Red Hat Decision Manager 7.1

DRL ルールを使用したデシジョンサービスの作成

Last Updated: 2019-02-06

Red Hat Decision Manager 7.1 DRL ルール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 Decision Manager 7.1 で、DRL ルール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

ビジネスルール開発者は、Decision Central で DRL (Drools Rule Language) デザイナーを使用してビジネスルールを定義できます。DRL ルールは、Decision Central におけるその他のルールアセットとは異なり、ガイド付きまたは表形式ではなく、フリーフォームの `.drl` テキストファイルで直接定義できます。このような DRL ファイルは、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

DRL ルールのチームおよびプロジェクトが Decision Central に作成されており、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ます。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 DECISION MANAGER におけるルール作成アセット

Red Hat Decision Manager は、デシジョンサービスにビジネスルールを作成するのに使用するアセットを提供します。ルール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適したアセットを 1 つ、または複数を組み合わせて使用できます。

デシジョンサービスでルールを作成する最適な方法を選択できるように、以下の表で、Decision Central のルール作成アセットを紹介します。

表1.1 Decision Central におけるルール作成アセット

アセット	主な特徴	ドキュメンテーション
ガイド付きデシジョンテーブル	• Decision 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のスプレッドシートをアップロードする代わりに、ウィザードを用いて作成する • 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	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
アップロードしたデシジョンテーブル	• Decision Central にアップロードした XLS または XLSX 形式のデシジョンテーブルスプレッドシート • 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 • Decision 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	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

アセット	主な特徴	ドキュメンテーション
ガイド付きルール	• Decision 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	**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**
ガイド付きルールテンプレート	• Decision 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	**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 • ルールと、ルール動作に関するその他の技術を定義する柔軟性が最も高い • スタンドアロン環境で作成し、Red Hat Decision 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しい構文要件がある	**DRL ルールを使用したデシジョンサービスの作成**

第2章 DRL (DROOLS RULE LANGUAGE) ルール言語

DRL (Drools Rule Language) ルールは、**.drl** テキストファイルに直接定義するビジネスルールです。このような DRL ファイルは、Decision Central の他のすべてのルールアセットが最終的にレンダリングされるソースとなります。DRL ファイルを Decision Central インターフェースで作成および管理したり、外部の Red Hat Developer Studio、Java オブジェクト、Maven アーキタイプを使用して作成したりできます。DRL ファイルには、最低限、ルールの条件 (**when**) およびアクション (**then**) を定義するルールを 1 つ以上追加できます。Decision Central の DRL デザイナーでは、Java、DRL、および XML の構文が強調表示されます。

DRL ルールに関連するデータオブジェクトはすべて、DRL ルールと同じ Decision Central プロジェクトパッケージに置く必要があります。同じパッケージのアセットはデフォルトでインポートされます。その他のパッケージの既存アセットは、DRL ルールを使用してインポートできます。

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、特定のビジネスプロセスに固有のものではありません。

手順

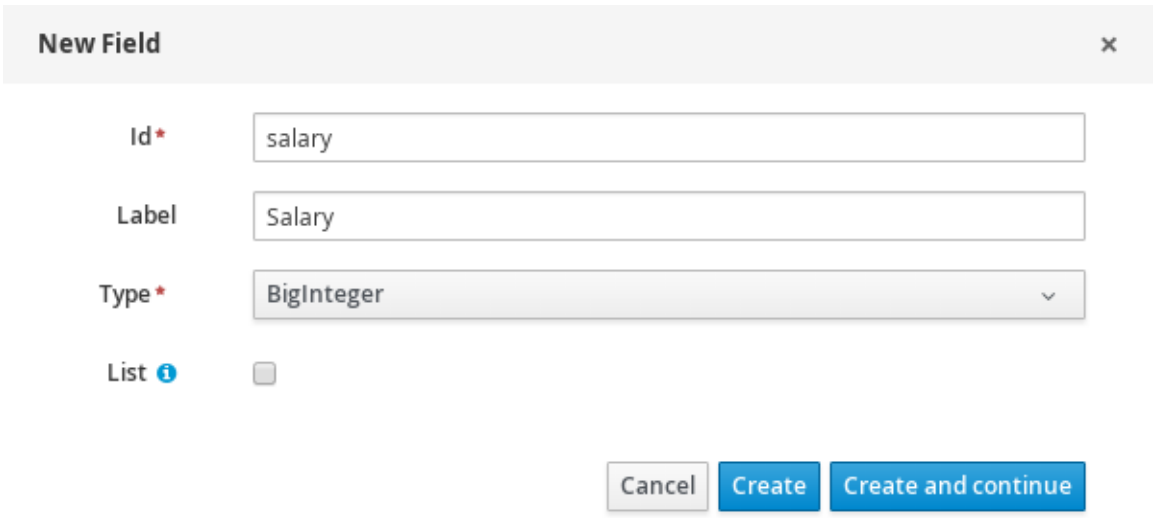
1. Decision Central で、**Menu** → **Design** → **Projects** に移動して、プロジェクト名をクリックします。
2. **Add Asset** → **Data Object** をクリックします。
3. 一意の **データオブジェクト** 名を入力し、**パッケージ** を選択します。これにより、その他のルールアセットでもデータオブジェクトを利用できるようになります。同じパッケージに、同じ名前のデータオブジェクトを複数作成することはできません。指定するパッケージは、そのデータオブジェクトを必要とするルールアセットが割り当てられている、もしく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。

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、ルールアセットのパッケージに、既存のデータオブジェクトをインポートすることもできます。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:** 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 DECISION CENTRAL における DRL ルールの作成

Decision Central で、プロジェクトに対して DRL ルールを作成して管理できます。パッケージに作成またはインポートするデータオブジェクトに基づいて、各 DRL ファイルで、ルールの条件、アクション、そしてルールに関連するその他のコンポーネントを定義します。

手順

1. Decision 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** → **DRL ファイル** をクリックします。
3. 参考となる **DRL ファイル** 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。
   ドメイン固有言語 (DSL) アセットがプロジェクトに定義されている場合は、**Show declared DSL sentences** を選択することもできます。この DSL アセットは、DRL デザイナーで定義する条件およびアクションに使用できるオブジェクトです。
4. **OK** をクリックして、ルールアセットを作成します。
   新しい DRL ファイルが、**Project Explorer** の **DRL** パネルに追加されます。**Show declared DSL sentences** オプションを選択した場合は、**DSLR** パネルに追加されます。この DRL ファイルを割り当てたパッケージは、ファイルの上位にリストされます。
5. DRL デザイナーの左パネルの **Fact types** リストで、ルールに必要なすべてのデータオブジェクトとデータオブジェクトフィールドがリストされていることを確認します (それぞれを展開します)。リストされていない場合は、DRL ファイルの `import` 命令文を使用して、その他のパッケージから関連する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、DRL デザイナーの **Model** タブに戻り、以下のいずれかのコンポーネントで DRL ファイルを定義します。

   DRL ファイルのコンポーネント

   ```
   package  //automatic

   import

   function  //optional

   query  //optional

   declare   //optional

   rule

   rule

   ...
   ```

   - **package**: (自動) これは、DRL ファイルを作成し、パッケージを選択すると定義されます。

- **import**: このパッケージ、または DRL ファイルで使用するその他のパッケージのデータオブジェクトを指定します。パッケージとデータオブジェクトを **package.name.object.name** 形式で指定し、1 行につき 1 つインポートします。

  データオブジェクトのインポート

  ```
  import mortgages.mortgages.LoanApplication;
  ```

- **function**: (任意) DRL ファイルのルールが使用する関数を指定します。関数は、ルールのソースファイルにセマンティックコードを追加します。関数は、特に、ルールのアクション (**then**) 部分が繰り返し使用され、パラメーターだけがルールごとに異なる場合に便利です。DRL ファイルのルールで、関数を宣言したり、静的メソッドを関数としてインポートしたりして、ルールの アクション (**then**) 部分に、名前を指定して関数を使用します。

  ルールに関数を宣言して使用 (オプション 1)

  ```
  function String hello(String applicantName) {
      return "Hello " + applicantName + "!";
  }

  rule "Using a function"
    when
      eval( true )
    then
      System.out.println( hello( "James" ) );
  end
  ```

  ルールに関数をインポートして使用 (オプション 2)

  ```
  import function my.package.applicant.hello;

  rule "Using a function"
    when
      eval( true )
    then
      System.out.println( hello( "James" ) );
  end
  ```

- **query**: (任意) DRL ファイルのルールに関連するファクトに対してデシジョンエンジンを検索するのに使用します。クエリーは、定義した条件セットを検索するため、**when** または **then** を指定する必要はありません。クエリー名は KIE ベースでグローバルとなるため、プロジェクトにあるその他のすべてのルールクエリーと重複しないようにする必要があります。クエリーの結果に戻るには、**ksession.getQueryResults("name")** を使用して、従来の **QueryResults** 定義を構成します (**"name"** はクエリー名)。これにより、クエリーの結果が返り、クエリーに一致したオブジェクトを取得できるようになります。DRL ファイルのルールに、クエリーと、クエリー結果パラメーターを定義します。

  ルールで、年齢が 21 歳未満の場合のクエリーと、その結果

  ```
  query "people under the age of 21"
      person : Person( age < 21 )
  end

  QueryResults results = ksession.getQueryResults( "people under
  ```

```
the age of 21" );
System.out.println( "we have " + results.size() + " people under
the age  of 21" );

rule "Underage"
  when
    application : LoanApplication( )
    Applicant( age < 21 )
  then
    application.setApproved( false );
    application.setExplanation( "Underage" );
end
```

- **declare**: (任意) DRL ファイルのルールが使用する新しいファクトタイプを宣言します。Red Hat Decision Manager の **java.lang** パッケージのデフォルトは **Object** ですが、必要に応じて DRL ファイルに別のタイプを宣言することもできます。DRL ファイルにファクトタイプを宣言すると、Java などの低級言語でモデルを作成せず、デシジョンエンジンに直接新しいファクトモデルを定義するようになります。

  新しいファクトタイプの宣言および使用

```
declare Person
  name : String
  dateOfBirth : java.util.Date
  address : Address
end

rule "Using a declared type"
  when
    $p : Person( name == "James" )
  then   // Insert Mark, who is a customer of James.
    Person mark = new Person();
    mark.setName("Mark");
    insert( mark );
end
```

- **rule**: DRL ファイルで各ルールを定義します。ルールは、**rule "name"** 形式のルール名、ルールの動作 (**salience**、**no-loop** など) を定義する任意の属性、**when** および **then** 定義が続きます。同じパッケージにルール名を重複させることはできません。ルールの **when** 部分には、アクションを実行する条件が含まれます。たとえば、銀行が、ローンの申し込みを 21 歳以上に限定した場合、**Underage** ルールの **when** 条件は **Applicant(age < 21)** になります。ルールの **then** 部分には、ルールの条件が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、ローンの申込者が 21 歳に満たない場合、**then** アクションは **setApproved(false)** になり、申込者の年齢条件を満たしていないためにローンの申し込みは承認されません。条件 (**when**) およびアクション (**then**) は、パッケージで利用可能なデータオブジェクトに基づいて、任意の制約、バインディングなどのサポートされる DRL 要素を持つ、定められている一連のファクトパターンで構成されます。このパターンは、定義したオブジェクトにルールがどのように影響するかを指定します。

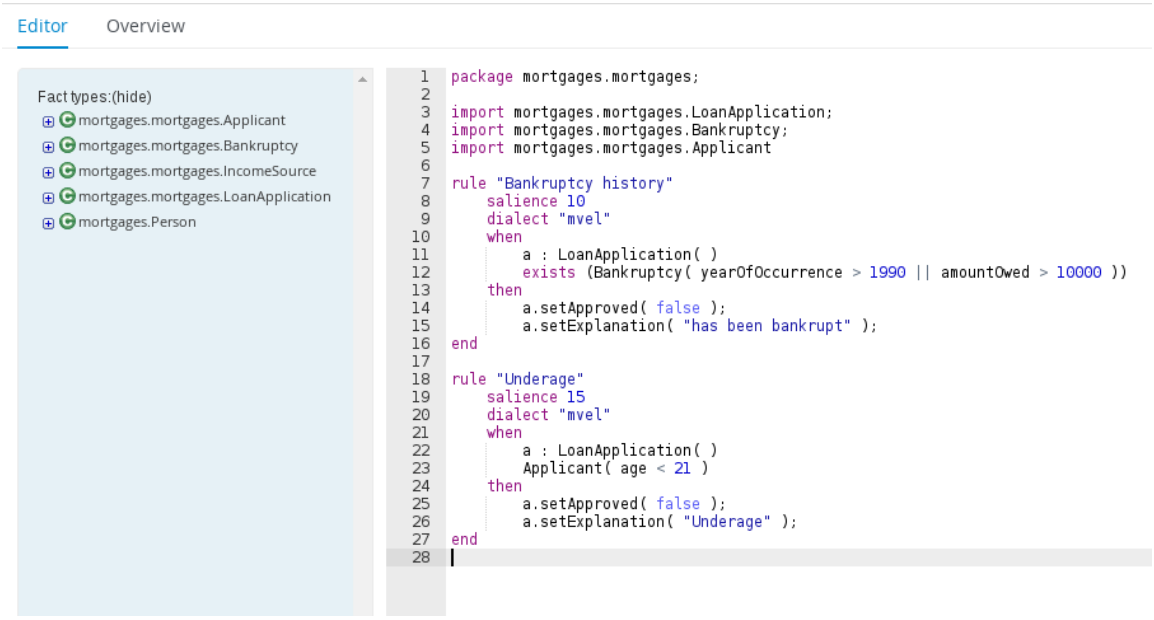
  申込者の年齢制限に関するルール

```
rule "Underage"
  salience 15
  dialect "mvel"
```

```
  when
    application : LoanApplication( )
    Applicant( age < 21 )
  then
    application.setApproved( false );
    application.setExplanation( "Underage" );
end
```

少なくても、各 DRL ファイルに **package** コンポーネント、**import** コンポーネント、**rule** コンポーネントを指定する必要があります。他のすべてのコンポーネントは任意です。

図4.1 必要なコンポーネントおよび任意のルール属性を持つ DRL ファイルのサンプル



7. ルールのコンポーネントをすべて定義したら、DRL デザイナーの右上ツールバーで **Validate** をクリックし、DRL ファイルの妥当性を確認します。ファイル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、DRL ファイルの構文およびコンポーネントを見直し、エラーが表示されなくなるまで妥当性確認を行います。

8. DRL 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。

DRL ルールに条件を追加する方法は 「DRL ルールへの WHEN 条件の追加」 を参照してください。

DRL ルールにアクションを追加する方法は 「DRL ルールへの THEN アクションの追加」 を参照してください。

4.1. DRL ルールへの WHEN 条件の追加

ルールの **when** 部分には、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、銀行のローン申し込みに年齢制限 (21 歳以上) が必要な場合、**Underage** ルールの **when** 条件は **`Applicant( age < 21 )`** となります。パッケージで利用可能なデータオブジェクトに基づいて、指定した一連のパターンおよび制約と、任意のバインディング、その他のサポートされる DRL 要素で構成されます。

前提条件

- **package** は DRL ファイルに定義されます。これは、ファイルの作成時に行われます。

- ルールで使用したデータオブジェクトの **import** リストが、DRL ファイルの **package** 行の下に定義されます。データオブジェクトは、このパッケージ、または別の Decision Central パッケージから使用できます。
- **rule** 名は、**package**、**import**、または DRL ファイル全体に適用されるその他の行の下に、**rule "name"** という形式で定義されます。同じパッケージでルール名を重複させることはできません。ルールの動作 (**salience**、**no-loop** など) を定義する任意のルール属性は、ルール名の下、**when** セクションの前に定義します。

手順

1. DRL デザイナーで、ルールに **when** を入力して、条件命令文を追加します。**when** セクションは、ルールの条件を定義するファクトパターンで構成されますが、ファクトパターンが 1 つも追加されない場合もあります。
when セクションを空にすると、デシジョンエンジンで **fireAllRules()** を呼び出すたびに、**then** セクションのアクションが実行します。これは、デシジョンエンジンのステートを設定するルールを使用する場合に便利です。

   条件のないルール

```
rule "bootstrap"
  when   // empty

  then   // actions to be executed once
    insert( new Applicant() );
end

// The above rule is internally rewritten as:

rule "bootstrap"
  when
    eval( true )
  then
    insert( new Applicant() );
end
```

2. 一致させる最初の条件のパターンを入力し、任意で制約、バインディング、およびサポートされる DRL 要素を入力します。基本的なパターンフォーマットは **patternBinding : patternType (constraints)** です。パターンは、パッケージで利用可能なデータオブジェクトに基づいており、**then** セクションのアクションを発生させるのに必要な条件を定義します。
   - **単純なパターン:** 制約のない単純なパターンは、指定したタイプのファクトに一致します。たとえば、次は、申込者が存在することだけが条件になります。

```
when
  Applicant( )
```

   - **制約のあるパターン:** 制約を持つパターンは、指定したタイプのファクトと、追加制限を括弧で指定したパターン (true または false) に一致します。たとえば、次は、申込者が 21 歳に満たないことを条件としています。

```
when
  Applicant( age < 21 )
```

- **Pattern with binding:** パターンのバインディングは簡単な参照となり、ルールのその他のコンポーネントが、定義したパターンに戻って参照します。たとえば、次の例で、**LoanApplication** のバインディング **a** が、underage の申込者に関連するアクションとして使用されます。

```
when
  a : LoanApplication( )
  Applicant( age < 21 )
then
  a.setApproved( false );
  a.setExplanation( "Underage" )
```

3. 引き続き、このルールに適用する条件パターンをすべて定義します。以下は、DRL 条件を定義するいくつかのキーワードオプションです。

- **and**: 条件コンポーネントを論理積に分類します。接中辞および接頭辞の **and** がサポートされます。デフォルトでは、結合演算子を指定しないと、リストされている条件またはアクションがすべて **and** と結合します。

```
a : LoanApplication( ) and Applicant( age < 21 )

a : LoanApplication( )
and Applicant( age < 21 )

a : LoanApplication( )
Applicant( age < 21 )

// All of the above are the same.
```

- **or**: 条件コンポーネントを論理和に分類します。接中辞および接頭辞の **or** がサポートされます。

```
Bankruptcy( amountOwed == 100000 ) or IncomeSource( amount ==
20000 )

Bankruptcy( amountOwed == 100000 )
or IncomeSource( amount == 20000 )
```

- **exists**: 存在すべきファクトおよび制約を指定します。これは、ファクトが存在していることを示しているのではなく、ファクトが存在すべきであることを示しています。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。

```
exists (Bankruptcy( yearOfOccurrence > 1990 || amountOwed > 10000
))
```

- **not**: 存在するべきでないファクトおよび制約を指定します。

```
not (Applicant( age < 21 ))
```

- **forall**: 最初のパターンに一致したすべてのファクトが残りのパターンに一致する制約を作成します。

```
forall( app : Applicant( age < 21 )
              Applicant( this == app, status = 'underage' ) )
```

- **from**: 条件パターンによりデータが一致するソースを指定します。

```
Applicant( ApplicantAddress : address )
Address( zipcode == "23920W" ) from ApplicantAddress
```

- **entry-point**: パターンのデータソースに対応するエントリーポイントを定義します。通常は **from** とともに使用します。

```
Applicant( ) from entry-point "LoanApplication"
```

- **collect**: 構成を条件の一部として使用できる、オブジェクトのコレクションを定義します。この例では、指定した各担保に対して、デシジョンエンジンで保留されているすべての申し込みが **ArrayLists** に分類されます。申し込みが 3 つ以上ある場合は、このルールが実行します。

```
m : Mortgage()
a : ArrayList( size >= 3 )
    from collect( LoanApplication( Mortgage == m, status ==
'pending' ) )
```

- **accumulate**: オブジェクトのコレクションを処理し、各要素のカスタムアクションを実行し、(制約が **true** と評価されると) 結果オブジェクトを 1 つ以上返します。このオプションは、**collect** よりも強力で、柔軟性が高いオプションです。**accumulate(<source pattern>; <functions> [;<constraints>])** 形式を使用します。この例では、**min**、**max**、および **average** は累積関数で、各センサーのすべての測定値から、最低気温、最高気温、そして平均気温の値を計算します。その他のサポートされる関数には、**count**、**sum**、**variance**、**standardDeviation**、**collectList**、および **collectSet** があります。

```
s : Sensor()
accumulate( Reading( sensor == s, temp : temperature );
            min : min( temp ),
            max : max( temp ),
            avg : average( temp );
            min < 20, avg > 70 )
```

高度な DRL オプション

これは、条件を定義する基本的なキーワードオプションおよびパターン構築の例です。さらに高度な DRL オプションと構文が DRL デザイナーでサポートされています。オンラインの『Drools ドキュメンテーション』を参照してください。

4. ルールの条件コンポーネントをすべて定義したら、DRL デザイナーの右上のツールバーの **Validate** をクリックして、DRL ファイルの妥当性を確認します。ファイル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、DRL ファイルの構文およびコンポーネントを見直し、エラーが表示されなくなるまで妥当性確認を行います。

5. DRL 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. DRL ルールへの THEN アクションの追加

ルールの **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、ローンの申込者が 21 歳に満たない場合は、**Underage** ルールの **then** アクションが **setApproved(false)** となり、年齢が基準に達していないためローンの申し込みが承認されません。アクションは、ルールの条件と、パッケージで利用可能オブジェクトに基づいて結果を実行します。

前提条件

- **package** は DRL ファイルに定義されます。これは、ファイルの作成時に行われます。
- ルールで使用したデータオブジェクトの **import** リストが、DRL ファイルの **package** 行の下に定義されます。データオブジェクトは、このパッケージ、または別の Decision Central パッケージから使用できます。
- **rule** 名は、**package**、**import**、または DRL ファイル全体に適用されるその他の行の下に、**rule "name"** という形式で定義されます。同じパッケージでルール名を重複させることはできません。ルールの動作 (**salience**、**no-loop** など) を定義する任意のルール属性は、ルール名の下、**when** セクションの前に定義します。

手順

1. DRL デザイナーで、ルールの **when** セクションの後に **then** を入力して、アクション命令文を追加します。
2. ルールの条件に基づいて、ファクトパターンに対して実行するアクションを 1 つ以上入力します。
   次は、DRL アクションを定義するキーワードオプションの例です。
   - **and**: アクションコンポーネントを論理積に分類します。接中辞および接頭辞の **and** がサポートされます。デフォルトでは、結合演算子を指定しないと、リストされている条件またはアクションがすべて **and** と結合します。

     ```
     application.setApproved ( false ) and application.setExplanation(
     "has been bankrupt" );

     application.setApproved ( false );
     and application.setExplanation( "has been bankrupt" );

     application.setApproved ( false );
     application.setExplanation( "has been bankrupt" );

     // All of the above are the same.
     ```

   - **set**: フィールドの値を設定します。

     ```
     application.setApproved ( false );
     application.setExplanation( "has been bankrupt" );
     ```

   - **modify**: ファクトに対して修正するフィールドを指定し、変更をデシジョンエンジンに通知します。

     ```
     modify( LoanApplication ) {
             setAmount( 100 )
     }
     ```

- **update**: フィールドと、修正される関連ファクト全体を指定して、その変更をデシジョンエンジンに通知します。ファクトが変更したら、アップデートした値の影響を受ける可能性がある別のファクトを変更する前に、**update** を呼び出す必要があります。**modify** キーワードには、この追加手順がありません。

```
update( LoanApplication ) {
        setAmount( 100 )
}
```

- **delete**: デシジョンエンジンからオブジェクトを削除します。キーワード **retract** も DRL デザイナーでサポートされ、同じアクションを実行しますが、キーワード **insert** との一貫性を保つために **delete** が好まれます。

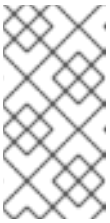
```
delete( LoanApplication );
```

- **insert**: **新しい** ファクトを挿入し、ファクトに必要な結果フィールドと値を定義します。

```
insert( new Applicant() );
```

- **insertLogical**: **新しい** ファクトをデシジョンエンジンに論理的に挿入し、ファクトに必要な結果フィールドと値を追加します。Red Hat Decision Manager の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に取り消されます。

```
insertLogical( new Applicant() );
```



高度な DRL オプション

これは、アクションを定義する基本的なキーワードオプションおよびパターン構築の例です。さらに高度な DRL オプションと構文が DRL デザイナーでサポートされています。オンラインの『Drools ドキュメンテーション』を参照してください。

3. ルールのアクションコンポーネントをすべて定義したら、DRL デザイナーの右上のツールバーの **Validate** をクリックして、DRL ファイルの妥当性を確認します。ファイル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、DRL ファイルの構文およびコンポーネントを見直し、エラーが表示されなくなるまで妥当性確認を行います。

4. DRL 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。

4.2.1. 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

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"**
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は、ルールを 1 つだけアクティブにできます。発生する最初のルールが、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されているルールをすべてキャンセルします。 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)

属性	値
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った場合に、そのルールが割り当てられたアジェンダグループにフォーカスが自動的に指定されます。 例: **auto-focus true**
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 ルールの実行

Decision Central でルールを作成したら、プロジェクトをビルドしてデプロイし、ローカルまたは Decision Server でルールを実行してテストできます。

手順

1. Decision 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. 右上にある **Build** をクリックしてから **Deploy** をクリックして、プロジェクトをビルドして Decision Server にデプロイします。ビルドに失敗したら、画面下部の **Alerts** パネルに説明されている問題に対処します。プロジェクトをデプロイする方法は**「Red Hat Decision Manager プロジェクトのパッケージングおよびデプロイメント」**を参照してください。
3. 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、Decision 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、Decision 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、Decision 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 Decision Manager 7.1 の依存関係の例

```
// For local execution:
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-ci</artifactId>
  <version>7.11.0.Final-redhat-00002</version>
</dependency>

// For remote execution on Decision Server:
<dependency>
  <groupId>org.kie.server</groupId>
  <artifactId>kie-server-client</artifactId>
  <version>7.11.0.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 Decision 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 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>7.1.0.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 Decision Manager and the Maven library version? を参照してください。

4. 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。
   Decision Central でプロジェクト **pom.xml** を表示するには、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、以下はクライアントと、デプロイしたプロジェクトの両方の **pom.xml** ファイルに表示されるため、**Person** クラスの依存関係です。

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

5. 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
   ```

6. 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) Decision 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。

ローカルでのルールの実行

```
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();
    }
  }
}
```

Decision 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。

Decision Server でのルールの実行

```
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();
    }
  }
}
```

7. 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
```

8. 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。ルールが期待通りに実行しない場合は、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、提供されるデータの妥当性を確認します。

第6章 その他の DRL ルールの作成および実行方法

Decision Central インターフェースに DRL ルールを作成し管理する代わりに、Red Hat Developer Studio、Java オブジェクト、または Maven アーキタイプを使用して、外部のスタンドアロンプロジェクトに DRL ルールファイルを作成できます。このスタンドアロンプロジェクトは、ナレッジ JAR (kJAR) 依存関係として、Decision Central の既存の Red Hat Decision Manager プロジェクトに統合できます。スタンドアロンプロジェクトの DRL ファイルには、少なくても、必要な **`package`** 仕様、**`import`** リスト、および **`rule`** 定義が含まれる必要があります。グローバル変数や関数など、その他の DRL コンポーネントは任意です。DRL ルールに関連するすべてのデータオブジェクトは、スタンドアロンの DRL プロジェクトまたはデプロイメントに含まれる必要があります。

Maven または Java プロジェクトで実行可能なルールモデルを使用して、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 Decision 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 Decision Manager アセットが多い場合は、特に有効です。

6.1. RED HAT JBOSS DEVELOPER STUDIO への DRL ルールの作成および実行

Red Hat JBoss Developer Studio を使用して、ルールが含まれる DRL ファイルを作成し、Red Hat Decision Manager デシジョンサービスにファイルを統合します。DRL ルールを作成する方法は、デシジョンサービスに Red Hat Developer Studio を使用する場合や、同じワークフローを継続する場合に便利です。この方法を使用していない場合は、Red Hat Decision Manager の代わりに Decision Central インターフェースを使用して、DRL ファイルや、その他のルールアセットを作成することが推奨されます。

前提条件

Red Hat カスタマーポータル から Red Hat JBoss Developer Studio をインストールしています。

手順

1. Red Hat JBoss Developer Studio で、**File** → **New** → **Project** をクリックします。
2. 開いた **New Project** ウィンドウで、**Drools** → **Drools Project** を選択し、**Next** をクリックします。
3. **Create a project and populate it with some example files to help you get started quickly** の 2 番目のアイコンをクリックして、**Next** をクリックします。
4. **Project name** を入力し、プロジェクトのビルドオプションで **Maven** ラジオボタンを選択します。GAV 値が自動的に生成されます。必要に応じて、プロジェクトに対してこの値を更新できます。
   - **Group ID:** **`com.sample`**
   - **Artifact ID:** **`my-project`**
   - **Version:** **`1.0.0-SNAPSHOT`**
5. **Finish** をクリックしてプロジェクトを作成します。
   これは、基本的なプロジェクト構造、クラスパス、サンプルルールを設定します。以下は、プロジェクト構造の概要です。

```
my-project
 `-- src/main/java
    | `-- com.sample
    |    `-- DecisionTableTest.java
    |    `-- DroolsTest.java
    |    `-- ProcessTest.java
    |
 `-- src/main/resources
    | `-- dtables
    |    `-- Sample.xls
    | `-- process
    |    `-- sample.bpmn
    | `-- rules
    |    `-- Sample.drl
    | `-- META-INF
    |
 `-- JRE System Library
    |
 `-- Maven Dependencies
    |
 `-- Drools Library
    |
 `-- src
    |
 `-- target
    |
 `-- pom.xml
```

以下の要素に注目してください。

- **src/main/resources** ディレクトリーの **Sample.drl** ルールファイル。これには、サンプルの **Hello World** ルールおよび **GoodBye** ルールが含まれます。
- **com.sample** パッケージの **src/main/java** ディレクトリーにある **DroolsTest.java** ファイル。**Sample.drl** ルールの実行には、**DroolsTest** クラスを使用できます。
- 実行するのに必要な JAR ファイルを含むカスタムのクラスパスとなる **Drools Library** ディレクトリー。

既存の **Sample.drl** ファイルおよび **DroolsTest.java** ファイルを必要に応じて新しい設定に変更するか、ルールファイルをオブジェクトファイルを新たに作成します。この手順では、ルールと Java オブジェクトを新たに作成します。

6. ルールが有効な Java オブジェクトを作成します。
この例では、 **my-project/src/main/java/com.sample** に **Person.java** ファイルが作成されます。**Person** クラスには、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。

```
public class Person {
  private String firstName;
  private String lastName;
  private Integer hourlyRate;
  private Integer wage;

  public String getFirstName() {
```

```
      return firstName;
    }

    public void setFirstName(String firstName) {
      this.firstName = firstName;
    }

    public String getLastName() {
      return lastName;
    }

    public void setLastName(String lastName) {
      this.lastName = lastName;
    }

    public Integer getHourlyRate() {
      return hourlyRate;
    }

    public void setHourlyRate(Integer hourlyRate) {
      this.hourlyRate = hourlyRate;
    }

    public Integer getWage(){
      return wage;
    }

    public void setWage(Integer wage){
      this.wage = wage;
    }
  }
```

7. **File** → **Save** をクリックして、ファイルを保存します。

8. **my-project/src/main/resources/rules** に、**.drl** 形式のルールファイルを作成します。DRL ファイルには、少なくともパッケージの指定と、(1 つまたは複数の) ルールで使用されるデータオブジェクトのインポートリストと、**when** 条件および **then** アクションを持つ 1 つ以上のルールが含まれます。
   以下の **Wage.drl** ファイルには、**Person** クラスをインポートする **Wage** ルールが含まれ、賃金および時給の値を計算し、その結果に基づいてメッセージを表示します。

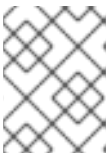
```
package com.sample;

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
```

9. **File** → **Save** をクリックして、ファイルを保存します。
10. メインクラスを作成し、Java オブジェクトを作成したディレクトリーに保存します。メインクラスは KIE ベースをロードし、ルールを実行します。



注記

また、**DroolsTest.java** サンプルファイルと同様に、**main()** メソッドと **Person** クラスを 1 つの Java オブジェクトファイルに追加できます。

11. メインクラスに、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のに必要な **import** 命令文を追加します。つぎに、KIE ベースをロードし、ファクトを挿入し、ファクトモデルをルールに渡す **main()** メソッドからルールを実行します。
この例では、必要なインポートと **main()** メソッドを使用して、**my-project/src/main/java/com.sample** に **RulesTest.java** ファイルを作成します。

```
package com.sample;

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 {
      // Load the KIE base:
      KieServices ks = KieServices.Factory.get();
      KieContainer kContainer = ks.getKieClasspathContainer();
      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();
    }
  }
}
```

12. **File** → **Save** をクリックして、ファイルを保存します。
13. プロジェクトで DRL アセットをすべて作成して保存したあと、プロジェクトフォルダーを右ク

リックして、**Run As** → **Java Application** を選択してプロジェクトをビルドします。プロジェクトのビルドに失敗したら、Developer Studio の下部ウィンドウの **Problems** タブに記載されている問題に対応し、プロジェクトビルドされるまで妥当性確認を行います。

RUN AS → JAVA APPLICATION オプションが利用できない場合

プロジェクトを右クリックして、**Run As** を選択した場合に **Java Application** が選択肢にない場合は、**Run As** → **Run Configurations** に移動して **Java Application** を右クリックし、**New** をクリックします。次に、**Main** タブで、**Project** と、関連する **Main class** を参照して選択します。**Apply** をクリックし、**Run** をクリックしてプロジェクトをテストします。再度プロジェクトフォルダーを右クリックすると、**Java Application** オプションが表示されます。

Red Hat Decision Manager で、新しいルールアセットを既存プロジェクトと統合するには、新しいプロジェクトをナレッジ JAR (KJAR) としてコンパイルし、Decision Central のプロジェクトの **pom.xml** ファイルに依存関係として追加します。

6.2. JAVA を使用した DRL ルールの作成および実行

Java オブジェクトを使用して、ルールが含まれる DRL ファイルを作成し、Red Hat Decision Manager デシジョンサービスにオブジェクトを統合します。DRL ルールを作成する方法は、デシジョンサービスに外部 Java オブジェクトを使用している場合や、同じワークフローを継続する場合に便利です。この方法を使用しなくなった場合は、Red Hat Decision Manager の Decision Central インターフェースを使用して、DRL ファイルや、その他のルールアセットを作成することが推奨されます。

手順

1. ルールが有効な Java オブジェクトを作成します。
この例では、**my-project** ディレクトリーに **Person.java** ファイルが作成されます。**Person** クラスには、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。

```
public class Person {
  private String firstName;
  private String lastName;
  private Integer hourlyRate;
  private Integer wage;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
```

```
    public Integer getHourlyRate() {
      return hourlyRate;
    }

    public void setHourlyRate(Integer hourlyRate) {
      this.hourlyRate = hourlyRate;
    }

    public Integer getWage(){
      return wage;
    }

    public void setWage(Integer wage){
      this.wage = wage;
    }
  }
```

2. **my-project** ディレクトリーに、**.drl** 形式のルールファイルを作成します。DRL ファイルには、少なくともパッケージの指定と、(1 つまたは複数の) ルールで使用されるデータオブジェクトのインポートリストと、**when** 条件および **then** アクションを持つ 1 つ以上のルールが含まれます。
以下の **Wage.drl** ファイルには、**Wage** ルールが含まれており、このルールで、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
```

3. メインクラスを作成し、Java オブジェクトを作成したディレクトリーに保存します。メインクラスは KIE ベースをロードし、ルールを実行します。

4. メインクラスに、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のに必要な **import** 命令文を追加します。つぎに、KIE ベースをロードし、ファクトを挿入し、ファクトモデルをルールに渡す **main()** メソッドからルールを実行します。
この例では、必要なインポートと **main()** メソッドを使用して、**my-project** に **RulesTest.java** ファイルを作成します。

```
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 {
```

```
        // Load the KIE base:
        KieServices ks = KieServices.Factory.get();
        KieContainer kContainer = ks.getKieClasspathContainer();
        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();
      }
    }
  }
```

5. Red Hat カスタマーポータル から **Red Hat Decision Manager 7.1.0 Source Distribution** の ZIP ファイルをダウンロードし、**my-project/dm-engine-jars/** で展開します。

6. **my-project/META-INF** ディレクトリーに、以下の内容の **kmodule.xml** メタデータファイルを作成します。

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
</kmodule>
```

この **kmodule.xml** ファイルは、KIE ベースへのリソースを選択し、セッションを設定する KIE モジュールの記述子です。このファイルを使用すると、KIE ベースを 1 つ以上定義して設定し、特定の KIE ベースの特定の **packages** から DRL ファイルを含めることができます。各 KIE ベースから KIE セッションを 1 つ以上作成することもできます。

次の例は、より高度な **kmodule.xml** ファイルを表示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.drools.org/xsd/kmodule">
  <kbase name="KBase1" default="true" eventProcessingMode="cloud"
equalsBehavior="equality" declarativeAgenda="enabled"
packages="org.domain.pkg1">
    <ksession name="KSession1_1" type="stateful" default="true" />
    <ksession name="KSession1_2" type="stateful" default="true"
beliefSystem="jtms" />
  </kbase>
  <kbase name="KBase2" default="false" eventProcessingMode="stream"
equalsBehavior="equality" declarativeAgenda="enabled"
```

```
packages="org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    <ksession name="KSession2_1" type="stateless" default="true"
clockType="realtime">
      <fileLogger file="debugInfo" threaded="true" interval="10" />
      <workItemHandlers>
        <workItemHandler name="name" type="new
org.domain.WorkItemHandler()" />
      </workItemHandlers>
      <listeners>
        <ruleRuntimeEventListener
type="org.domain.RuleRuntimeListener" />
        <agendaEventListener type="org.domain.FirstAgendaListener"
/>
        <agendaEventListener type="org.domain.SecondAgendaListener"
/>
        <processEventListener type="org.domain.ProcessListener" />
      </listeners>
    </ksession>
  </kbase>
</kmodule>
```

この例は、KIE ベースを 2 つ定義します。KIE ベース **KBase1** から 2 つの KIE セッションをインスタンス化し、**KBase2** セッションから 1 つの KIE セッションをインスタンス化します。**KBase2** の KIE セッションは、**ステートレス**な KIE セッションですが、これは 1 つ前の KIE セッションで呼び出されたデータ (1 つ前のセッションの状態) が、セッションの呼び出しと呼び出しの間で破棄されることを示しています。ルールアセットで特定の **packages** が両方の KIE ベースに含まれます。この方法でパッケージを指定した場合は、指定したパッケージを反映するフォルダー構造で DRL ファイルを整理する必要があります。

7. Java オブジェクトですべての DRL アセットを作成して保存したあと、コマンドラインで **my-project** ディレクトリーに移動し、以下のコマンドを実行して Java ファイルをビルドします。**RulesTest.java** を、Java のメインクラスの名前に置き換えます。

```
javac -classpath "./dm-engine-jars/*:." RulesTest.java
```

ビルドに失敗したら、コマンドラインのエラーメッセージに記載されている問題に対応し、エラーが表示されなくなるまで Java オブジェクトの妥当性確認を行います。

8. Java ファイルが問題なくビルトできたら、以下のコマンドを実行してローカルでルールを実行します。**RulesTest** を、Java のメインクラスの接頭辞に置き換えます。

```
java -classpath "./dm-engine-jars/*:." RulesTest
```

9. ルールを見直して、適切に実行したことを確認し、Java ファイルで必要な変更に対応します。

新しいルールアセットを、Red Hat Decision Manager の既存のプロジェクトと統合するには、新しい Java プロジェクトをナレッジ JAR (KJAR) としてコンパイルし、Decision Central でプロジェクトの **pom.xml** ファイルに依存関係として追加します。

6.3. MAVEN を使用した DRL ルールの作成および実行

Maven アーキタイプを使用して、ルールが含まれる DRL ファイルを作成し、アーキタイプを Red Hat Decision Manager デシジョンサービスに統合します。DRL ルールを作成する方法は、デシジョンサービスに外部 Maven アーキタイプを使用している場合や、同じワークフローを継続する場合に便利で

す。この方法を使用しなくなった場合は、Red Hat Decision Manager の Decision Central インターフェースを使用して、DRL ファイルや、その他のルールアセットを作成することが推奨されます。

手順

1. Maven アーキタイプを作成するディレクトリーに移動して、次のコマンドを実行します。

```
mvn archetype:generate -DgroupId=com.sample.app -DartifactId=my-app
-DarchetypeArtifactId=maven-archetype-quickstart -
DinteractiveMode=false
```

これにより、**my-app** という名前のディレクトリーが、以下の構造で作成されます。

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- sample
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- sample
                    `-- app
                        `-- AppTest.java
```

my-app ディレクトリーには、以下の重要なコンポーネントが含まれます。

- アプリケーションソースを保存する **src/main** ディレクトリー
- テストソースを保存する **src/test** ディレクトリー
- プロジェクト設定ファイル **pom.xml**

2. Maven アーキタイプに、ルールが有効な Java オブジェクトを作成します。
この例では、**my-app/src/main/java/com/sample/app** ディレクトリーに **Person.java** ファイルが作成されます。**Person** クラスには、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。

```
package com.sample.app;

  public class Person {

    private String firstName;
    private String lastName;
    private Integer hourlyRate;
    private Integer wage;

    public String getFirstName() {
      return firstName;
    }
```

```
    public void setFirstName(String firstName) {
      this.firstName = firstName;
    }

    public String getLastName() {
      return lastName;
    }

    public void setLastName(String lastName) {
      this.lastName = lastName;
    }

    public Integer getHourlyRate() {
      return hourlyRate;
    }

    public void setHourlyRate(Integer hourlyRate) {
      this.hourlyRate = hourlyRate;
    }

    public Integer getWage(){
      return wage;
    }

    public void setWage(Integer wage){
      this.wage = wage;
    }
  }
```

3. **my-app/src/main/resources/rules** に、**.drl** 形式のルールファイルを作成します。DRL ファイルには、少なくともパッケージの指定と、(1 つまたは複数の) ルールで使用されるデータオブジェクトのインポートリストと、**when** 条件および **then** アクションを持つ 1 つ以上のルールが含まれます。
以下の **Wage.drl** ファイルには、**Person** クラスをインポートする **Wage** ルールが含まれ、賃金および時給の値を計算し、その結果に基づいてメッセージを表示します。

```
package com.sample.app;

import com.sample.app.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
```

4. **my-app/src/main/resources/META-INF** ディレクトリーに、以下の内容の **kmodule.xml** メタデータファイルを作成します。

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
</kmodule>
```

この **kmodule.xml** ファイルは、KIE ベースへのリソースを選択し、セッションを設定する KIE モジュールの記述子です。このファイルを使用すると、KIE ベースを 1 つ以上定義して設定し、特定の KIE ベースの特定の **packages** から DRL ファイルを含めることができます。各 KIE ベースから KIE セッションを 1 つ以上作成することもできます。

次の例は、より高度な **kmodule.xml** ファイルを表示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.drools.org/xsd/kmodule">
  <kbase name="KBase1" default="true" eventProcessingMode="cloud"
equalsBehavior="equality" declarativeAgenda="enabled"
packages="org.domain.pkg1">
    <ksession name="KSession1_1" type="stateful" default="true" />
    <ksession name="KSession1_2" type="stateful" default="true"
beliefSystem="jtms" />
  </kbase>
  <kbase name="KBase2" default="false" eventProcessingMode="stream"
equalsBehavior="equality" declarativeAgenda="enabled"
packages="org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    <ksession name="KSession2_1" type="stateless" default="true"
clockType="realtime">
      <fileLogger file="debugInfo" threaded="true" interval="10" />
      <workItemHandlers>
        <workItemHandler name="name" type="new
org.domain.WorkItemHandler()" />
      </workItemHandlers>
      <listeners>
        <ruleRuntimeEventListener
type="org.domain.RuleRuntimeListener" />
        <agendaEventListener type="org.domain.FirstAgendaListener"
/>
        <agendaEventListener type="org.domain.SecondAgendaListener"
/>
        <processEventListener type="org.domain.ProcessListener" />
      </listeners>
    </ksession>
  </kbase>
</kmodule>
```

この例は、KIE ベースを 2 つ定義します。KIE ベース **KBase1** から 2 つの KIE セッションをインスタンス化し、**KBase2** セッションから 1 つの KIE セッションをインスタンス化します。**KBase2** の KIE セッションは、**ステートレス**な KIE セッションですが、これは 1 つ前の KIE セッションで呼び出されたデータ (1 つ前のセッションの状態) が、セッションの呼び出しと呼び出しの間で破棄されることを示しています。ルールアセットで特定の **packages** が両方の KIE ベースに含まれます。この方法でパッケージを指定した場合は、指定したパッケージを反映するフォルダー構造で DRL ファイルを整理する必要があります。

5. **my-app/pom.xml** 設定ファイルで、アプリケーションが要求するライブラリーを指定します。Red Hat Decision Manager の依存関係と、アプリケーションの **group ID**、**artifact ID**、および **version** (GAV) を提供します。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.sample.app</groupId>
<artifactId>my-app</artifactId>
<version>1.0.0</version>
<repositories>
  <repository>
    <id>jboss-ga-repository</id>
    <url>http://maven.repository.redhat.com/ga/</url>
  </repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-compiler</artifactId>
    <version>VERSION</version>
  </dependency>
  <dependency>
    <groupId>org.kie</groupId>
    <artifactId>kie-api</artifactId>
    <version>VERSION</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>
```

Red Hat Decision Manager における Maven 依存関係および BOM (Bill of Materials) については、「What is the mapping between Red Hat Decision Manager and Maven library version?」を参照してください。

6. **my-app/src/test/java/com/sample/app/AppTest.java** の **testApp** メソッドを使用してルールをテストします。Maven によって、**AppTest.java** ファイルがデフォルトで作成されます。

7. **AppTest.java** ファイルで、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のに必要な **import** 命令文を追加します。次に、KIE ベースをロードし、ファクトを挿入し、ファクトモデルをルールに渡す **testApp()** メソッドからルールを実行します。

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

public void testApp() {

  // Load the KIE base:
  KieServices ks = KieServices.Factory.get();
```

```
    KieContainer kContainer = ks.getKieClasspathContainer();
    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();
}
```

8. Maven アーキタイプにすべての DRL アセットを作成して保存したあと、コマンドラインで **my-app** ディレクトリーに移動し、以下のコマンドを実行してファイルを作成します。

```
mvn clean install
```

ビルドに失敗したら、コマンドラインのエラーメッセージに記載されている問題に対応し、ビルドに成功するまでファイルの妥当性確認を行います。

9. ファイルが問題なくビルドできたら、以下のコマンドを実行してローカルでルールを実行します。**com.sample.app** をパッケージ名に置き換えます。

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

10. ルールを見直して、適切に実行したことを確認し、ファイルで必要な変更に対応します。

Red Hat Decision Manager で、新しいルールアセットを既存のプロジェクトと統合するには、新しい Maven プロジェクトをナレッジ JAR (KJAR) としてコンパイルし、Decision Central のプロジェクトの **pom.xml** ファイルに依存関係として追加します。

6.4. 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 Decision 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 Decision 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 Decision 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 Decision 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、よりすばやく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。

6.4.1. 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

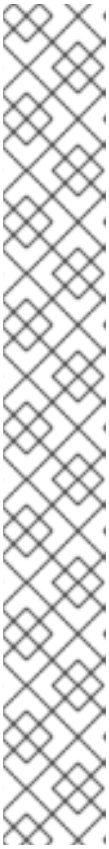
Red Hat Decision Manager ビジネスアセットを含む Maven 化されたプロジェクトがあること

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>${rhdm.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 Decision Manager の Maven アーティファクトのバージョン (例: 7.11.0.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 Decision 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 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>7.1.0.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 RHDM 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 Decision 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Red Hat Decision 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>${rhdm.version}</version>
</dependency>

<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-model-compiler</artifactId>
  <version>${rhdm.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、「**Packaging and deploying a Red Hat Decision Manager project**」を参照してください。

6.4.2. 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 Decision 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

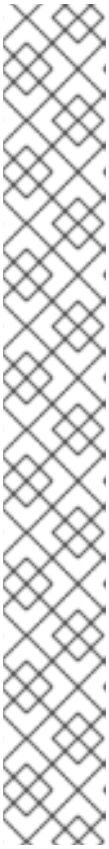
手順

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 Decision 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Red Hat Decision 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>${rhdm.version}</version>
</dependency>

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

<version> は、プロジェクトで現在使用する Red Hat Decision Manager の Maven アーティファクトバージョンです (例: 7.11.0.Final-redhat-00002)。



注記

個別の依存関係に対して Red Hat Decision 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 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>7.1.0.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 RHDM 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 四季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、**「Packaging and deploying a Red Hat Decision Manager project」**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
- Packaging and deploying a Red Hat Decision Manager project

付録A バージョン情報

本書の最終更新日: 2018 年 11 月 1 日