは以下の HTML タグの記述ルールにしたがっています。

- ページタイトル: `<h1></h1>`
- 大題目: `<h2></h2>`
- 中題目: `<h3></h3>`
- 小題目: `<h4></h4>`
- リソース名: `[]`



リソース名とは、DataSpider Servista 上で使われる設定名や表示名です。

31.2. アダプタのヘルプドキュメントを追加する

ヘルプドキュメントは `$DATASPIDER_HOME/server/doc/help/ja` にあります。この中で、`adapter` というディレクトリにすべてのアダプタのヘルプが置いてあります。

`adapter` ディレクトリの中は、

(`application`、`basic`、`bigdata`、`cloud`、`convert`、`directoryservice`、`database`、`file`、`network`) の 9 つのディレクトリに分かれています。これは、スクリプトデザイナーのツールパレットのカテゴリと同じ構成になります。

たとえば、file というディレクトリには、CSV アダプタや Excel アダプタなどのファイル系アダプタのヘルプドキュメントが HTML 形式で置いてあります。つまり、SDK で作成したアダプタがすでに存在するカテゴリのいずれかに属するのであれば、HTML ファイルは該当するディレクトリに配置することになります。カテゴリがない場合は、カテゴリを表すディレクトリと同階層に新規ディレクトリを作成し、その中に HTML ファイルを配置します。

ヘルプドキュメントの HTML ファイルの命名規則は特にありませんが、DataSpider Servista ではファイル名はすべて小文字で、「< アダプタ名 >_< オペレーション名 >.html」になっています。

例 :Simple CSV アダプタ読み取り処理の場合

- simplecsv_get_data.html

31.3. 追加したヘルプドキュメントを登録する

ヘルプドキュメントを適切な場所に置いたら、DataSpider Servista のヘルプ機構に、追加したヘルプドキュメントの情報を記述します。登録後はアダプタのオペレーションを選択した状態で F1 キーを押下すると、その処理に関連付けられたヘルプが表示されるようになります。

ヘルプドキュメントの情報が記述されているファイルは下記の 3 つです。

ヘルプセットファイル

- \$DATASPIDER_HOME/server/doc/help/< アダプタ名 >_jp.hs

マップファイル

- \$DATASPIDER_HOME/server/doc/help/ja/< アダプタ名 >Map.jhm

TOC ファイル

- \$DATASPIDER_HOME/server/doc/help/ja/< アダプタ名 >TOC.xml

31.3.1. ヘルプセットファイルの編集

ヘルプセットファイルは、後述のマップファイルおよび TOC ファイルをヘルプに関連付けします。

たとえば Simple CSV アダプタでは help/SimpleCSV_AdapterHelp_ja.hs が該当し、次のように設定されています。

```
<mapref location="ja/SimpleCSV_AdapterHelpMap.jhm"/>
```

```
// 中略
```

```
<data>ja/SimpleCSV_AdapterHelpTOC.xml</data>
```

mapref 要素の location 属性には、ヘルプセットファイルからマップファイルへの相対パスを設定してください。
data 要素には、ヘルプセットファイルから TOC ファイルへの相対パスを設定してください。

31.3.2. マップファイルの編集

マップファイルはヘルプドキュメントのパスと、キー文字列を関連付けします。

たとえば Simple CSV アダプタでは help/ja/SimpleCSV_AdapterHelpMap.jhm が該当し、次のように設定されています。


```
<mapID target="adapter.simple_csv_adapter.get_data" url="adapter/file/csv_get_data.html" />
```


```
<mapID target="adapter.simpe_csv_adapter.put_data" url="adapter/file/csv_put_data.html" />
```


mapID で定義する属性についてご説明します。

- **target 属性**

- adapter.< モジュール名 >.< オペレーション名 >

 < モジュール名 > とは、\$DATASPIDER_HOME/server/plugin/data_processing/modules ディレクトリ直下にあるアダプタのディレクトリ名です。

 < オペレーション名 >とは、OperationFactory インターフェースの実装クラスで実装している getOperationName メソッドで返す文字列です。

 target 属性値は、以下の方法で適切な値を参照することができます。

1. \$DATASPIDER_HOME/client/conf/dslog.properties にある dslog の項目を \${DEBUG} に変更します。
2. Studio を起動し、スクリプトデザイナのツールパレットから、スクリプトキャンバスにアダプタのオペレーションアイコンをドラッグ & ドロップして、F1 を押します。
3. \$DATASPIDER_HOME/client/logs/studio.log に下記のようなデバッグ情報が出力されます。(以下は CSV Adapter の場合です)

```
07/11 20:49:58|DEBUG|ds.studio|HELP_CONTENT_ID=adapter.csv_adapter.get_data
```

4. デバッグ情報の HELP_CONTENT_ID= で指定されている値が適切な target 属性値になります。

- url 属性

- target 属性で指定したアダプタ/オペレーションと関連させたいヘルプドキュメントのパスを指定します。
指定するパスは、\$DATASPIDER_HOME/server/doc/help/ja ディレクトリからの相対パスです。

上記例を参考にして、追加するアダプタのドキュメント情報を記述してください。

31.3.3. TOC ファイルの編集

TOC ファイルはマップファイルで設定したキーと、ヘルプに表示される目次の表示タイトルを設定しています。

たとえば、Simple CSV アダプタでは help/ja/SimpleCSV_AdapterHelpTOC.xml が該当し、次のように設定されています。

```
<tocitem text="サンプル" image="group">
. . .
  <tocitem text="Simple CSV" image="group">
    <tocitem text="読み取り処理" target="adapter.simple_csv_adapter.get_data" image="topic" />
    <tocitem text="書き込み処理" target="adapter.simple_csv_adapter.put_data" image="topic" />
  </tocitem>
. . .
</tocitem>
```

上記例では、<tocitem text="Simple CSV" image="group"> 要素の内容が Simple CSV アダプタの設定を表しています。この親要素の text 属性は"サンプル"と指定していますが、これは、Simple CSV アダプタがサンプルカテゴリに属するアダプタなので、text= "サンプル"の属性値を持った tocitem 要素の子要素として定義する必要があります。カテゴリを新規に作成した場合は、text= "サンプル"の属性値を持った tocitem 要素と同階層に、text= "新規カテゴリ名"の属性値を持った tocitem 要素を追加します。アダプタの tocitem 要素はその要素内に定義してください。

アダプタの tocitem 要素で定義する属性値についてご説明します。

- text 属性

- ヘルプのビューアに表示されるタイトル (目次) を定義します。

- image 属性

- group: このヘルプが子のヘルプを持つ場合に指定します。
- topic: このヘルプが子のヘルプを持たない場合に指定します。

- **target 属性**

- mapID 要素の target 属性で指定した値を設定します。

上記例を参考にして、追加するアダプタのドキュメント情報を記述してください。

31.4. ヘルプインデックスを再構築する

上記の作業で、HTML ファイルを追加し DataSpider Servista のヘルプ機構への追加も完了しました。最後に、ヘルプの検索機能で利用される各ヘルプのインデックスを再構築します。

再構築には「.synchronization」という名前でサイズ 0 バイトのファイルを

\$DATASPIDER_HOME/server/doc/help/ja ディレクトリに配置します。DataSpiderServer 起動時に

.synchronization ファイルが存在していると、ヘルプインデックスの再構築処理が実行されます。

31.5. 英語環境用ヘルプドキュメントを作成する

これまでの作業では日本語環境用ヘルプを作成しました。言語切り替え機能によって英語環境に切り替えた場合、ヘルプは表示されません。英語環境でヘルプを表示する場合、英語用ディレクトリにも同様にヘルプドキュメントを作成し、配置します。

ヘルプドキュメント

- \$DATASPIDER_HOME/server/doc/help/en

ヘルプセットファイル

- \$DATASPIDER_HOME/server/doc/help/< アダプタ名 >_en.hs

マップファイル

- \$DATASPIDER_HOME/server/doc/help/en/< アダプタ名 >TOC.xml

TOC ファイル

- \$DATASPIDER_HOME/server/doc/help/en/< アダプタ名 >Map.jhm

synchronization ファイル

- \$DATASPIDER_HOME/server/doc/help/en/.synchronization

31.5.1. 英語環境用ヘルプドキュメント編集

英語環境用のヘルプセットファイルは英語環境用のマップファイル、TOC ファイルのパスを参照するように変更してください。

また、必要に応じてヘルプドキュメントを英語で記述してください。

32. クラス図

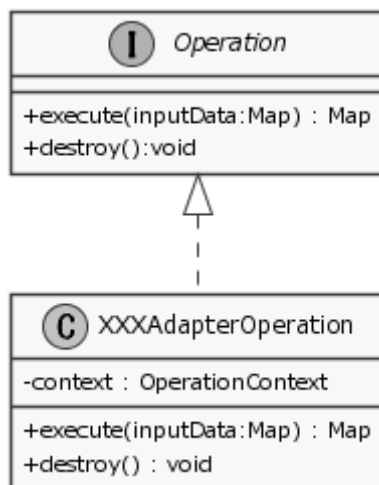


図 3. Operation クラス

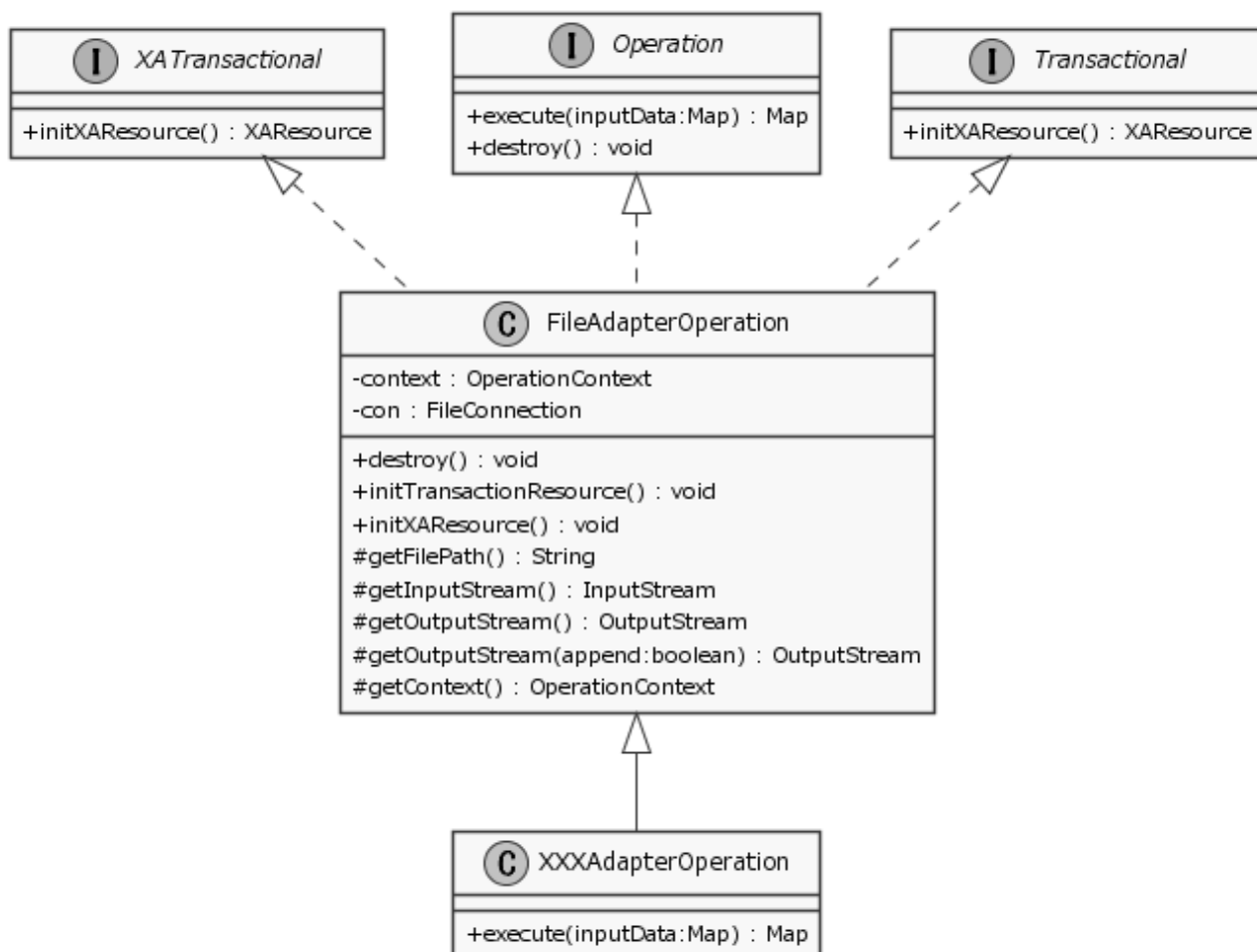


図 4. FileAdapterOperation クラス

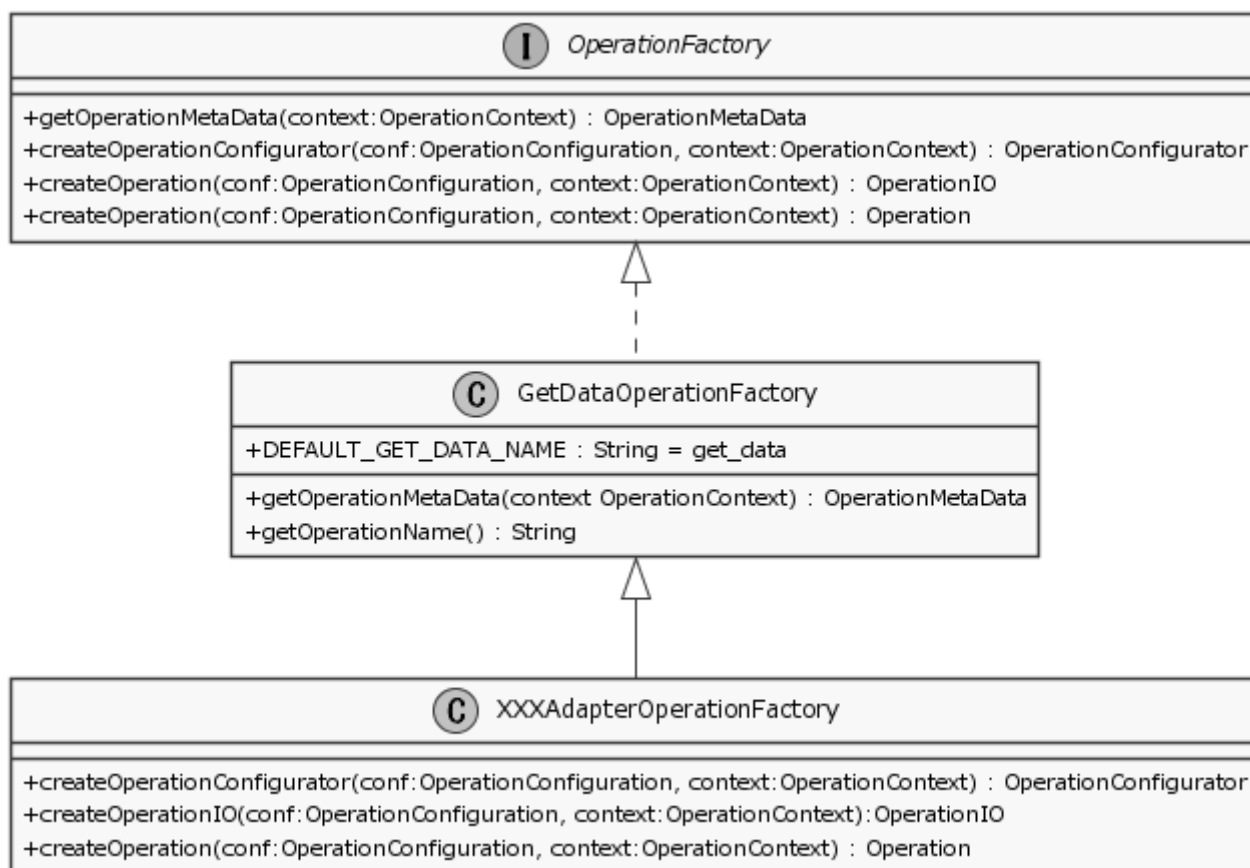


図 5. GetDataOperationFactory クラス

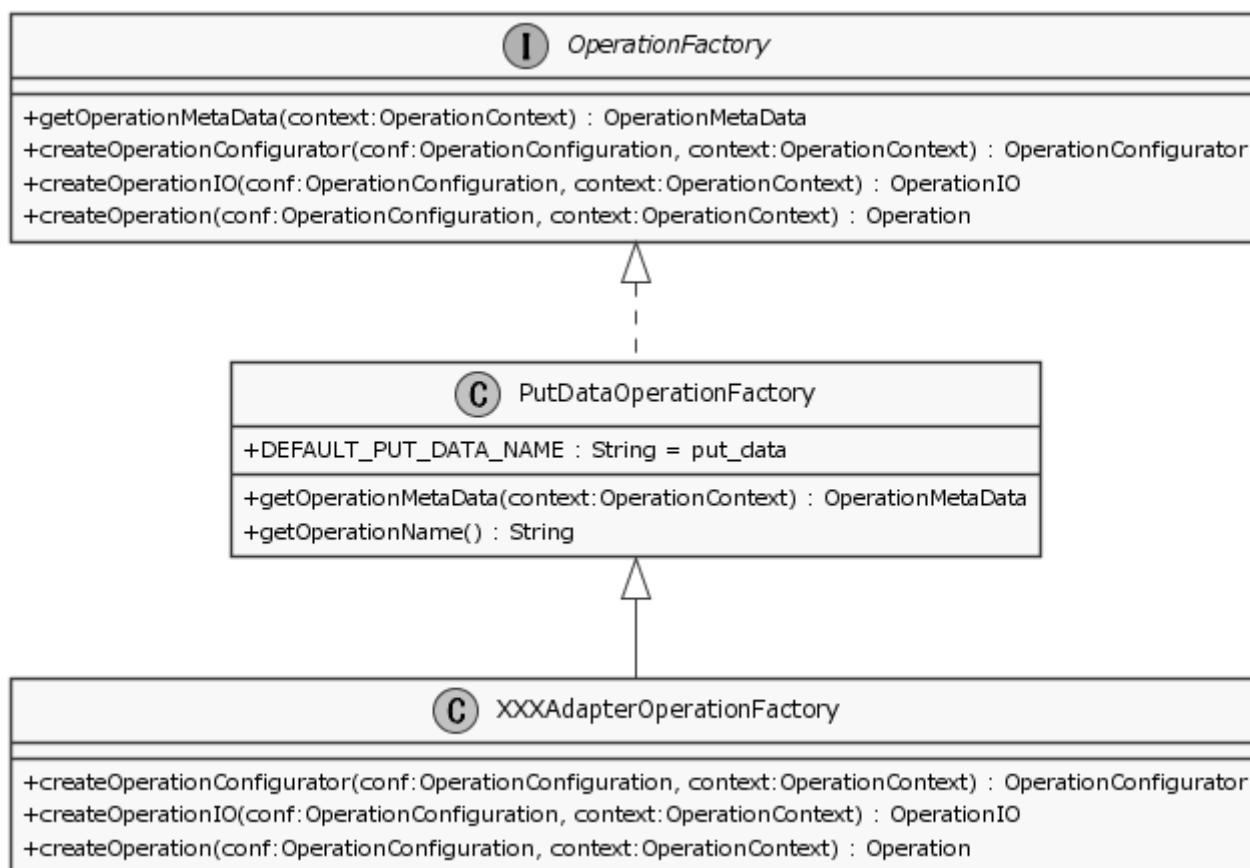


図 6. PutDataOperationFactory クラス

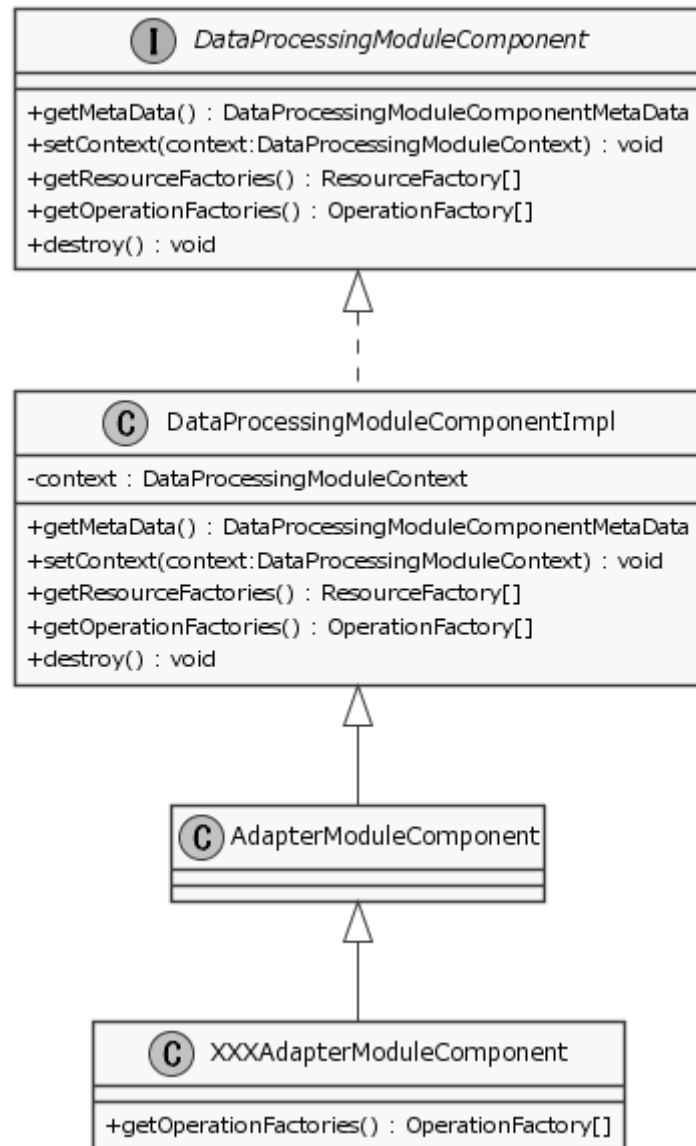


図 7. AdapterModuleComponent クラス

DataSpider Servista コンポーネントSDK 4.2 開発ガイド

第 1 版 2019.7.3

株式会社セゾン情報システムズ