Red Hat Process Automation Manager 7.4

ガイド付きルールテンプレートを使用したデシジョンサービスの作成

Last Updated: 2019-09-20

Red Hat Process Automation Manager 7.4 ガイド付きルールテンプレートを使用したデシジョンサービスの作成

Red Hat Customer Content Services
brms-docs@redhat.com

法律上の通知

概要

本書は、Red Hat Process Automation Manager 7.4 で、ガイド付きルールテンプレートを使用してデシジョンサービスを作成する方法を説明します。

目次

前書き

ビジネス分析者またはビジネスルールの開発者は、Business Central でガイド付きルールテンプレートデザイナーを使用してビジネスルールテンプレートを定義できます。このガイド付きルールテンプレートは Drools Rule Language (DRL) に組み込まれ、プロジェクトのデシジョンサービスの中心となります。

前提条件

- ガイド付きルールテンプレートのスペースおよびプロジェクトが Business Central に作成されていて、各アセットが、チームに割り当てられたプロジェクトに関連付けられている。詳細は『**デシジョンサービスの使用ガイド**』を参照してください。

第1章 RED HAT PROCESS AUTOMATION MANAGER におけるデシジョン作成アセット

Red Hat Process Automation Manager は、デシジョンサービスにビジネスデシジョンを定義するのに使用可能なアセットを複数サポートします。デシジョン作成アセットはそれぞれ長所が異なるため、ゴールおよびニーズに合わせて、アセットを 1 つ、または複数を組み合わせて使用できます。

以下の表では、デシジョンサービスでデシジョンを定義する最適な方法を選択できるように、Red Hat Process Automation Manager プロジェクトでサポートされている主要なデシジョン作成アセットを紹介します。

表1.1 Red Hat Process Automation Manager でサポートされるデシジョン作成アセット

アセット	主な特徴	オーサリングツール	ドキュメンテーション
DMN (Decision Model and Notation) モデル	• Object Management Group (OMG) が定義する標準記法をもとにしたデシジョンモデルである • 1 つまたは複数の DRG (decision requirements graph) を含むグラフィカルな DRD (decision requirements diagram) を使用してビジネスデシジョンのフローを追跡する • DMN モデルが DMN 準拠プラットフォーム間で共有できるようにする XML スキーマを使用する • DMN デシジョンテーブルおよび他の DMN ボックス式表現 (Boxed Expression) でデシジョンロジックを定義する Friendly Enough Expression Language (FEEL) をサポートする • Business Process Model and Notation (BPMN) プロセスモデルと効率的に統合できる • 包括性、具体性および安定性のあるデシジョンフローの作成に最適である	Business Central または DMN 準拠のエディター	『DMN モデルを使用したデシジョンサービスの作成』

アセット	主な特徴	オーサリングツール	ドキュメンテーション
ガイド付きデシジョンテーブル	• Business Central の UI ベースのテーブルデザイナーで作成するルールのテーブル • デシジョンテーブルをスプレッドシートで対応する代わりにウィザードで対応する • 条件を満たした入力に、フィールドとオプションを提供する • ルールテンプレートを作成するテンプレートキーと値をサポートする • その他のアセットではサポートされていない、ヒットポリシー、リアルタイム検証などの追加機能をサポートする • コンパイルエラーを最小限に抑えるため、制限されているテーブル形式でルールを作成するのに最適	Business Central	『ガイド付きデシジョンテーブルを使用したデシジョンサービスの作成』
スプレッドシートのデシジョンテーブル	• Business Central にアップロード可能な XLS または XLSX スプレッドシート形式のデシジョンテーブルである • ルールテンプレートを作成するテンプレートキーと値をサポートする • Business Central 外で管理しているデシジョンテーブルでルールを作成するのに最適 • アップロード時に適切にルールをコンパイルするために厳しい構文要件がある	スプレッドシートエディター	『アップロードしたデシジョンテーブルを使用したデシジョンサービスの作成』
ガイド付きルール	• Business Central の UI ベースのルールデザイナーで作成する個々のルール • 条件を満たした入力に、フィールドとオプションを提供する • コンパイルエラーを最小限に抑えるため、制御されている形式で単独のルールを作成するのに最適	Business Central	『ガイド付きルールを使用したデシジョンサービスの作成』

アセット	主な特徴	オーサリングツール	ドキュメンテーション
ガイド付きルールテンプレート	• Business Central の UI ベースのテンプレートデザイナーで作成する再利用可能なルール構造 • 条件を満たした入力に、フィールドとオプションを提供する • (このアセットの基本となる) ルールテンプレートを作成するテンプレートキーと値をサポートする • ルール構造が同じで、定義したフィールド値が異なるルールを多数作成するのに最適	Business Central	『ガイド付きルールテンプレートを使用したデシジョンサービスの作成』
DRL ルール	• **.drl** テキストファイルに直接定義する個々のルール • 最も柔軟性が高く、ルールと、ルール動作に関するその他の技術を定義できる • スタンドアロン環境で作成し、Red Hat Process Automation Manager に統合可能 • 高度な DRL オプションが必要なルールを作成するのに最適 • ルールを適切にコンパイルするために厳密な構文要件がある	Business Central または統合開発環境 (IDE)	『DRL ルールを使用したデシジョンサービスの作成』

アセット	主な特徴	オーサリングツール	ドキュメンテーション
予測モデルマークアップ言語 (PMML: Predictive Model Markup Language) モデル	• Data Mining Group (DMG) で定義した標準記法を元にした予測データ分析モデルである • PMML モデルが PMML 準拠プラットフォーム間で共有できるようにする XML スキーマを使用する • 回帰、スコアカード、ツリー、マイニングなどのモデルタイプをサポートする • スタンドアロンの Red Hat Process Automation Manager プロジェクトに追加したり、Business Central のプロジェクトにインポートしたりできる • Red Hat Process Automation Manager のデシジョンサービスに予測データを統合するのに最適である	PMML または XML エディター	『PMML モデルを使用したデシジョンサービスの作成』

第2章 ガイド付きルールテンプレート

ガイド付きルールテンプレートは、別のデータテーブルに定義した実際の値と置き換えるプレースホルダー値 (テンプレートキー) を使用したビジネスルール構造です。そのテンプレートに対応するデータテーブルに定義する各行の結果がルールになります。ガイド付きルールテンプレートは、多くのルールで条件、アクションなどの属性が同じで、ファクトまたは制約の値が異なる場合に適しています。このような場合、同じようなガイド付きルールを多数作成し、各ルールに値を定義する代わりに、各ルールに適用できるルール構造を持つガイド付きルールテンプレートを作成し、データテーブルに値だけを定義します。

ガイド付きルールテンプレートデザイナーは、ルールテンプレートを定義したデータオブジェクトと、テンプレートキー値を追加するデータテーブルに基づいて適切なテンプレート入力を行うフィールドとオプションを提供します。ガイド付きルールテンプレートを作成し、対応するデータテーブルに値を追加すると、定義したルールが、その他のすべてのルールアセットとともに Drools Rule Language (DRL) ルールにコンパイルされます。

ガイド付きルールテンプレートに関連するすべてのデータオブジェクトは、ガイド付きルールテンプレートと同じプロジェクトパッケージに置く必要があります。同じパッケージに含まれるアセットはデフォルトでインポートされます。必要なデータオブジェクトとガイド付きルールテンプレートを作成したら、ガイド付きルールテンプレートデザイナーの **Data Objects** タブから、必要なデータオブジェクトがすべてリストされていることを検証したり、 **新規アイテム** を追加してその他の既存データオブジェクトをインポートしたりできます。

第3章 データオブジェクト

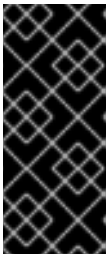
データオブジェクトは、作成するルールアセットの構成要素です。データオブジェクトは、プロジェクトで指定したパッケージに Java オブジェクトして実装されているカスタムのデータ型です。たとえば、データフィールド **Name**、**Address**、および **Date of Birth** を使用して **Person** オブジェクトを作成し、ローン申し込みルールに詳細な個人情報を指定できます。このカスタムのデータ型は、アセットとディシジョンサービスがどのデータに基づいているかを指定します。

3.1. データオブジェクトの作成

以下の手順は、データオブジェクトを作成する際の一般的な概要で、特定のビジネスアセットに固有のものではありません。

手順

1. Business Central で、**Menu → Design → Projects** に移動して、プロジェクト名をクリックします。
2. **Add Asset → Data Object** をクリックします。
3. 一意の **データオブジェクト** 名を入力し、**パッケージ** を選択します。これにより、その他のルールアセットでもデータオブジェクトを利用できるようになります。同じパッケージに、同じ名前のデータオブジェクトを複数作成することはできません。指定の DRL ファイルで、どのパッケージからでもデータオブジェクトをインポートできます。

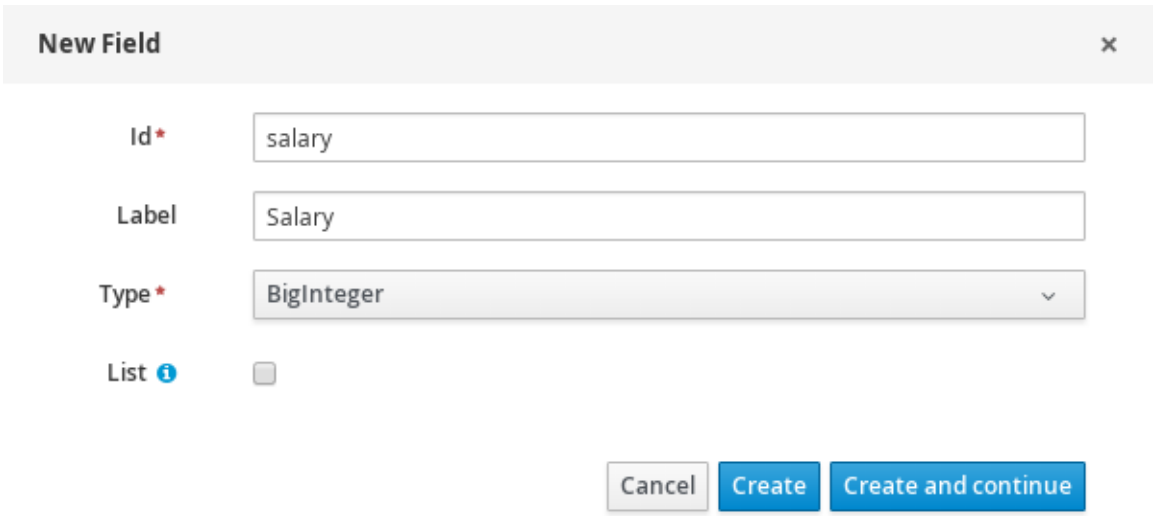


別のパッケージからのデータオブジェクトのインポート

別のパッケージから直接、ガイド付きルールやガイド付きデシジョンテーブルデザイナーなどのアセットデザイナーに、既存のデータオブジェクトをインポートすることができます。プロジェクトで関連するルールアセットを選択し、アセットデザイナーで **Data Objects → New item**に移動して、インポートするオブジェクトを選択します。

4. データオブジェクトを永続化するには、**Persistable** チェックボックスを選択します。永続型データオブジェクトは、JPA 仕様に準じてデータベースに保存できます。デフォルトの JPA は Hibernate です。
5. **OK** をクリックします。
6. データオブジェクトデザイナーで **add field** をクリックして、**Id** 属性、**Label** 属性、**Type** 属性を使用するオブジェクトにフィールドを追加します。必須属性にはアスタリスク (*) マークが付いています。
    - **Id:** フィールドの一意の ID を入力します。
    - **Label:** (任意) フィールドのラベルを入力します。
    - **Type:** フィールドのデータ型を入力します。
    - **List:** (任意) このチェックボックスを選択すると、このフィールドで、指定したタイプのアイテムを複数保持できるようになります。

図3.1 データオブジェクトへのデータフィールドの追加



7. **Create** をクリックして、新しいフィールドを追加します。**Create and continue** をクリックすると、新しいフィールドが追加され、別のフィールドを引き続き作成できます。

注記

フィールドを編集するには、フィールド行を選択し、画面右側の **general properties** を使用します。

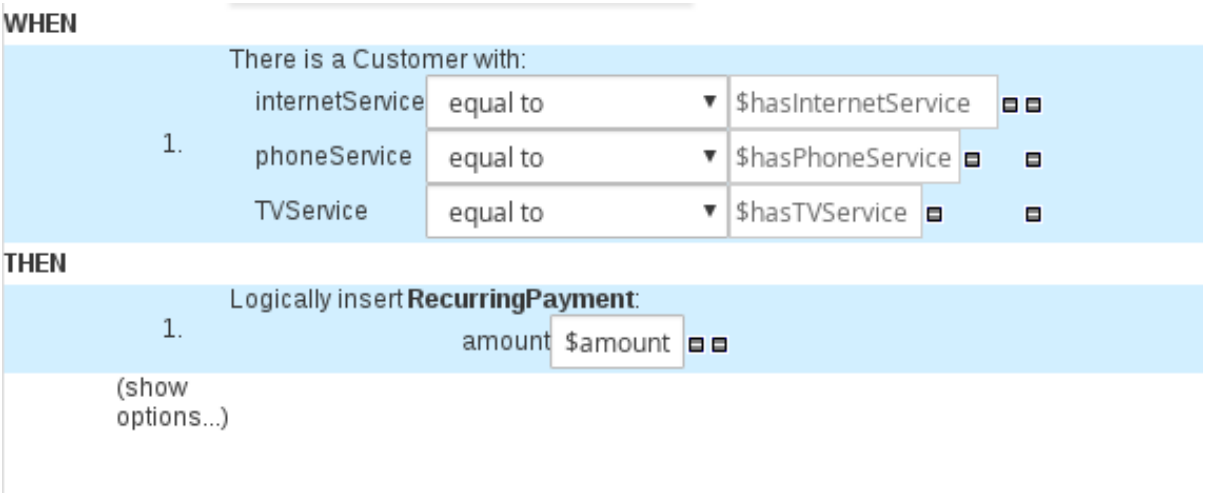
第4章 ガイド付きルールテンプレートの作成

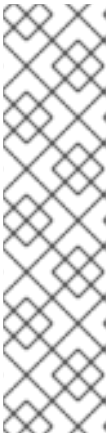
ガイド付きルールテンプレートを使用して、データテーブルに定義した値に対応するプレースホルダー値 (テンプレートキー) を持つルール構造を定義できます。ガイド付きルールテンプレートは、構造が同じ多数のガイド付きルールセットを個別に定義するのに利用できる効率的な方法です。

手順

1. Business Central で、**Menu → Design → Projects** に移動して、プロジェクト名をクリックします。
2. **Add Asset → Guided Rule Template** をクリックします。
3. 参考となる **ガイド付きルールテンプレート** 名を入力し、適切な **パッケージ** を選択します。指定するパッケージは、必要なデータオブジェクトが割り当てられている、またはこれから割り当てるパッケージにする必要があります。
4. **OK** をクリックして、ルールテンプレートを作成します。
新しいガイド付きルールテンプレートが、**Project Explorer** の **Guided Rule Templates** パネルに追加されます。
5. **Data Objects** タブをクリックして、ルールに必要なデータオブジェクトがすべてリストされていることを確認します。リストされていない場合は、**New item** をクリックして、他のパッケージからデータオブジェクトをインポートするか、パッケージに データオブジェクトを作成 します。
6. データオブジェクトをすべて配置したら、**Model** タブに戻り、ウィンドウの右側のボタンから、利用可能なデータオブジェクトに、ルールテンプレートの **WHEN** (条件) セクションおよび **THEN** (アクション) セクションを追加して定義します。ルールごとに変更するフィールド値については、ルールデザイナーで **$key** 形式、または (DRL を使用している場合は) free form DRL の **@{key}** 形式のテンプレートキーを使用します。

図4.1 ガイド付きルールテンプレートの例





テンプレートキーに関する注記

テンプレートキーは、ガイド付きルールテンプレートの基本となるものです。テンプレートキーは、同じテンプレートから異なるルールを生成するために、対応するデータテーブルで定義する実際の値と入れ替えるテンプレートのフィールド値を有効にするものです。**リテラル** や **式** などの値を使用することもできますが、その値は、そのテンプレートに基づいたすべてのルールのルール構造の一部となります。ただし、ルール間で異なる値については、特定のキーを使用した **テンプレートキー** フィールドタイプを使用します。ガイド付きルールテンプレートでテンプレートキーを使用しない場合、対応するデータテーブルはテンプレートデザイナーに生成されず、テンプレートは、個々のガイド付きルールとして機能します。

ルールテンプレートの **WHEN** 部分は、アクションを実行するのに必要な条件が含まれます。たとえば、電話会社が、顧客が契約したサービス内容 (インターネット、電話、およびテレビ番組) に基づいて利用代金を請求する場合、**WHEN** 条件の 1 つは **internetService | equal to | $hasInternetService** となります。テンプレートキー **$hasInternetService** は、そのテンプレートのデータテーブルに定義した実際のブール値 (**true** または **false**) に置き換えられます。

ルールテンプレートの **THEN** 部分には、ルールの条件部分に一致したときに実行するアクションが含まれます。たとえば、顧客がインターネットサービスだけを契約した場合、**RecurringPayment** の **$amount** テンプレートキーを使用した **THEN** アクションには、データテーブルのインターネットサービス料金に定義した整数値が実際の月額に設定されます。

7. ルールのコンポーネントをすべて定義したら、ガイド付きルールテンプレートデザイナーで **Save** をクリックして、設定した内容を保存します。

4.1. ガイド付きルールテンプレートへの WHEN 条件の追加

ルールの **WHEN** 部分は、アクションを実行するのに必要な条件が含まれます。たとえば、電話会社が、顧客が契約したサービス内容 (インターネット、電話、およびテレビ番組) に基づいて利用料金を請求する場合、**WHEN** 条件の 1 つは **internetService | equal to | $hasInternetService** になります。テンプレートキー **$hasInternetService** は、そのテンプレートのデータテーブルに定義した実際のブール値 (**true** または **false**) に置き換えられます。

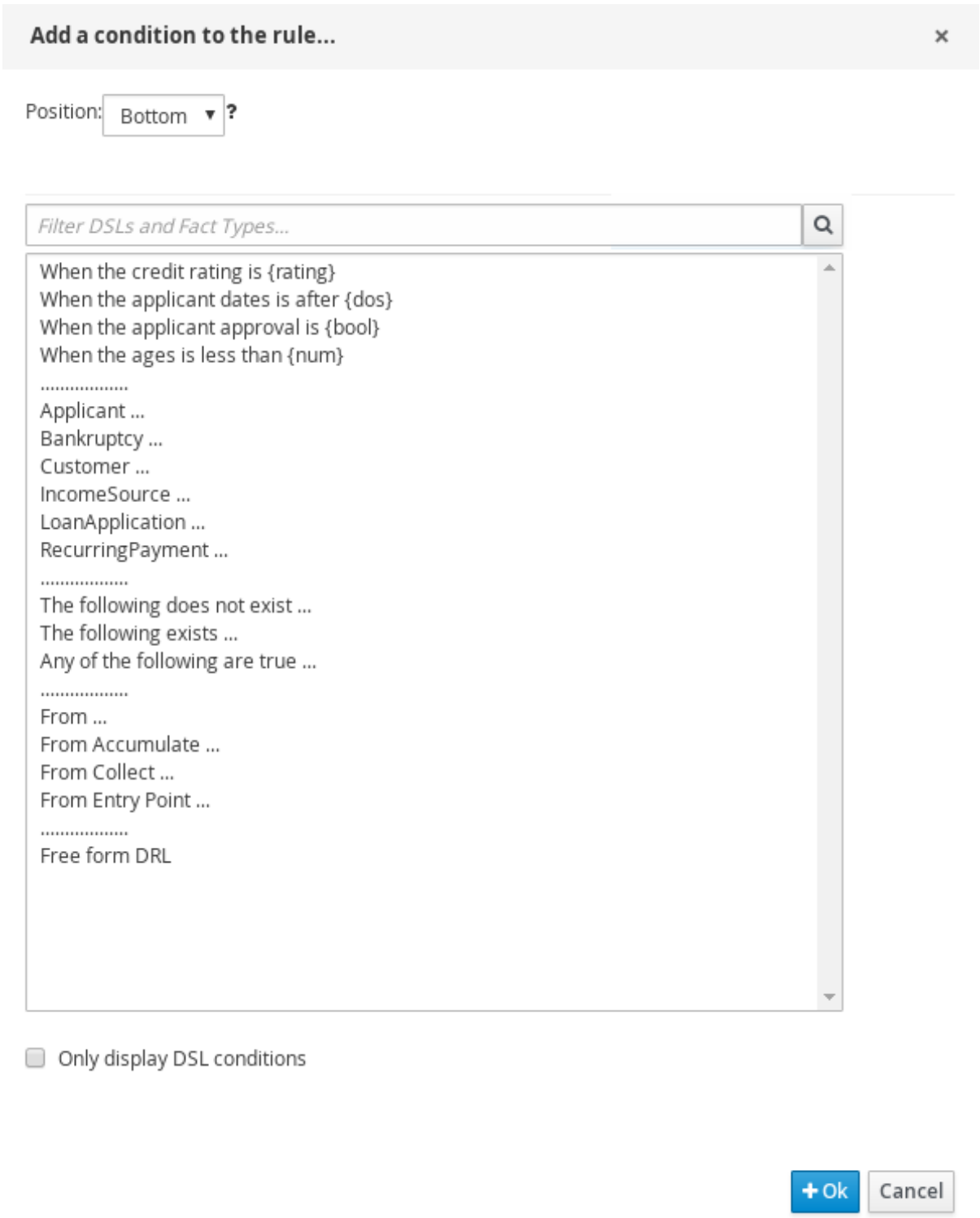
前提条件

- ルールに必要なデータオブジェクトがすべて作成、またはインポートされていて、ガイド付きルールテンプレートデザイナーの **Data Objects** タブにリストされている。

手順

1. ガイド付きルールテンプレートデザイナーで、**WHEN** セクションの右側のプラスアイコン ( ) をクリックします。
利用可能な条件要素が追加された **Add a condition to the rule**ウィンドウが開きます。

図4.2 ルールへの条件の追加



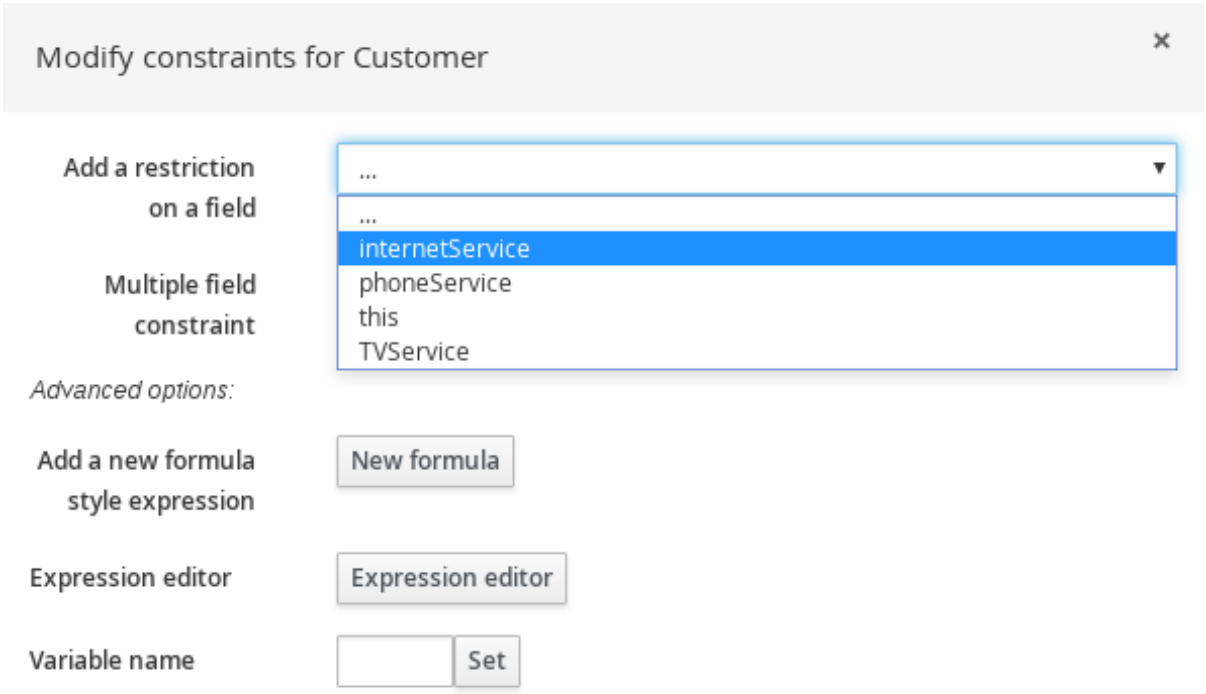
このリストには、ガイド付きルールテンプレートデザイナーの **Data Objects** タブのデータオブジェクトと、パッケージに定義した DSL オブジェクトと、以下の標準オプションが含まれます。

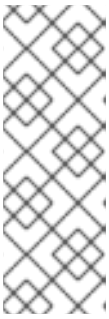
- **The following does not exist:** 存在すべきでないファクトと制約を指定します。
- **The following exists:** 存在すべきファクトと制約を指定します。このオプションは、最初に一致したものだけが適用され、その後一致するものは無視されます。
- **Any of the following are true:** true であるファクトと制約をリストします。
- **From:** ルールに **From** 条件要素を定義します。

- **From Accumulate:** ルールの **Accumulate** 条件要素を定義します。
- **From Collect:** ルールの **Collect** 条件要素を定義します。
- **From Entry Point:** パターンの **Entry Point** を定義します。
- **Free form DRL:** free form DRL フィールドを挿入します。このフィールドには、ガイド付きルールデザイナーを使用せずに、自由に条件要素を定義できます。free form DRL のテンプレートキーには、**@{key}** 形式を使用します。

2. 条件要素 (**Customer** など) を選択し、**OK** をクリックします。
3. ガイド付きルールテンプレートデザイナーで条件要素をクリックし、**Modify constraints for Customer** ウィンドウで、フィールドへの制限の追加、複数のフィールド制約の適用、新しい数式表現の追加、式エディターの適用、または変数名の設定を行います。

図4.3 条件の変更





注記

変数名を使用すると、ガイド付きルールの別の構成でファクトまたはフィールドを指定できます。たとえば、**Customer** の変数を **c** にし、**Customer** が **Applicant** であることを指定する別の **Applicant** 制約で、**c** を参照します。

```
c : Customer()
Applicant( this == c )
```

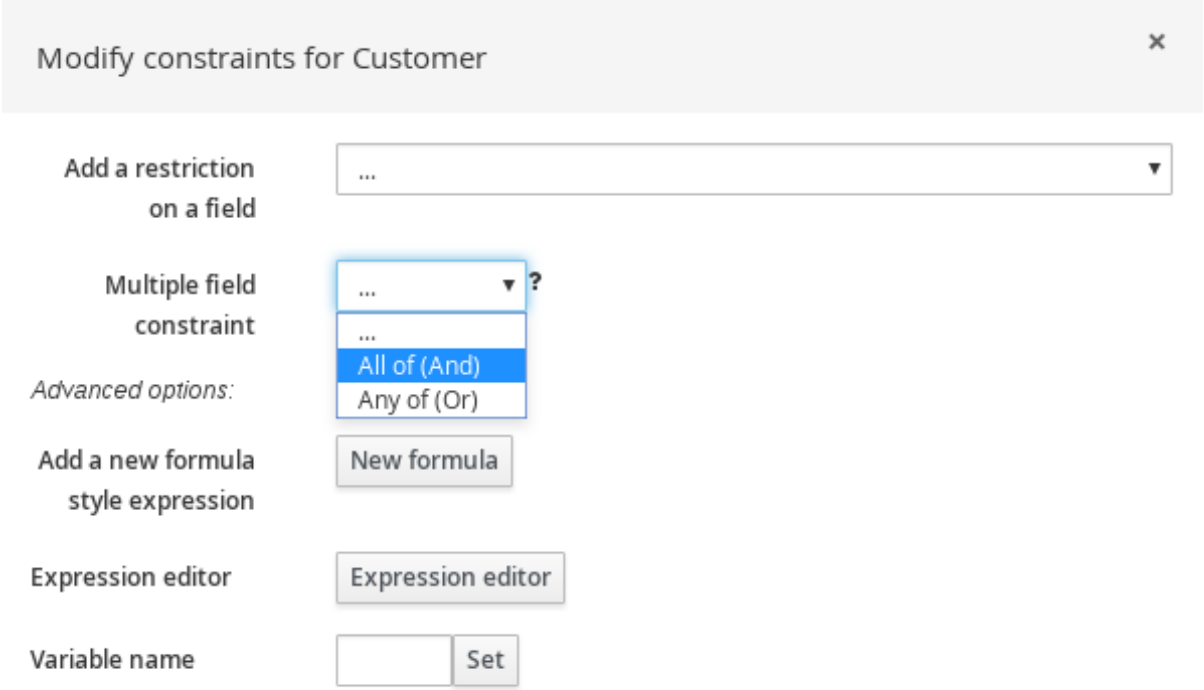
制約を選択したら、ウィンドウが自動的に閉じます。

4. 追加した制約の隣にあるドロップダウンメニューから、制限の演算子 (**greater than** など) を選択します。
5. 編集アイコン () をクリックして、フィールド値を定義します。

6. このテンプレートに基づいたルール間で値が異なる場合は、**テンプレートキー** を選択し、**$key** 形式でテンプレートキーを追加します。これにより、フィールド値を、対応するデータテーブルに定義する実際の値と入れ替えて、同じテンプレートから異なるルールを生成します。ルールテンプレートの中で、ルール間で変更しないフィールド値については、別の型の値を使用できます。

7. フィールド制約を複数適用するには、条件をクリックし、**Modify constraints for Customer** ウィンドウで、**Multiple field constraint** ドロップダウンメニューから **All of(And)** または **Any of(Or)** を選択します。

図4.4 複数のフィールド制約の追加



8. ガイド付きルールテンプレートデザイナーで制約をクリックして、フィールド値をさらに定義します。

9. 条件要素をすべて定義したら、ガイド付きルールテンプレートデザイナーで **Save** をクリックして、設定した内容を保存します。

4.2. ガイド付きルールテンプレートへの THEN アクションの追加

ルールテンプレートの **THEN** 部分には、ルールの条件部分に一致したときに実行するアクションが含まれます。たとえば、顧客がインターネットサービスだけを契約した場合、**RecurringPayment** の **$amount** テンプレートキーを使用した **THEN** アクションには、データテーブルのインターネットサービス料金に定義した整数値が実際の月額に設定されます。

前提条件

- ルールに必要なデータオブジェクトがすべて作成、またはインポートされていて、ガイド付きルールテンプレートデザイナーの **Data Objects** タブにリストされている。

手順

1. ガイド付きルールテンプレートデザイナーで、**THEN** セクションの右側のプラスアイコン ( ) をクリックします。

利用可能なアクション要素が追加された **Add a new action** ウィンドウが開きます。

図4.5 ルールへのアクションの追加

このリストには、ガイド付きルールテンプレートデザイナーの **Data Objects** タブのデータオブジェクトと、パッケージに定義した DSL オブジェクトに基づいた、挿入および修正のオプションが含まれます。

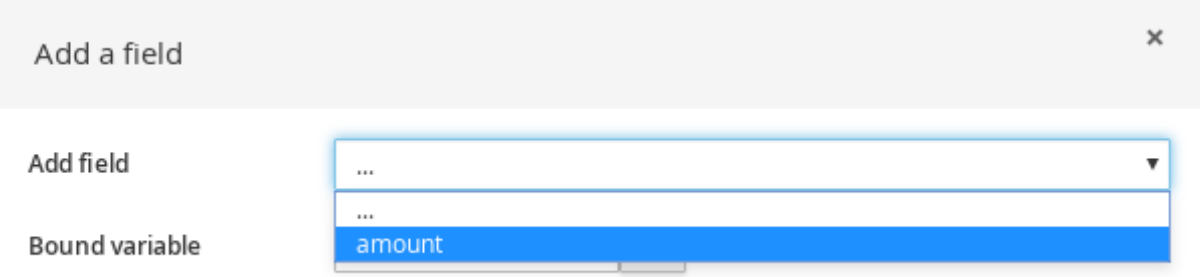
- **Insert fact (ファクトの挿入):** ファクトを挿入し、ファクトの結果フィールドと値を定義します。
- **Logically Insert fact (ファクトの論理的な挿入):** ファクトをデシジョンエンジンに論理的に挿入し、ファクトに対してフィールドと値を定義します。デシジョンエンジンは、ファク

トの挿入および取り消しに対して論理的な決断を行います。定期的な挿入、または指定した挿入の後に、ファクトを明示的に取り消す必要があります。論理挿入の後に、ファクトをアサートした条件が TRUE ではなくなると、ファクトは自動的に取り消されます。

- **Add free form DRL:**free form DRL フィールドを挿入します。このフィールドには、ガイド付きルールデザイナーを使用せずに、自由に条件要素を定義できます。free form DRL のテンプレートキーには、**@{key}** 形式を使用します。

2. アクション要素 (**Logically Insert fact RecurringPayment** など) を選択し、**OK** をクリックします。
3. ガイド付きルールテンプレートデザイナーでアクション要素をクリックし、**Add a field** ウィンドウを使用してフィールドを選択します。

図4.6 フィールドの追加



フィールドを選択したら、ウィンドウが自動的に閉じます。

4. 編集アイコン ( ) をクリックして、フィールド値を定義します。
5. このテンプレートに基づいたルール間で値が異なる場合は、**テンプレートキー** を選択し、**$key** 形式でテンプレートキーを追加します。これにより、フィールド値を、対応するデータテーブルに定義する実際の値と入れ替えて、同じテンプレートから異なるルールを生成します。ルールテンプレートの中で、ルール間で変更しないフィールド値については、別の型の値を使用できます。
6. アクション要素をすべて定義したら、ガイド付きルールテンプレートで **Save** をクリックして、設定した内容を保存します。

4.3. その他のルールオプションの追加

ルールデザイナーを使用してルールにメタデータを追加し、追加のルール属性 (**salience**、**no-loop** など) を定義し、条件またはアクションの変更を制限するために、ルールの領域を凍結します。

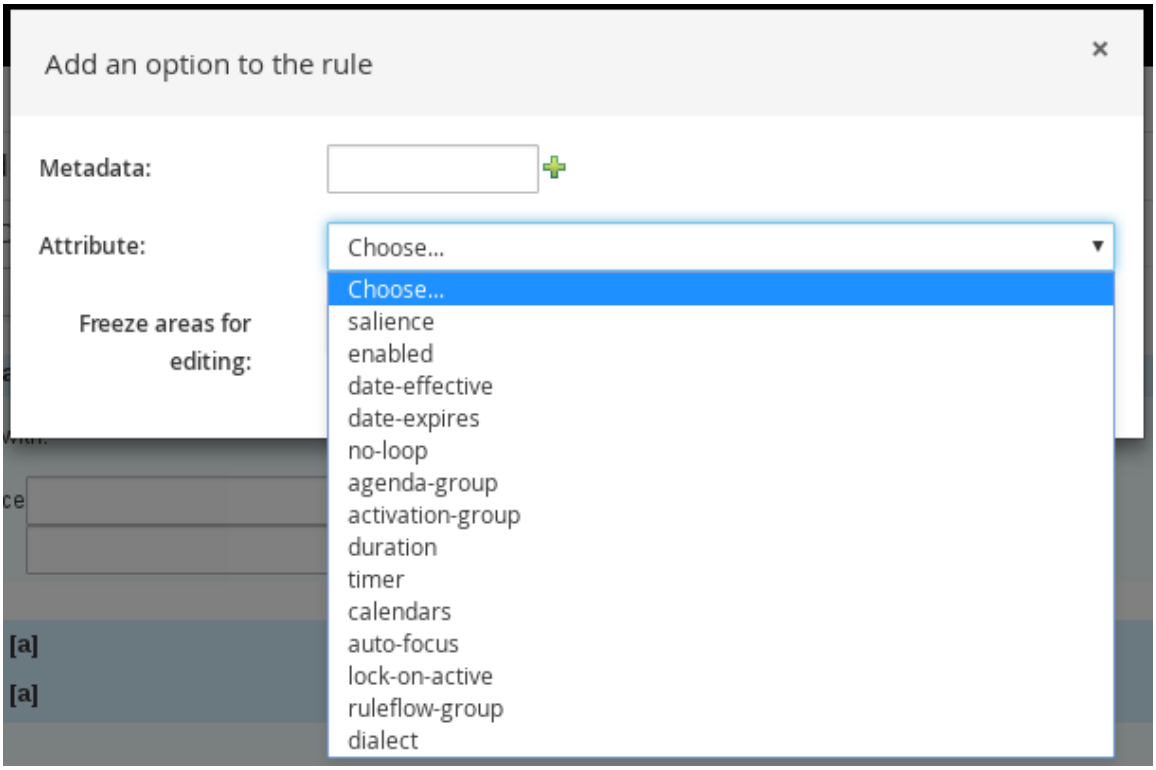
手順

1. ルールデザイナーの **THEN** セクションの下にある **(show options...)** をクリックします。
2. ウィンドウの右側にあるプラスアイコン ( ) をクリックして、オプションを追加します。
3. ルールに追加するオプションを選択します。

- **Metadata:** メタデータのラベルを入力し、プラスアイコン ( ) をクリックします。次に、ルールデザイナーに提供されるフィールドに必要なデータを入力します。

- **Attribute:** ルール属性のリストから選択します。次に、ルールデザイナーに表示されるフィールドまたはオプションに値を定義します。
- **Freeze areas for editing (編集する領域を制限):**ルールデザイナーで修正する領域を制限する **条件** または **アクション** を選択します。

図4.7 ルールオプション



4. ルールデザイナーで **Save** をクリックして、設定した内容を保存します。

4.3.1. ルールの属性

ルール属性は、ルールの動作を修正するビジネスルールを指定する追加設定です。次の表では、ルールに割り当て可能な属性の名前と、対応する値を紹介します。

表4.1 ルールの属性

属性	値
salience	ルールの優先順位を定義する整数。ルールの salience 値を高くすると、アクティベーションキューに追加したときの優先順位が高くなります。 例: **salience 10**
enabled	ブール値。このオプションを選択すると、ルールが有効になります。このオプションを選択しないと、ルールは無効になります。 例: **enabled true**
date-effective	日付定義および時間定義を含む文字列。現在の日時が **date-effective** 属性よりも後の場合は、このルールがアクティブになります。 例: **date-effective "4-Sep-2018"**

属性	値
date-expires	日時定義を含む文字列。現在日時が **date-expires** 属性よりも後になると、このルールをアクティブにすることはできません。 例: **date-expires "4-Oct-2018"**
no-loop	ブール値。このオプションを設定すると、以前一致した条件がこのルールにより再トリガーとなる場合に、このルールを再度アクティブにする (ループする) ことができません。条件を選択しないと、この状況でルールがループされます。 例: **no-loop true**
agenda-group	ルールを割り当てるアジェンダグループを指定する文字列。アジェンダグループを使用すると、アジェンダをパーティションで区切り、ルールのグループに対する実行をさらに制御できます。フォーカスを取得したアジェンダグループのルールだけがアクティブになります。 例: **agenda-group "GroupName"**
activation-group	ルールを割り当てるアクティベーション (または XOR) グループを指定する文字列。アクティベーショングループでアクティブにできるルールは 1 つだけです。最初のルールが実行されると、アクティベーショングループの中で、アクティベーションが保留中のルールはすべてキャンセルされます。 例: **activation-group "GroupName"**
duration	ルールの条件が一致している場合に、ルールがアクティブになってからの時間をミリ秒で定義する長整数値。 例: **duration 10000**
timer	ルールのスケジュールに対する **int** (間隔) または **cron** タイマー定義を指定する文字列。 例: **timer "*/5 * * * *"** (5 分ごと)
calendar	ルールのスケジュールを指定する Quartz カレンダーの定義。 例: **calendars "* * 0-7,18-23 ? * *"** (営業時間外を除く)
auto-focus	アジェンダグループ内のルールにのみ適用可能なブール値。このオプションが選択されている場合は、次にルールがアクティブになると、そのルールが割り当てられたアジェンダグループに自動的にフォーカスが移ります。 例: **auto-focus true**

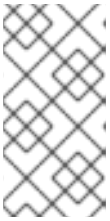
属性	値
lock-on-active	ルールフローグループまたはアジェンダグループ内のルールにのみ適用可能なブール値。このオプションを選択すると、次回、ルールのルールフローグループがアクティブになるか、ルールのアジェンダグループがフォーカスを受け取ると、(ルールフローグループがアクティブでなくなるか、アジェンダグループがフォーカスを失うまで) ルールをアクティブにすることができません。これは、**no-loop** 属性を強力にしたものです。なぜなら、一致するルールのアクティベーションが、(ルールそのものによるものだけでなく) アップデート元にかかわらず破棄されるためです。この属性は、ファクトを修正するルールが多数あり、ルールの再一致と再発行を希望しない計算ルールに適しています。 例: **lock-on-active true**
ruleflow-group	ルールフローグループを指定する文字列。ルールフローグループで、関連するルールフローによってそのグループがアクティブになった場合に限りルールを発行できます。 例: **ruleflow-group "GroupName"**
dialect	ルールのコード表記に使用される言語を指定する文字列 (**JAVA** または **MVEL**)。デフォルトでは、ルールは、パッケージレベルに指定されている方言を使用します。ここで指定した方言は、ルールのパッケージ方言設定を上書きします。 例: **dialect "JAVA"**

第5章 ガイド付きルールテンプレートのデータテーブルの定義

ガイド付きルールテンプレートを作成し、フィールド値にテンプレートキーを追加すると、ガイド付きルールテンプレートデザイナーの **Data** テーブルにデータテーブルが表示されます。データテーブルの各列は、ガイド付きルールテンプレートに追加したテンプレートキーに対応します。このテーブルを使用して、各テンプレートキーに値を 1 行ずつ定義します。そのテンプレートのデータテーブルに定義する値の各行の結果がルールになります。

手順

1. ガイド付きルールテンプレートデザイナーで、**Data** タブをクリックしてデータテーブルを表示します。データテーブルの各列は、ガイド付きルールテンプレートに追加したテンプレートキーに対応します。



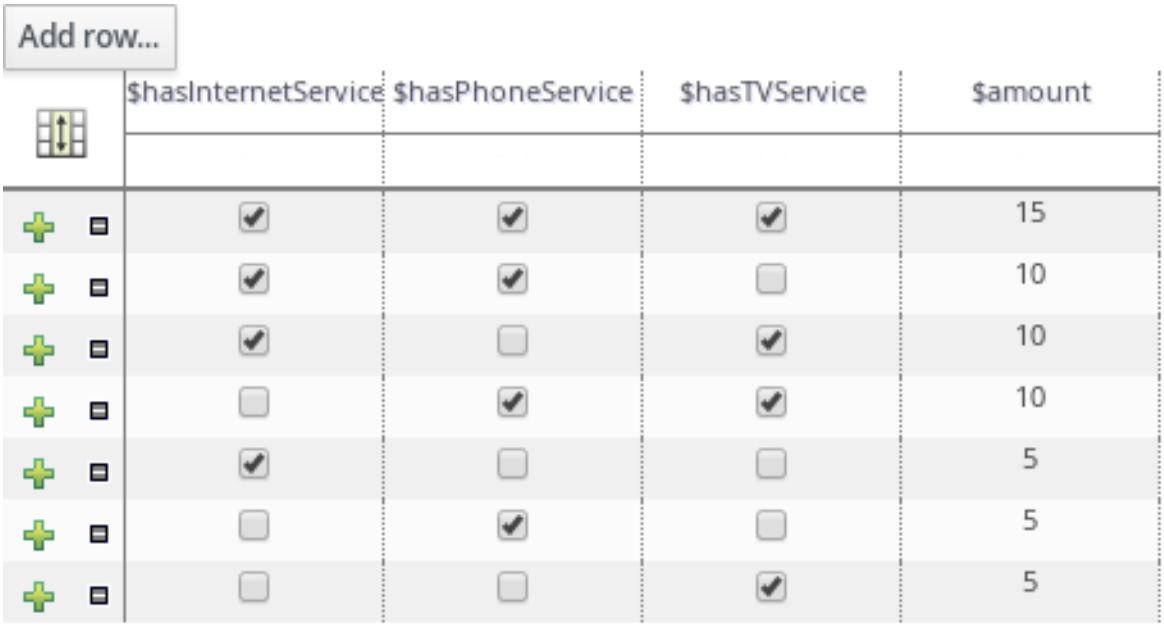
注記

ルールテンプレートにテンプレートキーを追加していないと、このデータテーブルは表示されず、テンプレートが正式なテンプレートとして機能しません。代わりに、個々のガイド付きルールとして機能します。このため、テンプレートキーは、ガイド付きルールテンプレートを作成するための基本となります。

2. **Add row** をクリックして、各テンプレートキー列にデータ値を定義して、ルール (行) を生成します。

3. 引き続き、行を追加して、生成する各ルールにデータ値を定義します。**Add row** をクリックするか、プラスアイコン () をクリックして行を追加するか、マイナスアイコンをクリックして行を削除します。

図5.1 ガイド付きルールテンプレートのデータテーブルの例



DRL コードを表示するには、ガイド付きルールテンプレートデザイナーで **Source** タブをクリックします。

例:

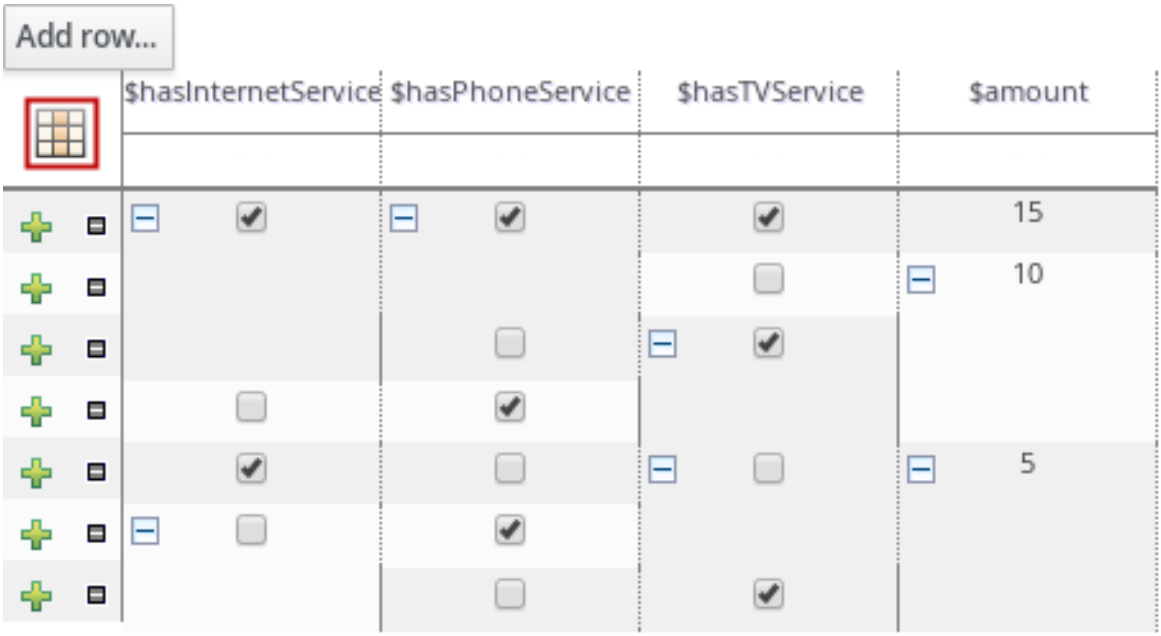
```
rule "PaymentRules_6"
```

```
 dialect "mvel"
 when
  Customer( internetService == false ,
   phoneService == false ,
   TVService == true )
 then
  RecurringPayment fact0 = new RecurringPayment();
  fact0.setAmount( 5 );
  insertLogical( fact0 );
end

rule "PaymentRules_5"
 dialect "mvel"
 when
  Customer( internetService == false ,
   phoneService == true ,
   TVService == false )
 then
  RecurringPayment fact0 = new RecurringPayment();
  fact0.setAmount( 5 );
  insertLogical( fact0 );
end
...
//Other rules omitted for brevity.
```

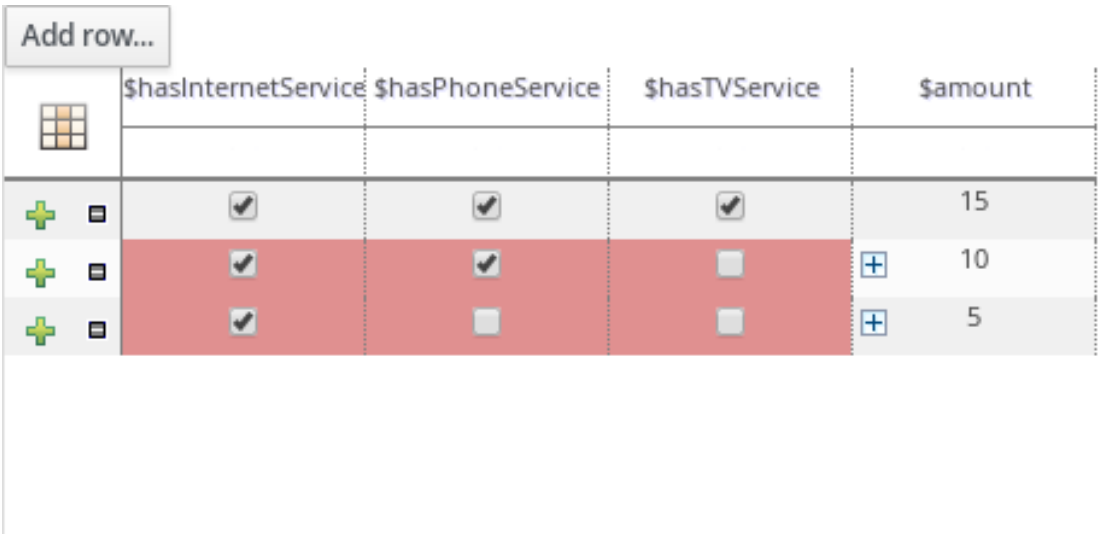
4. 必要に応じて、見やすくするために、データテーブルの左上にあるグリッドアイコンをクリックして、セルの結合をオンまたはオフに切り替えます。同じ列で、値が同じセルは、1つのセルに結合されます。

図5.2 データテーブルでセルの結合



次に、新たに結合した各セルの左上にある展開アイコンまたは折りたたみアイコン [+/-] を使用して、結合したセルに対応する行を折りたたんだり、折りたたんだ行を再展開したりします。

図5.3 結合したセルの折りたたみ



5. すべてのルールにテンプレートキーデータを定義し、必要に応じてテーブル表示を変更したら、ガイド付きルールテンプレートデザイナーの右上のツールバーで **Validate** をクリックして、ガイド付きルールテンプレートの妥当性を確認します。ルールテンプレートの妥当性確認に失敗したら、エラーメッセージに記載された問題に対応し、ルールテンプレートの全コンポーネントと、データテーブルに定義したデータを見直し、エラーが表示されなくなるまでルールテンプレートの妥当性確認を行います。

6. ガイド付きルールテンプレートデザイナーで **Save** をクリックして、設定した内容を保存します。

第6章 ルールの実行

ルールの例を特定するか、Business Central でルールを作成したら、関連のプロジェクトをビルドしてデプロイし、ローカルまたは Process Server でルールを実行してルールをテストできます。

前提条件

- Business Central および Process Server がインストールされ、実行されている。インストールオプションは『Red Hat Process Automation Manager インストールの計画』を参照してください。

手順

1. Business Central で、**Menu → Design → Projects** に移動して、プロジェクト名をクリックします。
2. プロジェクトの **Assets** ページの右上にある **Deploy** をクリックして、プロジェクトをビルドして Process Server にデプロイします。ビルドに失敗したら、画面下部の **Alerts** パネルに記載されている問題に対処します。
   プロジェクトデプロイメントオプションに関する詳細は、『Red Hat Process Automation Manager プロジェクトのパッケージおよびデプロイ』を参照します。
3. ローカルでのルール実行に使用するか、Business Server でルールを実行するクライアントアプリケーションとして使用できるように、まだ作成されていない場合には、Process Central 外に Maven または Java プロジェクトを作成します。プロジェクトには、**pom.xml** ファイルと、プロジェクトリソースの実行に必要なその他のコンポーネントを含める必要があります。
   テストプロジェクトの例については、「その他の DRL ルールの作成および実行方法」を参照してください。
4. テストプロジェクトまたはクライアントアプリケーションの **pom.xml** ファイルを開き、以下の依存関係が追加されていない場合は追加します。

   - **kie-ci**: クライアントアプリケーションで、**ReleaseId** を使用して、Business Central プロジェクトデータをローカルにロードします。
   - **kie-server-client**: クライアントアプリケーションで、Process Server のアセットを使用してリモートに接続します。
   - **slf4j**: (オプション) クライアントアプリケーションで、Process Server に接続したあと、SLF4J (Simple Logging Facade for Java) を使用して、デバッグのログ情報を返します。

   クライアントアプリケーションの **pom.xml** ファイルにおける、Red Hat Process Automation Manager 7.4 の依存関係の例

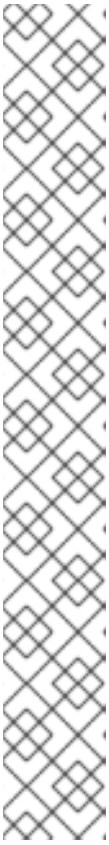
```
<!-- For local execution -->
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-ci</artifactId>
  <version>7.23.0.Final-redhat-00002</version>
</dependency>

<!-- For remote execution on Process Server -->
<dependency>
  <groupId>org.kie.server</groupId>
  <artifactId>kie-server-client</artifactId>
  <version>7.23.0.Final-redhat-00002</version>
```

```
</dependency>

<!-- For debug logging (optional) -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.25</version>
</dependency>
```

このアーティファクトで利用可能なバージョンについては、オンラインの Nexus Repository Manager でグループ ID とアーティファクト ID を検索してください。



注記

個別の依存関係に対して Red Hat Process Automation Manager **<version>** を指定するのではなく、Red Hat Business Automation 部品表 (BOM) の依存関係をプロジェクトの **pom.xml** ファイルに追加することを検討してください。Red Hat Business Automation BOM は、Red Hat Process Decision Manager と Red Hat Process Automation Manager の両方に適用します。BOM ファイルを追加すると、指定の Maven リポジトリーからの一時的な依存関係の内、正しいバージョンが、このプロジェクトに追加されます。

BOM 依存関係の例:

```
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.4.0.GA-redhat-00002</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

Red Hat Business Automation BOM (Bill of Materials) についての詳細情報は、「What is the mapping between Red Hat Process Automation Manager and the Maven library version?」を参照してください。

5. モジュールクラスを含むアーティファクトの依存関係が、クライアントアプリケーションの **pom.xml** ファイルに定義されていて、デプロイしたプロジェクトの **pom.xml** ファイルに記載されているのと同じであることを確認します。モデルクラスの依存関係が、クライアントアプリケーションとプロジェクトで異なると、実行エラーが発生します。
   Business Central でプロジェクトの **pom.xml** ファイルを利用するには、プロジェクトで既存のアセットを選択し、画面左側の **Project Explorer** メニューで **Customize View** ギアアイコンをクリックし、**Repository View → pom.xml** を選択します。

   たとえば、以下の **Person** クラスの依存関係は、クライアントと、デプロイしたプロジェクトの **pom.xml** ファイル両方に表示されます。

```
<dependency>
  <groupId>com.sample</groupId>
  <artifactId>Person</artifactId>
  <version>1.0.0</version>
</dependency>
```

6. デバッグ向けロギングを行うために、**slf4j** 依存関係を、クライアントアプリケーションの

pom.xml ファイルに追加した場合は、関連するクラスパス (Maven の **src/main/resources/META-INF** 内など) に **simplelogger.properties** ファイルを作成し、以下の内容を記載します。

```
org.slf4j.simpleLogger.defaultLogLevel=debug
```

7. クライアントアプリケーションに、必要なインポートを含む **.java** メインクラスと、KIE ベースをロードする **main()** メソッドを作成し、ファクトを挿入し、ルールを実行します。
たとえば、プロジェクトの **Person** オブジェクトには、名前、苗字、時給、賃金を設定および取得するゲッターメソッドおよびセッターメソッドが含まれます。プロジェクトにある以下の **Wage** ルールでは、賃金と時給を計算し、その結果に基づいてメッセージを表示します。

```
package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello" + " " + name + " " + surname + "!");
    System.out.println("You are rich!");
end
```

(必要に応じて) Process Server の外でローカルにこのルールをテストするには、**.java** クラスで、KIE サービス、KIE コンテナー、および KIE セッションをインポートするように設定し、その後、**main()** メソッドを使用して、定義したファクトモデルに対してすべてのルールを実行するようにします。

ローカルでルールの実行

```
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class RulesTest {

  public static final void main(String[] args) {
    try {
      // Identify the project in the local repository:
      ReleaseId rid = new ReleaseId();
      rid.setGroupId("com.myspace");
      rid.setArtifactId("MyProject");
      rid.setVersion("1.0.0");

      // Load the KIE base:
      KieServices ks = KieServices.Factory.get();
      KieContainer kContainer = ks.newKieContainer(rid);
      KieSession kSession = kContainer.newKieSession();

      // Set up the fact model:
      Person p = new Person();
```

```
        p.setWage(12);
        p.setFirstName("Tom");
        p.setLastName("Summers");
        p.setHourlyRate(10);

        // Insert the person into the session:
        kSession.insert(p);

        // Fire all rules:
        kSession.fireAllRules();
        kSession.dispose();
      }

      catch (Throwable t) {
        t.printStackTrace();
      }
    }
  }
```

Process Server でこのルールをテストするには、ローカル例と同じように、インポートとルール実行情報で **.java** クラスを設定し、KIE サービス設定および KIE サービスクライアントの詳細を指定します。

Process Server でルールの実行

```
package com.sample;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import org.kie.api.command.BatchExecutionCommand;
import org.kie.api.command.Command;
import org.kie.api.KieServices;
import org.kie.api.runtime.ExecutionResults;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.kie.server.api.marshalling.MarshallingFormat;
import org.kie.server.api.model.ServiceResponse;
import org.kie.server.client.KieServicesClient;
import org.kie.server.client.KieServicesConfiguration;
import org.kie.server.client.KieServicesFactory;
import org.kie.server.client.RuleServicesClient;

import com.sample.Person;

public class RulesTest {

  private static final String containerName = "testProject";
  private static final String sessionName = "myStatelessSession";

  public static final void main(String[] args) {
    try {
      // Define KIE services configuration and client:
      Set<Class<?>> allClasses = new HashSet<Class<?>>();
```

```
      allClasses.add(Person.class);
      String serverUrl = "http://$HOST:$PORT/kie-server/services/rest/server";
      String username = "$USERNAME";
      String password = "$PASSWORD";
      KieServicesConfiguration config =
        KieServicesFactory.newRestConfiguration(serverUrl,
                                        username,
                                        password);
      config.setMarshallingFormat(MarshallingFormat.JAXB);
      config.addExtraClasses(allClasses);
      KieServicesClient kieServicesClient =
        KieServicesFactory.newKieServicesClient(config);

      // Set up the fact model:
      Person p = new Person();
      p.setWage(12);
      p.setFirstName("Tom");
      p.setLastName("Summers");
      p.setHourlyRate(10);

      // Insert Person into the session:
      KieCommands kieCommands = KieServices.Factory.get().getCommands();
      List<Command> commandList = new ArrayList<Command>();
      commandList.add(kieCommands.newInsert(p, "personReturnId"));

      // Fire all rules:
      commandList.add(kieCommands.newFireAllRules("numberOfFiredRules"));
      BatchExecutionCommand batch = kieCommands.newBatchExecution(commandList,
sessionName);

      // Use rule services client to send request:
      RuleServicesClient ruleClient =
kieServicesClient.getServicesClient(RuleServicesClient.class);
      ServiceResponse<ExecutionResults> executeResponse =
ruleClient.executeCommandsWithResults(containerName, batch);
      System.out.println("number of fired rules:" +
executeResponse.getResult().getValue("numberOfFiredRules"));
    }

    catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

8. 設定した **.java** クラスをプロジェクトディレクトリーから実行します。(Red Hat JBoss Developer Studio などの) 開発プラットフォーム、またはコマンドラインでファイルを実行できます。
(プロジェクトディレクトリーにおける) Maven の実行例:

```
mvn clean install exec:java -Dexec.mainClass="com.sample.app.RulesTest"
```

(プロジェクトディレクトリーにおける) Java の実行例

```
javac -classpath "./$DEPENDENCIES/*:." RulesTest.java
java -classpath "./$DEPENDENCIES/*:." RulesTest
```

9. コマンドラインおよびサーバーログで、ルール実行のステータスを確認します。ルールが想定通りに実行されない場合は、プロジェクトに設定したルールと、メインのクラス設定を確認して、指定したデータの妥当性を確認します。

6.1. 実行可能ルールモデル

実行可能ルールモデルは埋め込み可能なモデルで、ビルド時に実行するルールセットの Java ベース表記を提供します。実行可能モデルは Red Hat Process Automation Manager の標準アセットパッケージングの代わりとなるもので、より効率的です。KIE コンテナーと KIE ベースの作成がより迅速にでき、DRL (Drools Rule Language) ファイルリストや他の Red Hat Process Automation Manager アセットが多い場合は、特に有効です。このモデルは詳細レベルにわたり、インデックス評価の lambda 表記など、必要な実行情報すべてを提供できます。

実行可能なルールモデルでは、プロジェクトにとって具体的に以下のような利点があります。

- **コンパイル時間**: 従来のパッケージ化された Red Hat Process Automation Manager プロジェクト (KJAR) には、制限や結果を実装する事前生成済みのクラスと合わせて、ルールベースを定義する DRL ファイルのリストやその他の Red Hat Process Automation Manager アーティファクトが含まれています。これらの DRL ファイルは、KJAR が Maven リポジトリーからダウンロードされて、KIE コンテナーにインストールされた時点で、解析してコンパイルする必要があります。特に大規模なルールセットの場合など、このプロセスは時間がかかる可能性があります。実行可能なモデルでは、プロジェクト KJAR 内で、Java クラスをパッケージして、プロジェクトルールベースの実行可能なモデルを実装し、はるかに早い方法で KIE コンテナーと KIE ベースをを再作成することができます。Maven プロジェクトでは、**kie-maven-plugin** を使用してコンパイルプロセス中に DRL ファイルから 実行可能なモデルソースを自動的に生成します。

- **ランタイム**: 実行可能なモデルでは、制約はすべて、Java lambda 式で定義されます。同じ lambda 式も制約評価に使用するので、**mvel** ベースの制約をバイトコードに変換するのに、解釈評価用の **mvel** 式も、Just-In-Time (JIT) プロセスも使用しません。これにより、さらに迅速で効率的なランタイムを構築できます。

- **開発時間**: 実行可能なモデルでは、DRL 形式で直接要素をエンコードしたり、DRL パーサーを対応するように変更したりする必要なく、デシジョンエンジンの新機能で開発および試行することができます。

6.1.1. Maven プロジェクトへの実行可能なルールモデルの埋め込み

Maven プロジェクトに実行可能ルールモデルを埋め込み、ビルド時にルールアセットをより効率的にコンパイルすることができます。

前提条件

- Red Hat Process Automation Manager ビジネスアセットを含む Maven 化したプロジェクトがある。

手順

1. Maven プロジェクトの **pom.xml** ファイルで、パッケージタイプを **kjar** に設定し、**kie-maven-plugin** ビルドコンポーネントを追加します。

```
<packaging>kjar</packaging>
...
<build>
  <plugins>
    <plugin>
      <groupId>org.kie</groupId>
      <artifactId>kie-maven-plugin</artifactId>
      <version>${rhpam.version}</version>
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>
```

kjar パッケージングタイプは、**kie-maven-plugin** コンポーネントをアクティブにして、アーティファクトリソースを検証してプリコンパイルします。**<version>** は、プロジェクトで現在使用される Red Hat Process Automation Manager の Maven アーティファクトのバージョン (例: 7.23.0.Final-redhat-00002) で、デプロイメントに Maven プロジェクトを適切にパッケージがするのに必要です。

注記

個別の依存関係に対して Red Hat Process Automation Manager **<version>** を指定するのではなく、Red Hat Business Automation 部品表 (BOM) の依存関係をプロジェクトの **pom.xml** ファイルに追加することを検討してください。Red Hat Business Automation BOM は、Red Hat Process Decision Manager と Red Hat Process Automation Manager の両方に適用します。BOM ファイルを追加すると、指定の Maven リポジトリーからの一時的な依存関係の内、正しいバージョンが、このプロジェクトに追加されます。

BOM 依存関係の例:

```
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.4.0.GA-redhat-00002</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

Red Hat Business Automation BOM (Bill of Materials) についての詳細情報は、What is the mapping between RHPAM product and maven library version? を参照してください。

2. 以下の依存関係を **pom.xml** ファイルに追加して、ルールアセットが実行可能なモデルからビルドできるようにします。

- **drools-canonical-model**: Red Hat Process Automation Manager から独立するルールセットモデルの実行可能な正規表現を有効にします。
- **drools-model-compiler**: デシジョンエンジンで実行できるように Red Hat Process Automation Manager の内部データ構造に実行可能なモデルをコンパイルします。

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-canonical-model</artifactId>
```

```
  <version>${rhpam.version}</version>
</dependency>

<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-model-compiler</artifactId>
  <version>${rhpam.version}</version>
</dependency>
```

3. コマンドターミナルで Maven プロジェクトディレクトリーに移動して、以下のコマンドを実行し、実行可能なモデルからプロジェクトをビルドします。

```
mvn clean install -DgenerateModel=<VALUE>
```

-DgenerateModel=<VALUE> プロパティーで、プロジェクトが DRL ベースの KJAR ではなく、モデルベースの KJAR としてビルドできるようにします。

<VALUE> は、3 つの値のいずれかに置き換えます。

- **YES**: オリジナルプロジェクトの DRL ファイルに対応する実行可能なモデルを生成し、生成した KJAR から DRL ファイルを除外します。
- **WITHDRL**: オリジナルプロジェクトの DRL ファイルに対応する実行可能なモデルを生成し、文書化の目的で、生成した KJAR に DRL ファイルを追加します (KIE ベースはいずれの場合でも実行可能なモデルからビルドされます)。
- **NO**: 実行可能なモデルは生成されません。

ビルドコマンドの例:

```
mvn clean install -DgenerateModel=YES
```

Maven プロジェクトのパッケージ化に関する詳細は、『Red Hat Process Automation Manager プロジェクトのパッケージ化およびデプロイ』を参照してください。

6.1.2. Java アプリケーションページへの実行可能なルールモデルの埋め込み

Java アプリケーションに実行可能ルールモデルをプログラミングを使用して埋め込み、ビルド時にルールアセットをより効率的にコンパイルすることができます。

前提条件

- Red Hat Process Automation Manager ビジネスアセットを含む Java アプリケーションがあること

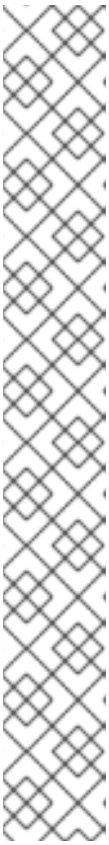
手順

1. Java プロジェクトの適切なクラスパスに、以下の依存関係を追加します。
   - **drools-canonical-model**: Red Hat Process Automation Manager から独立するルールセットモデルの実行可能な正規表現を有効にします。
   - **drools-model-compiler**: デシジョンエンジンで実行できるように Red Hat Process Automation Manager の内部データ構造に実行可能なモデルをコンパイルします。

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-canonical-model</artifactId>
  <version>${rhpam.version}</version>
</dependency>

<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-model-compiler</artifactId>
  <version>${rhpam.version}</version>
</dependency>
```

<version> は、プロジェクトで現在使用する Red Hat Process Automation Manager の Maven アーティファクトバージョンです (例: 7.23.0.Final-redhat-00002)。



注記

個別の依存関係に対して Red Hat Process Automation Manager **<version>** を指定するのではなく、Red Hat Business Automation 部品表 (BOM) の依存関係をプロジェクトの **pom.xml** ファイルに追加することを検討してください。Red Hat Business Automation BOM は、Red Hat Process Decision Manager と Red Hat Process Automation Manager の両方に適用します。BOM ファイルを追加すると、指定の Maven リポジトリーからの一時的な依存関係の内、正しいバージョンが、このプロジェクトに追加されます。

BOM 依存関係の例:

```
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.4.0.GA-redhat-00002</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

Red Hat Business Automation BOM (Bill of Materials) についての詳細情報は、What is the mapping between RHPAM product and maven library version? を参照してください。

2. ルールアセットを KIE 仮想ファイルシステム **KieFileSystem** に追加して、**KieBuilder** に **buildAll(ExecutableModelProject.class)** を指定して使用し、実行可能なモデルからアセットをビルドします。

```
import org.kie.api.KieServices;
import org.kie.api.builder.KieFileSystem;
import org.kie.api.builder.KieBuilder;

  KieServices ks = KieServices.Factory.get();
  KieFileSystem kfs = ks.newKieFileSystem()
  kfs.write("src/main/resources/KBase1/ruleSet1.drl", stringContainingAValidDRL)
  .write("src/main/resources/dtable.xls",
    kieServices.getResources().newInputStreamResource(dtableFileStream));

  KieBuilder kieBuilder = ks.newKieBuilder( kfs );
```

```
// Build from an executable model
kieBuilder.buildAll( ExecutableModelProject.class )
assertEquals(0, kieBuilder.getResults().getMessages(Message.Level.ERROR).size());
```

実行可能なモデルから **KieFileSystem** をビルドした後に、作成された **KieSession** は効率のあまりよくない **mvel** 式ではなく、lambda 式をもとにした制約を使用します。**buildAll()** に引数が含まれていない場合には、プロジェクトは実行可能なモデルのない標準の手法でビルドされます。

KieFileSystem を使用する代わりに、手作業を多く使用して実行可能なモデルを作成する別の方法として、Fluent API で **Model** を定義して、そこから **KieBase** を作成することができます。

```
Model model = new ModelImpl().addRule( rule );
KieBase kieBase = KieBaseBuilder.createKieBaseFromModel( model );
```

Java アプリケーション内でプロジェクトをプログラミングを使用してパッケージ化する方法については、『**Red Hat Process Automation Manager プロジェクトのパッケージ化およびデプロイ**』を参照してください。

第7章 次のステップ

- 『テストシナリオを使用したデシジョンサービスのテスト』
- 『Red Hat Process Automation Manager プロジェクトのパッケージ化およびデプロイ』

付録A バージョン情報

本書の最終更新日: 2019 年 8 月 12 日 (月)