



# Drools

---

David THIBAU – 2023

[david.thibau@gmail.com](mailto:david.thibau@gmail.com)



# Agenda

---

- Introduction
  - BRMS
  - Les projets KIE
  - Le moteur de règles Drools
  - Alternatives d'architecture
- Prise en main
  - KIE APIs
  - Session stateless
  - Session stateful et inférence
  - Agenda et conflits
  - Listeners, Channels, Entry-points
- Le langage de règles DRL
  - Éléments principaux
  - Règles et attributs
  - LHS
  - RHS
  - Requêtes
- Performance
  - ReteOO et PHREAK
  - Drools metric
- Projets connexes
  - Intégration avec jBPM
  - Drools Fusion
- Intégrer Kie
  - Considérations et patterns d'intégration
  - Dépôts de connaissance Maven
  - Knowledge As A Service
  - Drools Workbench
- Annexes



# Introduction

---

## **BRMS**

Les projets KIE

Le moteur de règles Drools

Mise en place IDE



# Problématique

---

Aujourd'hui, le principal challenge des applications métier est l'agilité

Les applications doivent pouvoir s'adapter rapidement et réagir :

- A des demandes d'évolutions fonctionnelles
- A des changement de règles métier
- A des changements d'organisation
- A l'intégration de nouveaux systèmes



# Approche traditionnelle

---

L'approche traditionnelle pour la mise à jour des règles métier même simples introduit de long délais de réalisation :

- 1) Expression de la règle
- 2) Prioritisation et allocation de ressources
- 3) Analyse
- 4) Code, Test/ Debug
- 5) Déploiement



# Solutions classiques

---

L'externalisation des règles métier dans des paramètres de configuration (Fichiers ou BD)

=> Ne répond pas à

*Si le client habite Paris, est âgé de plus de 55 ans et est client depuis plus de 2 ans, accorder une réduction de 10%*

Le codage de ce type de règles dans l'application

- Maintenance très délicate
- Code spaghetti pouvant être inefficace



# Système de gestion de règles

---

Un **système de gestion de règles (BRMS<sup>1</sup>)**

identifie la notion de règle métier comme une ressource pouvant être gérée indépendamment du code applicatif

- Les règles peuvent donc être gérées, versionnées, monitorées
- Elles peuvent être mises à jour par un technicien ou un expert métier sans redéploiement de l'application
- Elles peuvent être testées indépendamment
- Elles peuvent être documentées et auditées



# Moteur de règle

---

Les BRMS intègrent un **moteur de règle**

C'est un composant qui évaluent des instructions IF-THEN basées sur le modèle métier : les règles

Lorsque les conditions des règles sont satisfaites (*IF*), il exécute les actions associées (*THEN*) qui généralement modifient le modèle

=> La programmation devient alors déclarative

=> La logique métier n'est plus dispersée dans le code mais centralisée dans un repository de règles





# Scénario d'utilisation

---

## Service de règle :

- Parse et compile l'ensemble des règles. (Fait en général une seule fois au démarrage de l'application)

## Client (code applicatif) :

- Obtient une référence (locale ou service web) sur le service de règle
- Insère des faits (Objets du domaine) dans la mémoire de travail
- Demande au service de déclencher les règles
- Récupère les objets du domaine modifiés par le service de règles



# Mise en place

---

Adopter une solution basée sur les règles nécessite :

- L'identification des règles métier : Expert métier
- Leur organisation et leur implémentation dans un langage de règles : Expert métier / technique
- L'intégration du service de règles à l'application : Embarqué / distant
- La mise à disposition d'une interface de gestion des règles (Outil BRMS) : Métier / Technique
- La mise en place de procédure de mise à jour (Automatisation des tests/Déploiement) : Technique



# Considérations

---

- Un moteur de règles s'appuie sur des technologies complexes
- Il est difficile de se baser sur une boîte noire
- Comment les règles se déclenchent n'est pas très intuitif
- L'extraction des règles n'est pas toujours simple



# Avantages

---

L'utilisation d'un moteur de règles apporte certains avantages :

- **Programmation déclarative** : Les moteurs de règles permettent de spécifier « Quoi faire » et non pas « Comment le faire ». Ils sont capables de résoudre des problèmes difficiles et en plus de fournir des explications !
- **Séparation de la logique et des données** : Les données sont dans les objets du domaine, la logique centralisée dans un fichier de règles (différent de l'approche OO qui encapsule attributs et méthodes). La logique n'est plus dispersée
- **Centralisation de la connaissance** : Tout est centralisé dans la base de connaissance, relativement lisible et pouvant servir de documentation
- **Règles compréhensibles** : En définissant des langages spécifiques au domaine ou au métier, les règles sont exprimées en langage quasi-naturel et deviennent accessibles aux experts métier



# Quand utiliser un moteur de règles ?

---

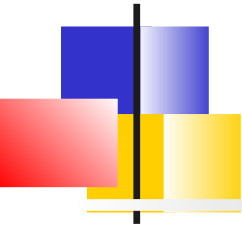
Le problème est trop complexe pour le code classique (problèmes d'optimisation par exemple, système expert)

Le problème n'est pas complexe mais aucune solution robuste s'impose.

La logique change souvent, dans ce cas les règles peuvent être changées rapidement sans trop de risques.

Les experts métiers existent mais ne sont pas techniques. Les règles permettent alors d'exprimer la logique métier dans leur propre termes.

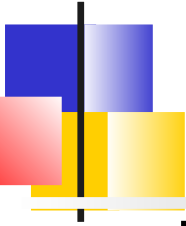
# Quand ne pas utiliser de moteur de règles ?



Un moteur de règles est juste une partie d'une application complexe, il n'est pas nécessaire de tout implémenter sous forme de règles

Un bon indicateur est le degré de couplage entre les règles.

- Si le déclenchement d'une règle déclenche **invariablement** une chaînes de règles, alors l'implémentation de cette logique à base de règles n'est sûrement pas adaptée, un arbre décisionnel peut suffire



# Domaines et cas d'utilisation

---

## Domaines

- Finance et Assurance : Aide à la décision
- Réglementation, règles gouvernementales
- E-commerce, campagne de promotion
- Gestion des risques

## Exemple

- Autorisation basée sur de nombreux critères (Rôle, propriété de l'entité, organisation, Localité, ...)
- Personnalisation d'application (ex : gestion d'une page d'accueil personnalisée d'un site de e-commerce)
- Diagnostique
- Validation complexe
- Problèmes d'optimisation (routage, planning, ...)



# Introduction

---

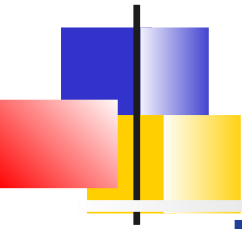
BRMS

**Les projets KIE**

Le moteur de règles Drools

Mise en place IDE



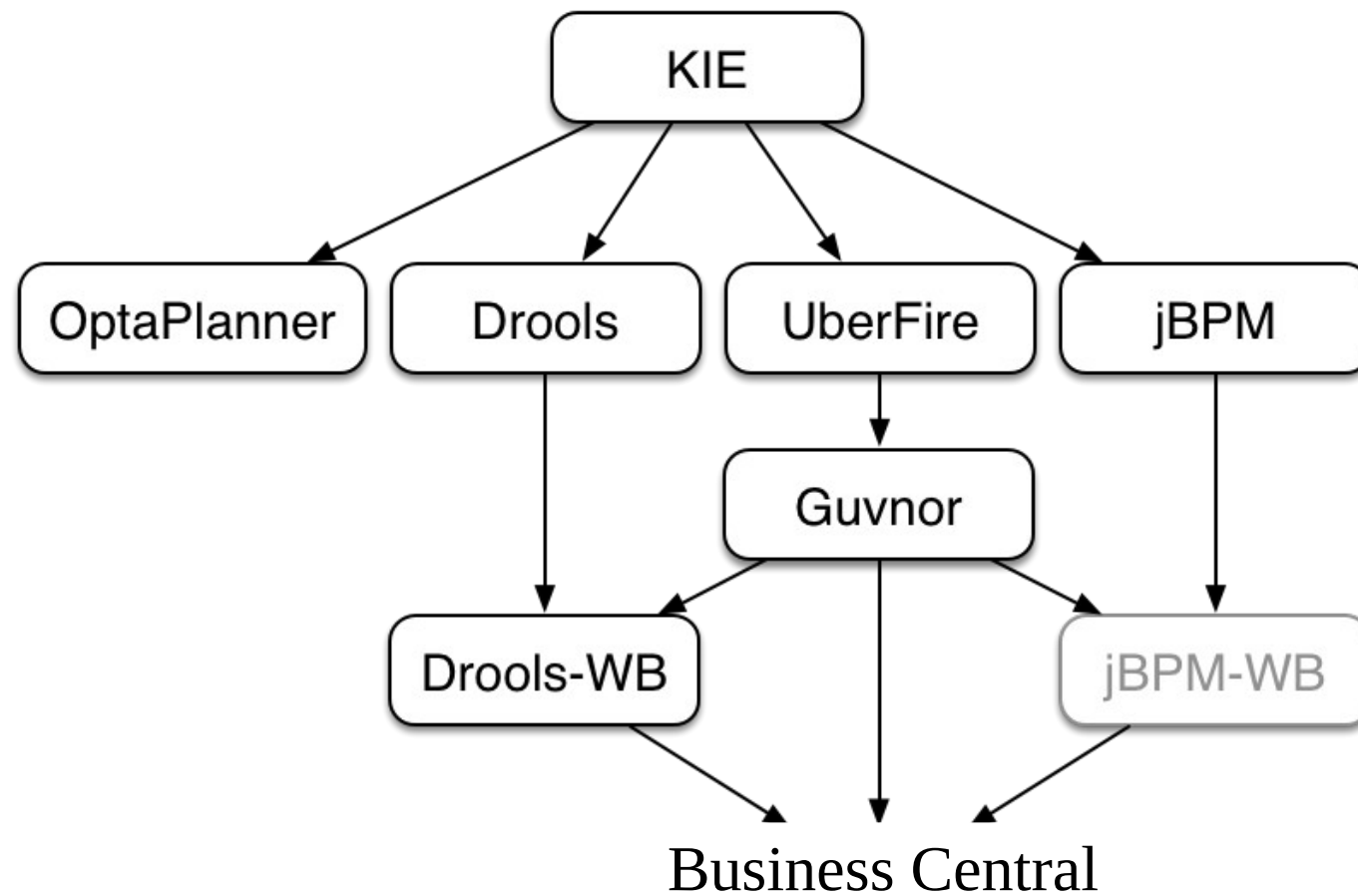


# Projets KIE

**KIE (Knowledge Is Everything)** regroupe plusieurs projets qui partagent la même API et les mêmes techniques de construction et de déploiement (basées sur Maven et Git)

- **Drools** : Moteur de règles et traitement complexe d'événements (Complex Event Processing)
- **jBPM** : Moteur de workflow
- **OptaPlanner** permet d'optimiser des problèmes complexe soumis à des contraintes (Planification, itinéraire des véhicules, ...)
- **Business Central** : Application web pour la gestion des règles et processus métier.
- **UberFire** : Un framework pour facilement construire des interfaces web : les workbenches

# Projets KIE





# Cycle de vie d'un projet Business Central

---

**Authoring** : Création de la connaissance en utilisant un langage spécifique : DRL, BPMN2, Table de décision, ...

**Build** : Construction d'un artefact de la connaissance déployable (**.jar**)

**Test** : Tests de la règle, du processus

**Déploiement** : Déploiement dans un dépôt (Typiquement **Maven**)

**Utilisation** : Le jar est exposé via un objet **KieContainer**. Pour interagir avec le système, une application crée des **KieSession** à partir du conteneur pour interagir avec le système de règles

**Exécution** : Appels de l'API de *KieSession* (*embarqué ou API Rest*)

**Travaux** : Interaction utilisateur avec la *KieSession* via des commandes en ligne ou une interface utilisateur

**Exploitation** : Surveillance des sessions, monitoring, reporting



# Environnement

---

## Développement - 2 alternatives

- Business Central + 1 ou plusieurs KieServer permettant de déployer et tester les règles
- Eclipse plugin avec classes de test Java et procédure de déploiement vers KieServer ou autre

## Exécution des règles : 2 alternatives

- Application Java embarquant les librairies Kie et Drools
- KieServer

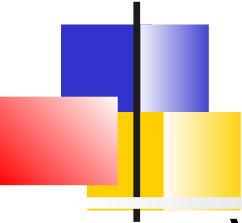


# Ecriture des règles

---

Drools supporte différents assets pour spécifier les règles :

- Decision Model and Notation (DMN) : Diagramme de décision à base de XML
- Tables de décision : Excel ou Table de décision guidée de Business Central
- Règle guidée : Expression d'une seule règle via Business Central
- DRL : Fichier texte offrant le plus de possibilité
- Predictive Model Markup Language (PMML) :  
Modèles d'analyse de données prédictifs en XML



# Options de stockage et de build pour les règles

---

## Versionning :

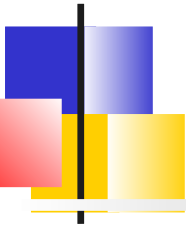
- Business Central Git VFS
- Dépôt Git externe

## Dépôts

- Business Central Maven repository
- Dépôt Maven externe

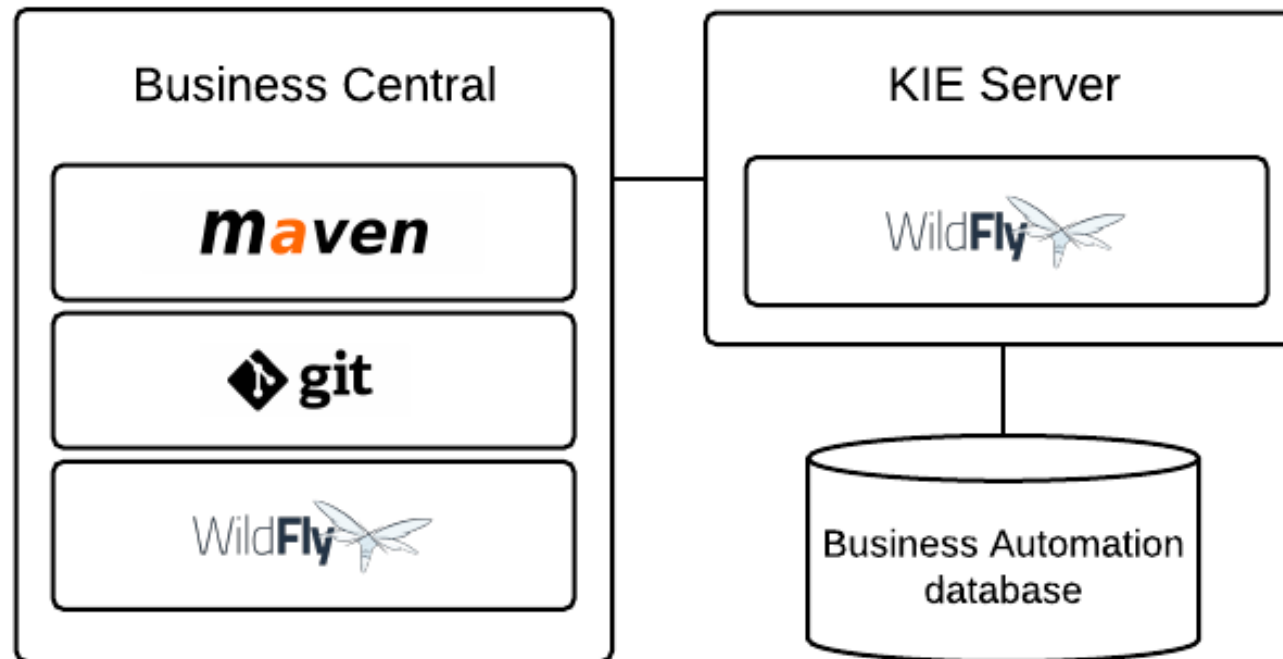
## Build

- Business Central
- Projet Maven indépendant
- Embarqué dans une application Java

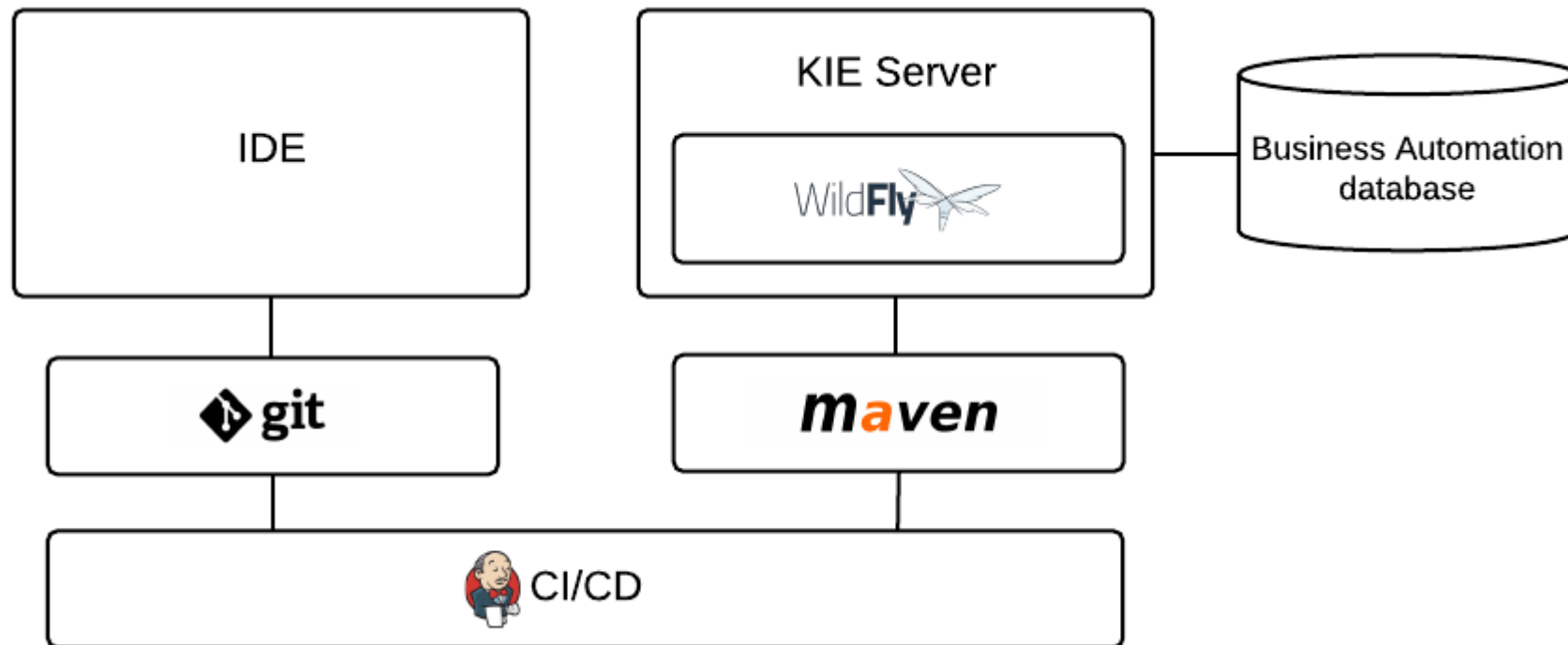


# Architecture Business Central

---



# IDE + KieServer

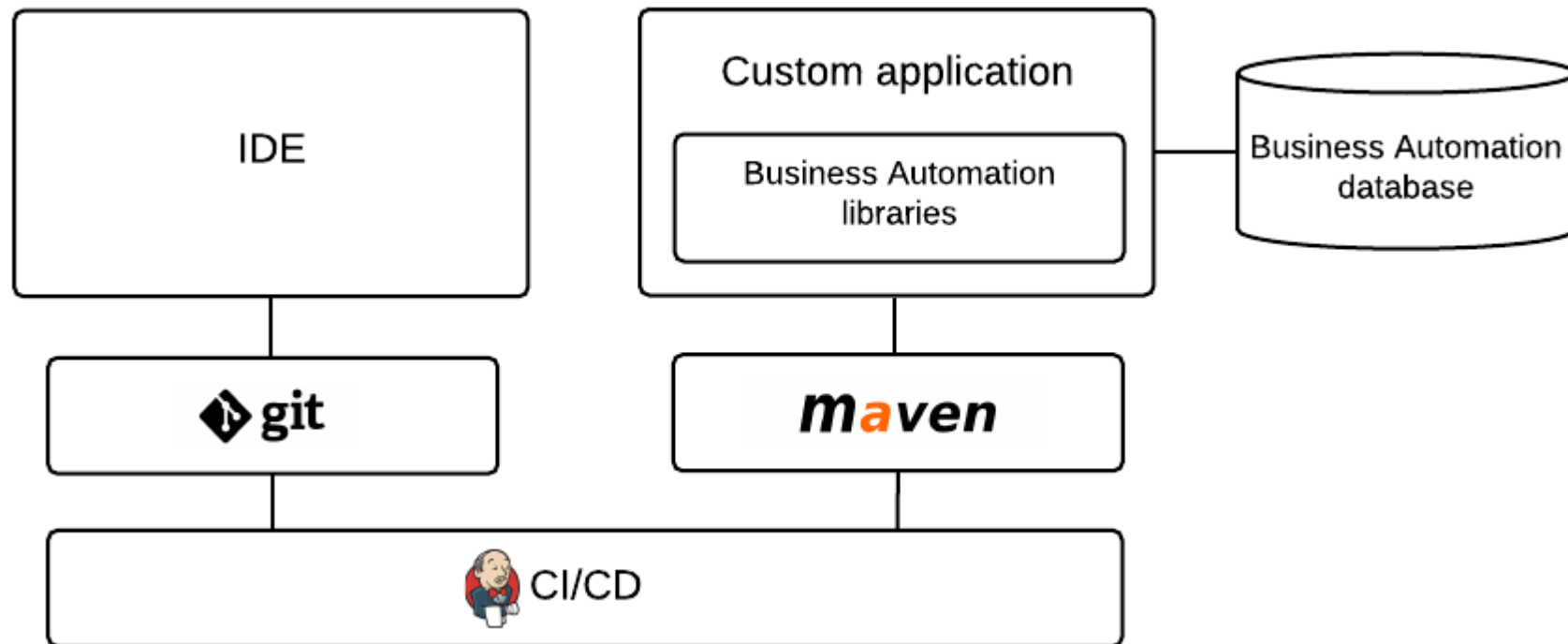






# IDE et Java embarqué

---





# Packaging

---

Quelque soit l'option d'architecture choisie, l'artefact produit est un JAR avec l'extension KJAR qui contient :

- Le descripteur de module *META-INF/kmodule.xml*
- Les fichiers ressources contenant les règles métiers
- Les classes du modèle du domaine.

L'outil utilisé pour le packaging est généralement Maven, un plugin permet de valider les fichiers de règles

Le code client et le runtime drools est :

- Packagé dans un projet distinct si déploiement sur KieServer
- Packagé avec le KJAR



# kmodule.xml

---

## Le descripteur *META-INF/kmodule.xml*

- Configure une ou plusieurs bases de connaissance en spécifiant les ressources (fichiers de règles ou processus)
- Pour chaque base de connaissance, configure une ou plusieurs types de sessions pouvant être créées.

Un descripteur vide applique une configuration par défaut :

- tous les fichiers ressources trouvés dans le classpath sont ajoutés à la même base de connaissance.
- 2 types de sessions (stateless et stateful) sont associées à l'unique base de connaissances



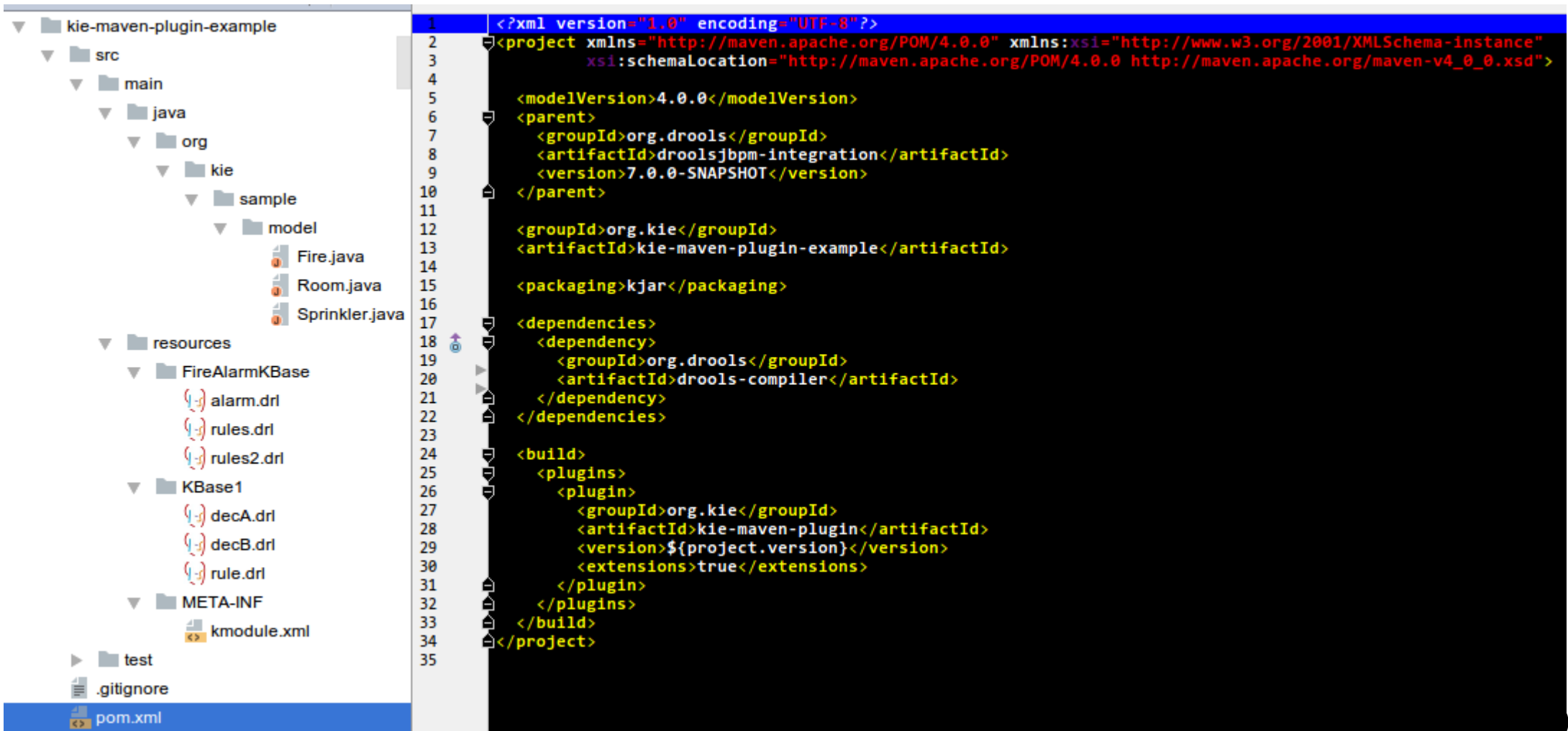
# Exemple *kmodule.xml*

```
<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="KSession2_1" type="stateful" default="true"/>
    <ksession name="KSession2_2" type="stateless" default="false" 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="KSession3_1" type="stateful" default="false" clockType="realtime">
      <fileLogger file="drools.log" threaded="true" interval="10"/>
      <workItemHandlers>
        <workItemHandler name="name" type="org.domain.WorkItemHandler"/>
      </workItemHandlers>
      <calendars>
        <calendar name="monday" type="org.domain.Monday"/>
      </calendars>
      <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>
```

# Exemple projet de règles indépendant



The image displays a screenshot of an IDE with a project structure on the left and the `pom.xml` file on the right.

**Project Structure (Left Panel):**

- kie-maven-plugin-example
  - src
    - main
      - java
        - org
          - kie
            - sample
              - model
                - Fire.java
                - Room.java
                - Sprinkler.java
    - resources
      - FireAlarmKBase
        - alarm.drl
        - rules.drl
        - rules2.drl
      - KBase1
        - decA.drl
        - decB.drl
        - rule.drl
      - META-INF
        - kmodule.xml
    - test
    - .gitignore
    - pom.xml

**pom.xml Content (Right Panel):**

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
4
5     <modelVersion>4.0.0</modelVersion>
6     <parent>
7         <groupId>org.drools</groupId>
8         <artifactId>droolsjbpm-integration</artifactId>
9         <version>7.0.0-SNAPSHOT</version>
10    </parent>
11
12    <groupId>org.kie</groupId>
13    <artifactId>kie-maven-plugin-example</artifactId>
14
15    <packaging>kjar</packaging>
16
17    <dependencies>
18        <dependency>
19            <groupId>org.drools</groupId>
20            <artifactId>drools-compiler</artifactId>
21        </dependency>
22    </dependencies>
23
24    <build>
25        <plugins>
26            <plugin>
27                <groupId>org.kie</groupId>
28                <artifactId>kie-maven-plugin</artifactId>
29                <version>${project.version}</version>
30                <extensions>true</extensions>
31            </plugin>
32        </plugins>
33    </build>
34 </project>
35
```



# Introduction

---

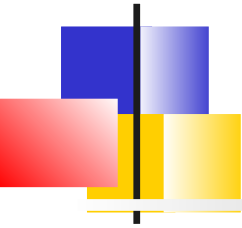
BRMS

Les projets KIE

**Le moteur de règles Drools**

Mise en place IDE

# Systeme de production de règle



Un système de production de règle est composé d'une **base de connaissance**, d'un **moteur d'inférence** et d'une **mémoire de travail**

- La base de connaissance regroupe les **règles compilées**
- L'application insère des **faits** (objets du modèle métier) dans la mémoire de travail
- Le moteur d'inférence, capable de manipuler un volume important de règles et de faits, a pour rôle de **comparer les faits aux conditions des règles**, si les conditions des règles sont satisfaites les actions correspondantes sont effectuées.  
Les actions modifient les faits de la mémoire de travail, ce qui peut déclencher l'activation d'autres règles.



# Composant Agenda

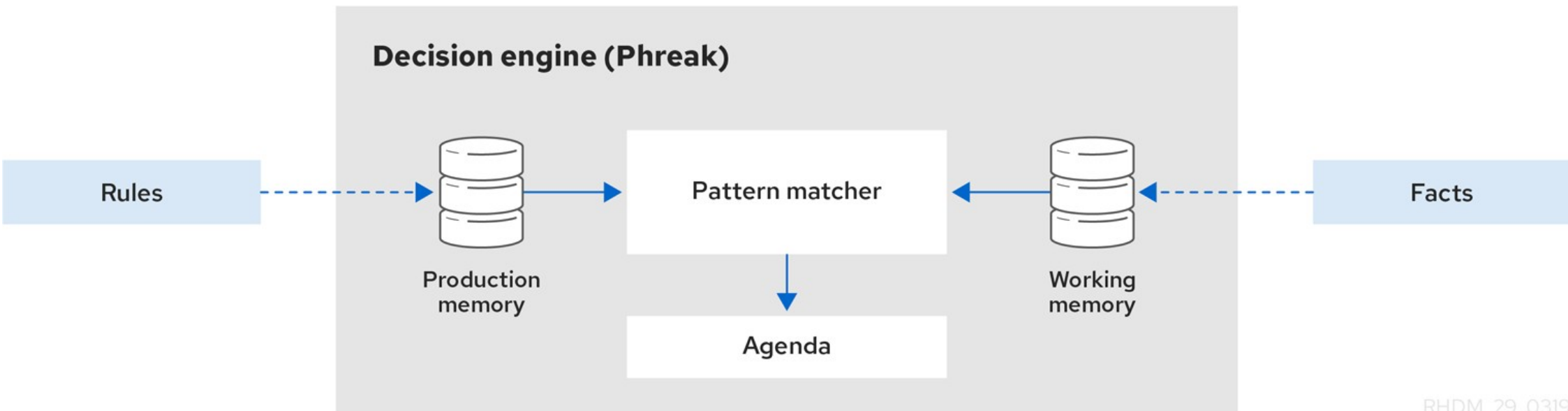
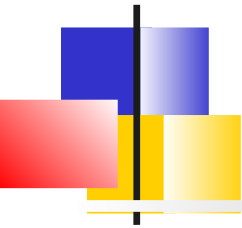
---

Lors du déclenchement des règles, il se peut que plusieurs règles soient actives simultanément, on dit alors qu'elles sont en conflit.

Le composant **Agenda** est responsable de gérer l'ordre d'exécution des règles en conflit en utilisant une stratégie de résolution de conflit. (priorité ou autre)

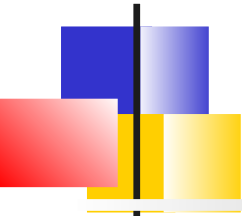


# Composants du moteur



RHDM\_29\_0319

# Algorithmes Drools



Le traitement de comparer les faits aux règles est appelé le **Pattern Matching**, il existe de nombreux algorithmes de pattern matching : *Linear, Rete, Treat, Leaps*.

Drools a implémenté l'algorithme de Rete dans une technologie objet

- Jusqu'à la version 5, Drools utilisait **ReteOO** qui se déclenchait à chaque insertion de fait
- A partir de la version 6, Drools utilise **PHREAK** par défaut qui doit être explicitement déclenché via l'API (Par ex : `ksession.fireAllRules()` )



# Introduction

---

BRMS

Les projets KIE

Le moteur de règles Drools

**Mise en place IDE**



# Installation IDE

---

Drools fournit des plugins Eclipse qui malheureusement sera abandonné, la version actuelle est buggée

Il offre :

- Une perspective Drools
- Des assistants de création de projet, de fichier .drl, de table de décision et de DSL
- Un éditeur et validateur de fichier drl
- Des vues pour le debugging



# Principaux Jars et dépendances

---

***knowledge-api.jar*** : Interfaces et usines. Identifie clairement l'API utilisateur de l'API moteur. Requis au runtime

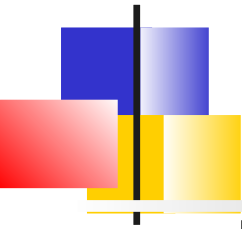
***knowledge-internal-api.jar*** : Interfaces et usines à usage interne

***drools-core.jar*** : Le moteur implémentant les algorithmes de *pattern matching*. Requis au runtime

***drools-compiler.jar*** : Compilateur des règles. Requis au runtime si les règles n'ont pas été pré-compilées

***drools-jsr94.jar*** : Couche d'implémentation de la JSR-94 au dessus de drools-compiler. Cependant, en général il est plus facile d'utiliser l'API Drools

***drools-decisiontables.jar*** : Compilateur pour les tables de décision supportant les formats excel et CSV.



# Vue Drools

---

Pour le debug, le plugin propose plusieurs vues permettant d'inspecter le moteur de règles. Ces vues sont disponibles lors de l'arrêt de l'exécution sur un point d'arrêt (par exemple sur `ksession.fireAllRules()` )

1. La **Working Memory View** permet d'inspecter tous les faits insérés
2. L' **Agenda View** permet de voir toutes les règles activées dans l'agenda. Pour chaque règle, le nom de la règle et ses variables sont montrées. (Ne fonctionne pas avec PHREAK)
3. La **Global Data View** permet de voir toutes les globales définies dans la mémoire de travail.



# La vue d'audit










---

La vue audit permet de visualiser un fichier de trace qui a pu être généré grâce au code suivant :

```
KieRuntimeLogger logger =  
KieServices.Factory.get().getLoggers().newFileLogger(k  
    session, "logdir/mylogfile");  
ksession.insert(...);  
ksession.fireAllRules();  
// stop logging  
logger.close();
```

Ou qui a pu être configuré via *kmodule.xml*

# Événements

1. Objet inséré : 
2. Objet mis à jour: 
3. Objet détruit : 
4. Activation créé : 
5. Activation annulé : 
6. Activation exécuté : 
7. Séquence de règle démarre ou se termine :
8. Activation ou désactivation d'un groupe de séquence de règle: 
9. Ajout ou suppression de packages: 
10. Ajout ou suppression de règle : 

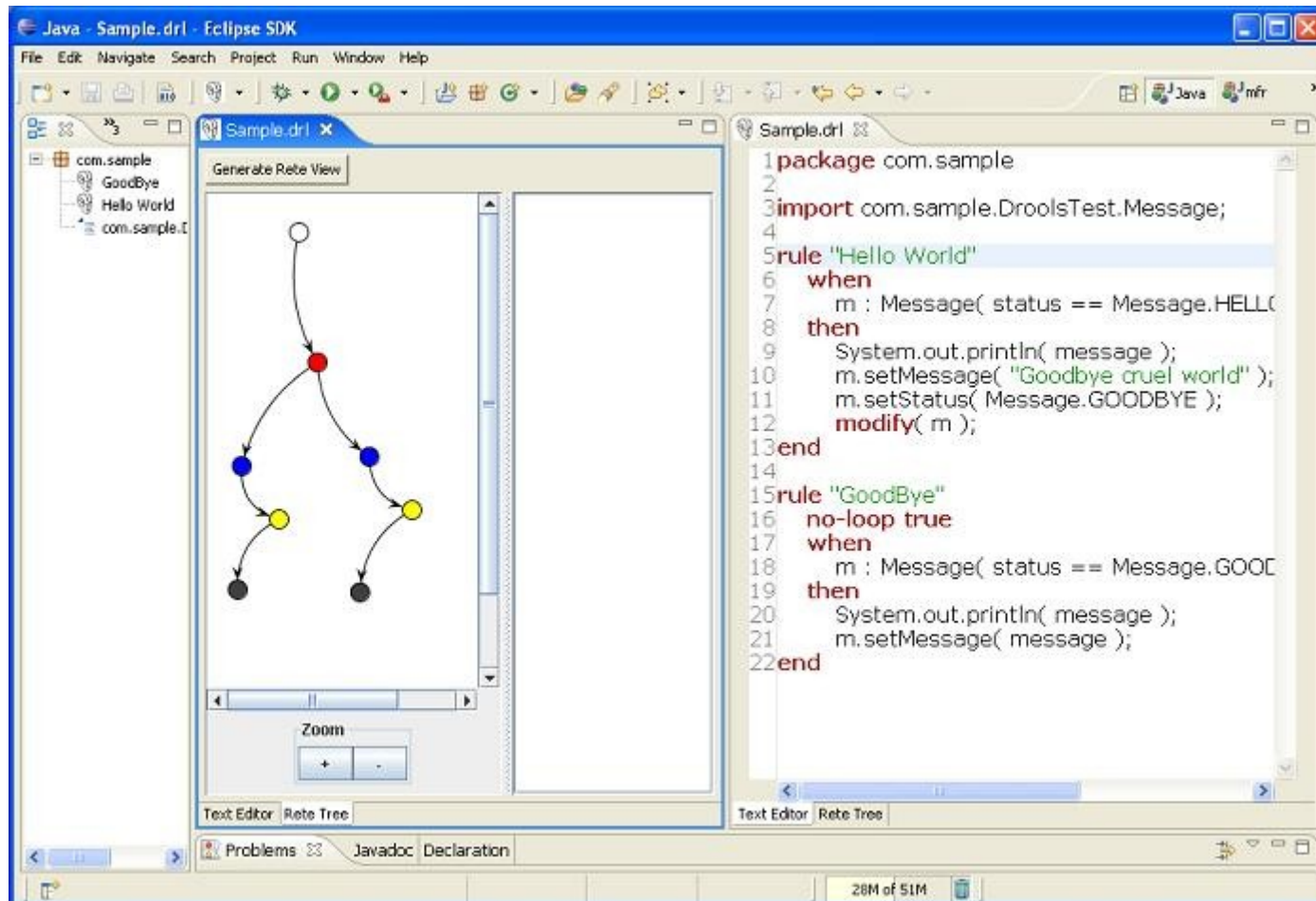


# Example

Activation executed: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:11]; guest=[fid:8:8]

- Object asserted (12): [Seating id=1, pid=0, pathDone=true, leftSeat=1, leftGuestName=n5, rightSeat=1, rightGuestName=n5]
- Object asserted (13): [Path id=1, seat=1, guest=n5]
- Object modified (11): [Count value=2]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:0:0]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:1:1]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:2:2]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:3:3]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:4:4]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:5:5]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:6:6]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:7:7]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:8:8]
- Object modified (10): [Context state=ASSIGN\_SEATS]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:2:2]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:4:4]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:8:8]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:6:6]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:1:1]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:3:3]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:7:7]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:5:5]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:0:0]
- Activation created: Rule findSeating leftGuestName=[fid:0:0]; rightGuestSex=[fid:8:8]; seatingId=[fid:12:12]; seatingRightGuestName=[fid:12:12]; context=[fid:10:16];
- Activation created: Rule findSeating leftGuestName=[fid:4:4]; rightGuestSex=[fid:8:8]; seatingId=[fid:12:12]; seatingRightGuestName=[fid:12:12]; context=[fid:10:16];

# Vue Rete





# Archetype Maven

---

Une alternative à l'utilisation de l'assistant de création de projet, il est possible d'utiliser un archetype Maven :

`org.kie:kie-drools-archetype`



# Légende Vue Rete

---

- Vert : Point d'entrée
- Rouge : ObjectTypeNode
- Bleu : AlphaNode
- Jaune : Adaptateur de l'entrée gauche
- Vert : BetaNode
- Black : Nœud règle

Sélectionner un nœud et visualiser la fenêtre propriétés



# Prise en main

---

## **KIE API**

Session stateless

Session stateful et inférence

Conflits et Agenda

Listeners, Channels, entry-point



# Classes principales

---

**KieServices** : Un singleton donnant accès aux autres services fournis par Kie

**KieModule** : Un conteneur des ressources nécessaires à une base de connaissance (inclut le pom.xml, kmodule.xml, fichiers de règles)

**KieContainer**: Un conteneur de toutes les bases de connaissances définies dans un module .

**KieBase** : Une base de connaissance contenant un ensemble de règles .

**KieSession** : Définit une manière d'interagir avec une base de connaissance



# *KieModule*

---

Un *KieModule* configure différentes  
*KieBase* et *KieSessions*

La configuration est faite

- Via un fichier XML :  
*META-INF/kmodule.xml*
- Ou programmatiquement via  
*KieModuleModel*



# Exemple KieServices

---

**// Récupérer le singleton KieServices**

```
KieServices kieServices =  
    KieServices.Factory.get();
```

**// Instancier un container chargeant les modules**

**// à partir du classpath**

```
KieContainer container =  
    KieServices.getKieClasspathContainer()
```





# *KieContainer*

---

Différents *KieContainer* sont fournis

- A partir du classpath
- A partir d'un dépôt maven
- Via une API REST

Un *KieContainer* permet de récupérer les bases de connaissance du module : les *KieBase*

Une *KieBase* permet d'instancier des sessions : *KieSession*



# Exemple

---

```
KieServices kieServices = KieServices.Factory.get();  
KieContainer kContainer = kieServices.getKieClasspathContainer();
```

```
KieBase kBase1 = kContainer.getKieBase("KBase1");  
KieSession kieSession1 = kContainer.newKieSession("KSession2_1");
```

```
StatelessKieSession kieSession2 =  
    kContainer.newStatelessKieSession("KSession2_2");
```



# *ReleaseId*

---

Un projet Kie est également un projet Maven

Le groupId, l'artifactId et la version déclarés dans le fichier pom.xml sont utilisés pour générer un ***ReleaseId***.

Un *KieContainer* peut également être obtenu via son *ReleaseId*.

```
KieServices kieServices = KieServices.Factory.get();  
ReleaseId rId = kieServices.newReleaseId( "org.acme", "myartifact", "1.0" );  
KieContainer kieContainer = kieServices.newKieContainer( rId );
```



# Types de sessions

---

Deux types de session Drools sont possibles :

- ***stateless*** : ***StatelessKieSession***.  
(N'utilise pas l'inférence)  
*StatelessKieSession statelessKieSession =  
kContainer.newStatelessKieSession();*
- ***stateful*** : ***KieSession***  
*KieSession ksession =  
kcontainer.newKieSession();*



# Prise en main

---

KIE API

**Session stateless**

Session stateful et inférence

Conflits et Agenda

Listeners, Channels, entry-point



# *Stateless Knowledge Session*

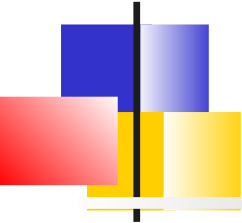
---

Les sessions ***stateless*** sont le cas le plus simple car elles n'utilisent pas le moteur d'inférence.

Elles peuvent être assimilées à une fonction à laquelle on passerait des arguments et qui provoquerait un résultat.

Les cas d'utilisation des sessions *stateless* sont généralement :

- La validation : Une personne est elle éligible pour un prêt ?
- Le calcul : Calculer une réduction
- Le routage ou le filtrage : Filtrer des emails, transférer des messages vers des destinations
- ...

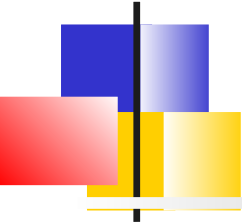


# Exemple (Modèle métier/ les faits)

---

```
public class Applicant {  
    private String name;  
    private int age;  
    // getter and setter methods here  
}
```

```
public class Application {  
    private Date dateApplied;  
    private boolean valid;  
    // getter and setter methods here  
}
```



# Exemple - Règles

---

```
package com.company.license
```

```
rule "Is of valid age"
```

```
when
```

```
    Applicant( age < 18 )
```

```
    $a : Application()
```

```
then
```

```
    $a.setValid( false );
```

```
end
```

```
rule "Application was made this year"
```

```
when
```

```
    $a : Application( dateApplied > "01-jan-2014" )
```

```
then
```

```
    $a.setValid( false );
```

```
end
```





# Pattern matching (Rete)

---

Lorsqu'une instance d'un *Applicant* est insérée dans le moteur, ce fait est évalué en regard des contraintes des règles.

Dans ce cas, les 2 contraintes de la première règle (contrainte sur le type et sur le champ âge).

Une contrainte sur un type d'objet et une ou plusieurs contraintes sur ses champs est appelé un **pattern**.

Lorsqu'une instance insérée satisfait le pattern, la **conséquence** de la règle s'exécute.

La notation ***\$a*** représente une **variable** permettant de référencer l'objet matché dans la conséquence. Ses propriétés peuvent être mises à jour dans la partie conséquence.

Le caractère ('\$') est optionnel mais rend l'expression de la règle plus lisible.



# Configuration

---

Il faut fournir un fichier *kmodule.xml* qui est placé dans le répertoire projet ;  
***ressources/META-INF/kmodule.xml***

Si on utilise la configuration par défaut, le fichier peut être vide :

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule"/>
```



# Création du conteneur

---

```
// Singleton des services Kie
KieServices kieServices = KieServices.Factory.get();
// Charger le module à partir du classpath
// Compile les fichiers drl trouvés
// Positionne le résultat compilé
KieContainer kContainer =
    kieServices.getKieClasspathContainer();
```



# Exécution

---

**// Création d'une session stateless**

```
StatelessKieSession ksession = kContainer.newStatelessKieSession();
```

```
Applicant applicant = new Applicant( "Mr John Smith", 16 );
```

```
Application application = new Application();
```

```
assertTrue( application.isValid() );
```

**// Exécution des règles pour ces 2 faits**

```
ksession.execute(  
    Arrays.asList( new Object[] { application, applicant } ) );
```

```
assertFalse( application.isValid() );
```



# *BatchExecutor et CommandFactory*

---

Les 2 méthodes *execute(Object object)* et *execute(Iterable objects)* sont en fait des raccourcis pour la méthode ***execute(Command command)*** de l'interface *BatchExecutor*

Par exemple, la seconde méthode est équivalent à :

```
ksession.execute(  
    CommandFactory.newInsertIterable( new Object[]  
        { application, applicant } ) ) ;
```

# *BatchExecutor et CommandFactory*

*BatchExecutor* et *CommandFactory* sont particulièrement utiles pour travailler avec plusieurs commandes et plusieurs identifiants de résultats :

```
List<Command> cmds = new ArrayList<Command>();  
  
cmds.add(CommandFactory.newInsert( new Person( "Mr John Smith" ), "mrSmith" );  
cmds.add(CommandFactory.newInsert( new Person( "Mr John Doe" ), "mrDoe" );  
  
BatchExecutionResults results =  
    ksession.execute( CommandFactory.newBatchExecution( cmds ) );  
  
assertEquals( new Person( "Mr John Smith" ), results.getValue( "mrSmith" ) );
```



# Prise en main

---

KIE API

Session stateless

**Session stateful et inférence**

Conflicts et Agenda

Listeners, Channels, entry-point



# Stateful Session

---

Les sessions **Stateful** ont une durée de vie plus longue et permettent des changements itératifs des faits.

Les cas d'utilisations classiques sont :

- Monitoring : Surveillance des marchés boursiers et achats semi-automatique
- Diagnostique : Découverte de faute, diagnostic médical
- Logistique : Suivi de colis, approvisionnement
- Conformité : Validation d'une législation





# Stateful versus stateless

---

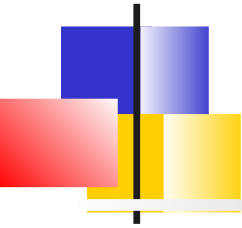
Contrairement à une Stateless Session, la méthode ***dispose()*** doit être appelée pour éviter les fuites mémoire (*KieBase* contient des références vers les Stateful Sessions)

Une *Stateful Knowledge Session* est tout simplement appelée ***KieSession***.

*KieSession* supporte également l'interface *BatchExecutor*

- Mais la commande ***FireAllRules*** n'est pas automatiquement déclenchée à la fin d'un session stateful.

# Exemple – Modèle métier



```
public class Room {  
    private String name  
    // getter and setter methods here  
}  
  
public class Sprinkler {  
    private Room room;  
    private boolean on;  
    // getter and setter methods here  
}  
  
public class Fire {  
    private Room room;  
    // getter and setter methods here  
}  
  
public class Alarm { }
```



# Exemple – Règle

---

```
rule "When there is a fire turn on the sprinkler"
```

```
when
```

```
    Fire($room : room)
```

```
    $sprinkler : Sprinkler( room == $room, on ==  
false )
```

```
then
```

```
    modify( $sprinkler ) { setOn( true ) };
```

```
    System.out.println( "Turn on the sprinkler for room  
" + $room.getName() );
```

```
end
```



# Inférence et *modify*

---

A la différence de l'exemple sur le *StatelessSession* qui utilisait la syntaxe Java pour modifier l'attribut d'un fait, l'utilisation de l'instruction ***modify*** permet d'avertir le moteur des changements effectués sur les faits et donc lui permettre d'effectuer d'autres raisonnements.

C'est ce qu'on appelle **l'inférence**



# Opérateur *not*

---

L'opérateur ***not*** permet de matcher lorsqu'aucune instance de l'objet n'existe dans la base de connaissance :

```
rule "When the fire is gone turn off the sprinkler"
when
    $room : Room( )
    $sprinkler : Sprinkler( room == $room, on == true )
    not Fire( room == $room )
then
    modify( $sprinkler ) { setOn( false ) };
    System.out.println( "Turn off the sprinkler for room " +
        $room.getName() );
end
```



# Opérateur *exists*

---

Son complément ***exist*** teste  
l'existence d'un objet :

```
rule "Raise the alarm when we have one or more  
fires"
```

```
when
```

```
    exists Fire()
```

```
then
```

```
    insert( new Alarm() );
```

```
    System.out.println( "Raise the alarm" );
```

```
end
```



# Instruction *delete*

---

L'instruction ***delete*** permet de retirer un fait de la base de connaissance

rule "Cancel the alarm when all the fires have gone"

when

not Fire()

\$alarm : Alarm()

then

***delete***( \$alarm );

System.out.println( "Cancel the alarm" );

end



# Inférence et maintenance

---

L'inférence est définie comme l'insertion d'un nouveau fait à partir d'une connaissance précédente :

```
rule "Infer Adult"  
when  
    $p : Person( age >= 18 )  
then  
    insert( new IsAdult( $p ) )  
end
```

L'inférence permet de découpler les responsabilités et d'encapsuler la connaissance.

=> systèmes beaucoup plus maintenables.

Par exemple, une autre règle pourra s'appuyer sur le fait d'être adulte plutôt que sur la valeur 18. Le changement de l'âge adulte sera alors facilité





# *insertLogical*

---

***insertLogical*** permet de rétracter le fait dès que la clause *when* redevient fausse.

Par exemple :

```
rule "Infer Child"
when
  $p : Person( age < 16 )
then
  insertLogical( new IsChild( $p ) )
end
```

Le fait *IsChild(\$p)* est automatiquement rétractée dès que la personne atteint 16 ans



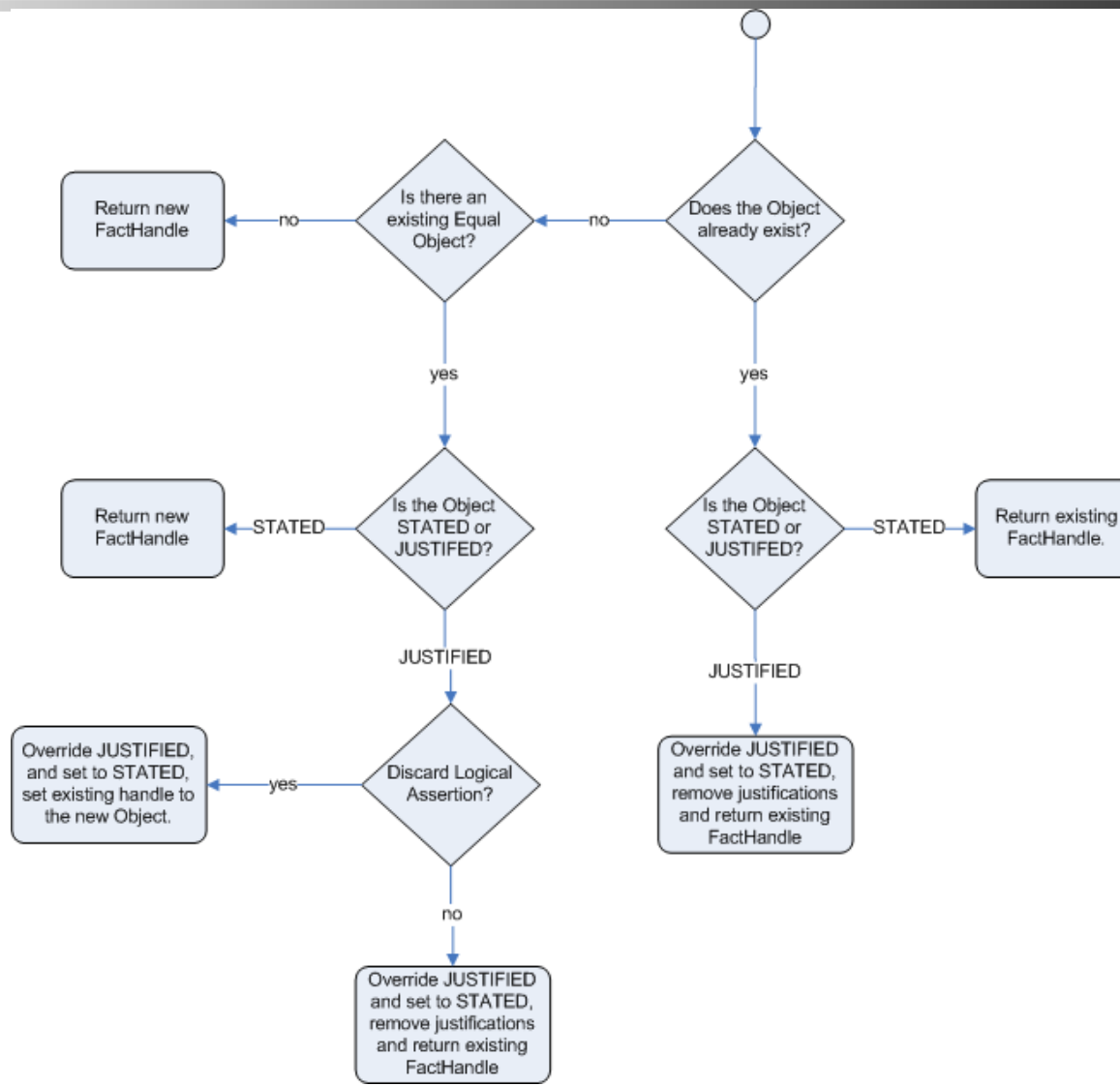
# Recommandations

---

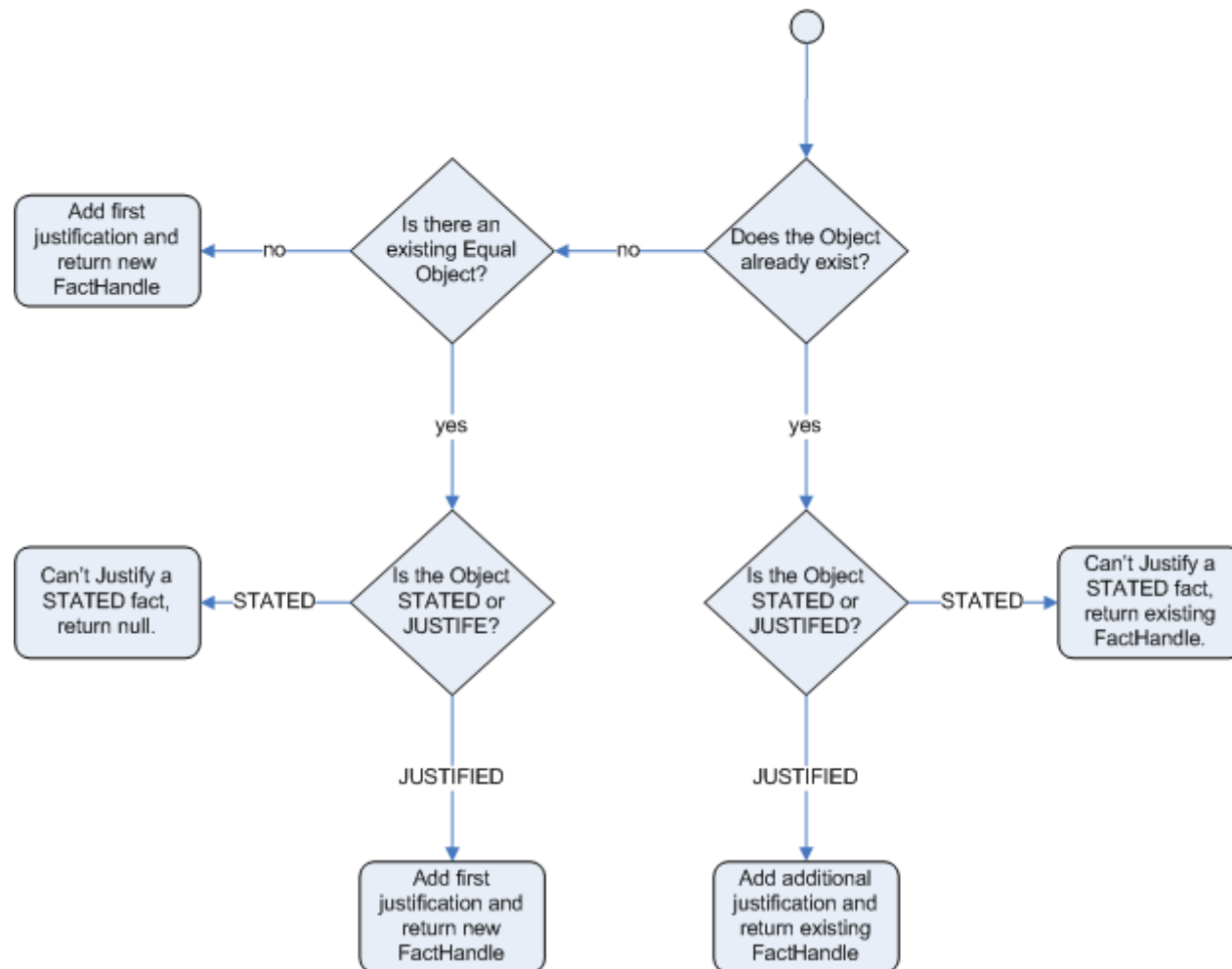
## Bonnes pratiques

- Dissocier les responsabilités liées aux connaissances
- Encapsuler les connaissances
- Fournir des abstractions sémantiques pour ces encapsulations

# Insertion simple (STATED)



# Insertion logique (JUSTIFIED)





# Egalité des faits

---

Détermine quand 2 faits sont identiques.

2 modes disponibles :

- ***identity***: (Default) Égalité basée sur l'identité de pointeur
- ***equality*** : Égalité basée sur la méthode *equals()*

```
<kbase name="KBase2" default="false"  
equalsBehavior="equality" packages="org.domain.pkg2,  
org.domain.pkg3" includes="KBase1">
```



# Création d'une session *stateful*

---

```
// Compile les fichiers drl trouvés dans le classpath
// Positionne le résultat compilé dans le conteneur
KieServices kieServices = KieServices.Factory.get();
KieContainer kContainer = kieServices.getKieClasspathContainer();
// Création de la session stateful
KieSession ksession = kContainer.newKieSession();
```



# Exécution

```
String[] names = new String[]{"kitchen", "bedroom", "office",  
    "livingroom"};  
Map<String,Room> name2room = new HashMap<String,Room>();  
for( String name: names ){  
    Room room = new Room( name );  
    name2room.put( name, room );  
    ksession.insert( room );  
    Sprinkler sprinkler = new Sprinkler( room );  
    ksession.insert( sprinkler );  
}
```

```
ksession.fireAllRules() ;
```

```
> Everything is ok
```



# *FactHandle*

Un **FactHandle** permet d'obtenir une référence sur un fait inséré.

```
Fire kitchenFire = new Fire( name2room.get( "kitchen" ) );  
Fire officeFire = new Fire( name2room.get( "office" ) );  
FactHandle kitchenFireHandle = ksession.insert( kitchenFire );  
FactHandle officeFireHandle = ksession.insert( officeFire );  
ksession.fireAllRules() ;
```

- > Raise the alarm
- > Turn on the sprinkler for room kitchen
- > Turn on the sprinkler for room office

Cette référence permettra de retirer le fait plus tard :

```
ksession.retract( kitchenFireHandle );  
ksession.retract( officeFireHandle );  
ksession.fireAllRules() ;
```

- > Turn on the sprinkler for room office
- > Turn on the sprinkler for room kitchen
- > Cancel the alarm
- > Everything is ok





# Prise en main

---

KIE API

Session stateless

Session stateful et inférence

**Conflits et Agenda**

Listeners, Channels, entry-point



# Méthodes versus Règles

---

## Méthodes :

- Elles sont appelées explicitement
- Des instances spécifiques sont passées en argument
- Un appel provoque une exécution unique

## Règles :

- Elles s'exécutent dès lors que des données correspondantes sont insérées dans le moteur (Drools5)
- Elles ne sont jamais appelées explicitement
- Des instances spécifiques ne peuvent pas être passées en argument
- Une règle peut se déclencher une fois, plusieurs fois ou aucune fois.



# Activations, Agenda et Conflits (Drools5)

---

Les faits sont évalués dès leurs insertions, cependant les règles ne sont déclenchées qu'au moment de l'appel à *fireAllRules()*.

Entre temps, les règles et les faits matchés sont placés dans l'agenda pour une exécution ultérieure.

Une entrée de l'agenda est appelée une **Activation**

L'agenda est donc une table des activations susceptibles d'être déclenchées lors de l'appel à *fireAllRules()*

L'ordre des activations est par défaut arbitraire, ainsi lorsqu'il y a plusieurs activations dans l'agenda, celles-ci sont dites en **conflit** et une **stratégie de résolution** de conflit doit être appliquée pour déterminer l'ordre des activations.



# Activations, Agenda et Conflits (Drools6)

---

Lors de l'appel à *fireAllRules()*, les règles sont évaluées indépendamment les unes des autres

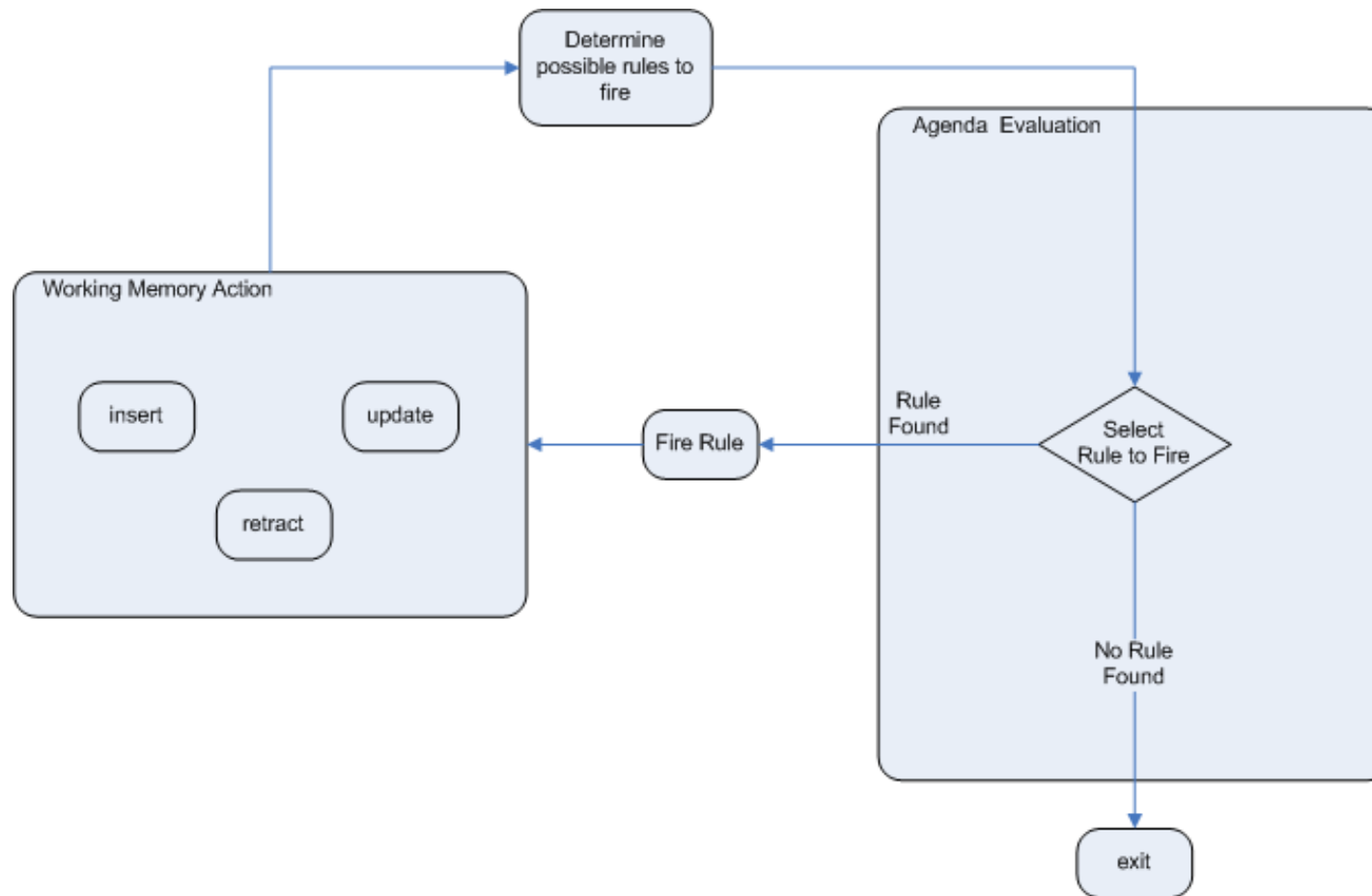
L'agenda choisit, parmi les règles activées, une seule règle en fonction du **groupe d'activation actif** et des attributs **salience** des règles.

Si 2 règles ont la même propriété , c'est l'ordre de déclaration qui joue

La règle est ensuite déclenchée.

Si la conséquence de la règle modifie un fait de la mémoire de travail, l'inférence se produit

# Cycle d'évaluation de l'Agenda





# Les groupes d'agenda

---

Les **groupes d'agenda** permettent de partitionner les règles dans des groupes.

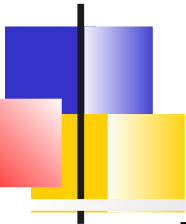
Les groupes sont gérés sous forme de pile

Au départ, la pile contient le groupe main

Lorsqu'un groupe est activé, il est placé au dessus de la pile

Lors d'un déclenchement de règles, l'agenda évalue les règles du groupe placées sur le haut de la pile

Lorsque plus aucune règle de groupe n'est activée, l'agenda dépile le groupe courant.



# Mode d'exécution des règles

---

Drools supporte différents modes d'exécution des règles :

- **Mode passif (défaut)** : Les règles sont déclenchées par ***fireAllRules()***  
Idéal dans la plupart des cas
- **Mode actif** : Lors de l'appel de ***fireUntilHalt()***, Drools évalue les règles en continu jusqu'au moment où la méthode ***halt()*** est appelée  
Application où les faits sont insérés régulièrement, ou on met en place des timer



# Filtre d'agenda

---

Lors de l'exécution des règles, Drools permet de positionner un ***AgendaFilter*** permettant de filtrer les règles que l'Agenda pourra activer.

```
ksession.fireAllRules( new RuleNameEndsWithAgendaFilter( "Test" ) );
```





# Prise en main

---

KIE API

Session stateless

Session stateful et inférence

Conflits et Agenda

**Listeners, Channels, entry-point**



# Event Model

---

Kie permet d'enregistrer des listeners à l'écoute d'événements.

- Dans le contexte de Drools les événements sont le déclenchement d'une règle, l'insertion d'un fait, etc.
- Cela permet de séparer les activités de trace ou d'audit, de l'expression des règles

Dans le contexte de Drools, 2 interfaces sont intéressantes:

- ***RuleRuntimeEventListener*** et ***AgendaEventListener***.



# Listeners de Debug

---

Drools fournit 2 listeners :

- *DebugRuleRuntimeEventListener* et *DebugAgendaEventListener* qui affiche des messages de debug

***KieRuntimeLogger*** utilise également le système événementiel pour générer un fichier de traces visible par la vue Audit d'Eclipse

```
KieRuntimeLogger logger =  
KieServices.Factory.get().getLoggers().newFileLogger(ksession,  
    "logdir/mylogfile");  
  
...  
logger.close();
```

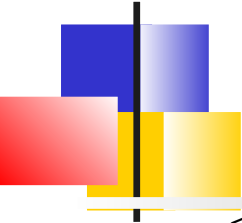


# Exemple de Listener personnalisé

---

```
ksession.addEventListener( new  
    DefaultAgendaEventListener() {  
public void afterMatchFired(AfterMatchFiredEvent event) {  
    super.afterMatchFired( event );  
    System.out.println( event );  
}  
});
```

```
ksession.addEventListener( new  
    DebugRuleRuntimeEventListener() );
```



# Configuration via *kmodule.xml*

```
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="KBase1" default="true" eventProcessingMode="cloud" equalsBehavior="equality">
    <ksession name="KSession2_1" type="stateful" default="true"/>
    <ksession name="KSession2_1" type="stateless" default="false" 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="stateful" default="false" clockType="realtime">
      <fileLogger file="drools.log" threaded="true" interval="10"/>
      <listeners>
        <ruleRuntimeEventListener type="org.domain.RuleRuntimeListener"/>
        <agendaEventListener type="org.domain.FirstAgendaListener"/>
        <agendaEventListener type="org.domain.SecondAgendaListener"/>
      </listeners>
    </ksession>
  </kbase>
</kmodule>
```



# Channels

---

Un **channel** est un moyen de transmettre des données de la mémoire de travail au monde extérieur.

- Alternative aux variables globales (Voir +loin).

Un *Channel* est l'implémentation d'une interface Java interface définissant une seule méthode :  
**void send(Object object)**

Les *Channels* ne peuvent être utilisées que dans la partie conséquence des règles



# Enregistrement

---

Les channels doivent être enregistrés  
programmatically :

```
ksession.registerChannel("audit-channel",  
auditChannel);
```

Ensuite, utilisés dans les règles :

```
rule "Send Suspicious Operation to Audit Channel"  
when  
$so: SuspiciousOperation()  
then  
channels["audit-channel"].send($so);  
end
```



# Points d'entrée

---

Les **entry points** sont un moyen de partitionner la mémoire de travail. Les règles peuvent alors s'appliquer sur certains entry point

// **Raisonnement à partir d'un point d'entrée**

```
rule "Routing transactions from small resellers"
```

```
when
```

```
    t: TransactionEvent()
```

```
        from entry-point "small resellers"
```

A l'insertion, le point d'entrée peut être spécifié:

```
ksession.getEntryPoint("myEntryPoint").insert(new Object());
```





# Le langage de règle : DRL

---

## **Principaux éléments**

Attributs des règles

LHS

RHS

Requête



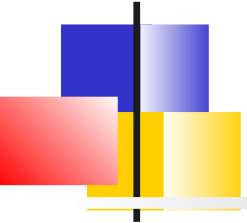
# Fichier de règles

Un fichier de règles est un simple fichier texte avec l'extension **.drl**,

Sa structure est la suivante :

- **package** : Espace de nom
- **imports** : Types Java utilisés
- **declare** : Déclaration interne de nouveaux types
- **globals** : Variables globales accessibles de l'extérieur
- **functions** : Réutilisation de logique
- **queries** : Recherche de faits
- **rules** : Règles

Il est également possible de répartir les règles sur plusieurs fichiers qui ont alors généralement l'extension **.rule**



# Structure d'une règle

---

```
rule "name"  
  attributs  
  when  
    LHS  
  then  
    RHS  
end
```

La ponctuation ", les retours à la ligne sont optionnels

Les attributs sont optionnels

LHS est la partie conditionnelle de la règle

RHS est un bloc qui permet de spécifier en différents dialectes un code à exécuter.



# Mots-clés

---

## Mots-clés :

- Hard (*true, false, accumulate, collect, from, null, over, then, when*). Ne peuvent pas être utilisés comme identifiant d'objets
- Soft : seulement reconnu dans leur contexte (*package, import, attributes, rule, ...*)

Le caractère **d'échappement** est ` :

*Holiday( `when` == "july" )*

## Commentaires :

- ligne : **#** ou **//**
- multi-lignes : **/\* .... \*/**



# Message d'erreur

[ERR 101] Line 6:35 no viable alternative at input ']' in rule "test rule" in pattern WorkerPerformanceContext

1st Block    2nd Block    3rd Block    4th Block    5th Block

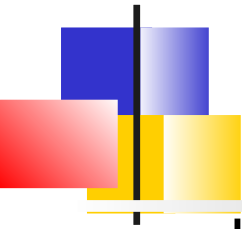
1er bloc : **Code d'erreur**

2nd bloc : Information de **colonne et ligne**.

3ème bloc : **Description** de l'erreur.

4ème bloc optionnel : Premier **contexte** de l'erreur.  
Généralement, la règle, la fonction, la requête ou l'erreur s'est produite.

5ème bloc optionnel : Identifie le **pattern** où l'erreur s'est produite.

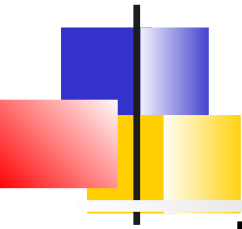


# Package

---

Un package regroupe un ensemble de règles, d'imports et de globales qui sont en relation.

- Un package représente un espace de nom unique. Il suit les conventions de nommage des packages Java
- Il est possible de répartir les règles d'un même package dans plusieurs fichiers. Dans ce cas, un seul fichier contient la configuration du package.
- Les éléments déclarés dans un package peuvent être dans n'importe quel ordre, à l'exception de l'instruction *package*
- Les ';' sont optionnels.



# Import

---

Les instructions **import** sont similaires aux *imports* dans Java.

Il est nécessaire de spécifier le nom complet des types d'objets que l'on utilise dans les règles.

Drools importe automatiquement les classes Java du package du même nom ainsi que le package *java.lang*.



# Global

---

Avec le mots-clé **global**, on définit des variables globales.

- Peuvent être utilisées dans les conséquences des règles.
- Ne sont pas insérées dans la mémoire de travail  
=> Ne doivent donc être utilisées dans LHS sauf en tant que constante
- Le moteur n'est pas prévenu lors d'un changement de valeur d'une variable globale





# Exemple

---

```
global java.util.List myGlobalList;
```

```
rule "Using a global"  
when  
  eval( true )  
then  
  myGlobalList.add( "Hello World" );  
end
```

-----

```
List list = new ArrayList();  
WorkingMemory wm = rulebase.newStatefulSession();  
wm.setGlobal( "myGlobalList", list );
```



# Fonctions

---

Les **fonctions** permettent d'insérer du code dans les fichiers de règles.

- Elles ne peuvent rien faire de plus que des classes *Helper*.  
Le seul avantage est de centraliser la logique à un seul endroit.
- Elles sont utilisées pour invoquer des actions dans la partie conséquence des règles .



# Exemple

---

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

```
rule "using a static function"  
when  
    eval( true )  
then  
    System.out.println( hello( "Bob" ) );  
end
```



# Déclarations

---

2 types de déclarations sont possibles :

- La déclaration de **nouveaux types** :  
Il est possible de définir le modèle métier directement dans les règles ou de créer des objets modèles qui n'ont d'utilité que lors du raisonnement.  
Au moment de la compilation, Drools génère le bytecode Java correspondant
- La déclaration de **méta-données** ou **annotations** :  
Les faits ou leurs attributs peuvent être annotés.  
Les annotations peuvent être utilisées lors du raisonnement ou lors de la recherche de fait.



# Déclaration de nouveaux types

---

```
declare Address
```

```
    number : int
```

```
    streetName : String
```

```
    city : String
```

```
end
```

```
declare Person
```

```
    name : String
```

```
    dateOfBirth : java.util.Date
```

```
    address : Address
```

```
end
```



# Accès aux types déclarés

---

Il est possible d'accéder aux type déclarés à partir du code applicatif via l'interface

***org.drools.definition.type.FactType***

```
// Récupérer le type de données
```

```
FactType personType = kbase.getFactType( "org.drools.examples",  
                                           "Person" );
```

```
// Instancier
```

```
Object bob = personType.newInstance();
```

```
// Positionner les attributs
```

```
personType.set( bob, "name", "Bob" );
```

```
personType.set( bob, "age", 42 );
```



# Déclaration de méta-données

---

Le caractère **@** est utilisé. Les méta-données peuvent concerner le type ou un de ses attributs, un nouveau type ou un type existant.

```
declare Person
  @author( Bob )
  @dateOfCreation( 01-Feb-2009 )

  name : String @key @maxLength( 30 )
  dateOfBirth : Date
  address : Address
end
```

Ou sur un type existant

```
declare Person
  @author( Bob )
  @dateOfCreation( 01-Feb-2009 )
end
```



# Utilisation des méta-données

---

Les méta-données peuvent être utilisées par les filtres ou par les requêtes<sup>1</sup>.

Certaines méta-données sont prédéfinies et ont un sens pour le moteur. Elles sont surtout utile pour *Drools-fusion* :

*@role, @timestamp, @duration, ...*

1. Les requêtes permettent de sélectionner un sous-ensemble de faits de la mémoire de travail répondant à certains critères. Les filtres permettent de sélectionner un sous-ensemble de règles





# Exemple d'utilisation des méta-données

---

```
StatefulKnowledgeSession ksession= createKSession();
```

```
ksession.fireAllRules(new AgendaFilter() {  
    public boolean accept(Activation activation) {  
        Map<String, Object> metaData =  
            activation.getRule().getMetaData();  
        if (metaData.containsKey("LegalRequirement")) {  
            return true;  
        }  
        return false;  
    }  
});
```



# Le langage de règle : DRL

---

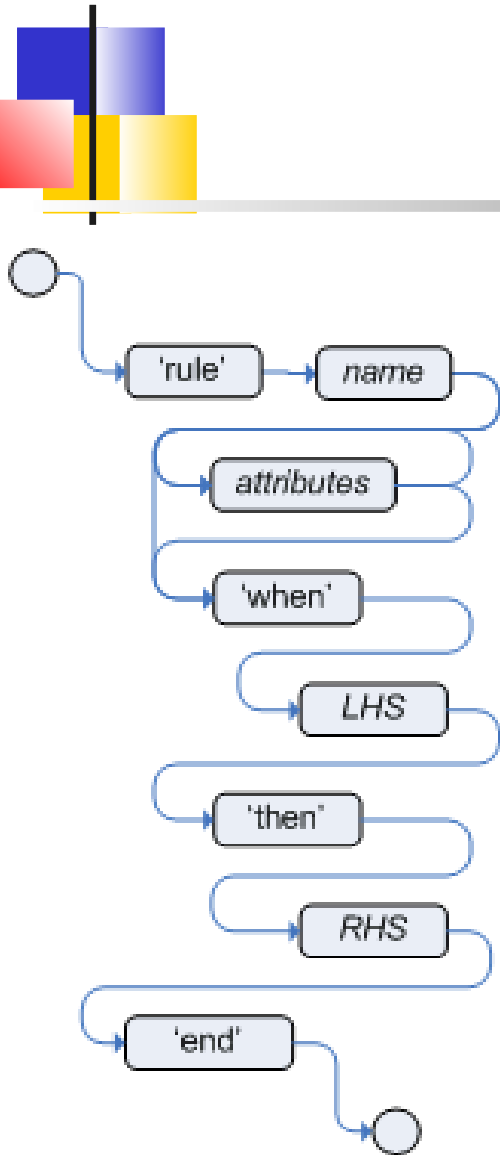
Principaux éléments  
**Attributs des règles**

LHS

RHS

Requête

# Règle



- ✓ Doit avoir un nom, unique à l'intérieur du *package*.
- ✓ Le nom peut contenir des espaces si il est délimité par ".
- ✓ La partie gauche de la règle (*Left Hand Side*) ou condition suit le mot-clé **when**
- ✓ La partie droite (*Right Hand Side*) ou conséquence suit le mot-clé **then**
- ✓ La règle se termine par le mot-clé **end**.
- ✓ Les règles ne peuvent pas être imbriquées.



# Attributs des règles (1)

---

***no-loop (booléen, false)*** : Lorsque la conséquence de la règle modifie un fait, il peut provoquer l'activation de la règle une nouvelle fois. La récursion peut donc être évitée avec l'attribut *no-loop* à *true*.

***salience (Entier, 0)*** : Chaque règle a un attribut *salience* de type entier qui détermine la priorité de la règle dans l'agenda.

***dialect (String, "java" or "mvel")*** : Le dialecte est en général spécifié au niveau du package. Cet attribut surcharge la définition au niveau package.



# Attributs des règles (2)

---

***agenda-group (String, MAIN)*** : Cet attribut permet de partitionner l'Agenda et de contrôler l'exécution. Seules les règles du groupe d'agenda qui a acquis le focus sont autorisées à se déclencher.

***activation-group (String)*** : Les règles appartenant au même groupe d'activation sont exclusives. La première règle qui se déclenche annule les autres.

***ruleflow-group (String)*** : Permet de grouper plusieurs règles. Les règles de ce groupe ne seront activées que lorsque le processus jBPM associé sera dans un nœud particulier.



# Attributs des règles (3)

---

***auto-focus (booléen, false)*** : Quand une règle est activée avec l'attribut *auto-focus* positionné, le groupe indiqué par un de ses attributs *agenda-group* ou *activation-group* gagne le focus.

***lock-on-active (booléen, false)*** : Lorsqu'un groupe (*ruleflow* ou *agenda*) devient actif, toutes les règles de ce groupe qui ont l'attribut *lock-on-active* positionné ne seront plus activées dans le futur quelque soit l'origine de la mise à jour. Elles pourront être réactivées à la réactivation de leur groupe.

***date-effective (String définissant une date)*** : Une règle ne peut être activée que si la date courante est supérieure à la date effective.

***date-expires (String définissant une date)*** : Une règle ne peut plus être activée que si la date courante est supérieure à la date d'expiration.



# Timer

Drools supporte les *timers* basés sur des intervalles ou exprimés par des expressions *cron*. Les *timers* remplacent l'attribut *duration*.

```
timer ( int: <initial delay> <repeat interval>? )
```

```
timer ( cron: <cron expression> )
```

## Exemple

```
rule "Send SMS every 15 minutes"
    timer (cron:* 0/15 * * * ?)
when
    $a : Alarm( on == true )
then
    channels[ "sms" ].insert( new Sms( $a.mobileNumber, "The alarm is still on" );
end
```



# *fireUntilHalt()*

---

Afin que les règles utilisant des *timers* puisse être déclenchées, le moteur doit être actif.

Dans ce cas, il ne faut pas appeler *fireAllRules()* mais ***fireUntilHalt()*** qui évalue les règles tant qu'il ne reçoit pas de signal *halt*

Dans ce cas, arrêter le moteur peut se faire :

- Via une règle : ***drools.halt()***
- Via Java : ***ksession.halt()***

Dans ce cas, la méthode *fireUntilHalt* est généralement lancé dans une thread indépendante afin que le code Java puisse l'arrêter





# Le langage de règle : DRL

---

Principaux éléments  
Attributs des règles

**LHS**

RHS

Requête



# LHS

---



La partie LHS est la partie conditionnelle de la règle.  
Elle est constituée de zéro ou plusieurs éléments conditionnels

- Si aucune condition, la LHS est évaluée à *true* et sera activée
- Les éléments conditionnels sont constitués de ***patterns*** qui sont implicitement reliés par une logique ***and***

# Pattern



Un pattern consiste en :

- Un pattern binding permettant de créer une **variable** utilisée dans la règle, le caractère **\$** est facultatif mais recommandé
- Le caractère **:**
- Une restriction sur le **type** (Un fait, une interface, une classe abstraite)
- Un ensemble de **contraintes** reliées par des opérateurs

Ex : \$c : Cheese()



# Syntaxe des contraintes

---

2 types de syntaxe sont disponibles :

- Les contraintes de **champ**

- Portent sur un seul attribut
- Sont combinés avec les opérateurs && et || et ()

- Ex: Cheese( quantity == 5 && quantity < 10 )

- Les contraintes **groupes**

- Portent sur plusieurs attributs d'un même fait
- Sont combinées avec l'opérateur ,

- Ex: Person( age > 50, weight > 80, height > 2 )

# Contraintes de champ



3 types de restriction sont possibles :

- **Valeur unique** : le champ est comparé à une seule valeur
- **Plusieurs valeurs** : le champ est comparé à plusieurs valeurs
- **Multi-contraintes** : Plusieurs contraintes sont spécifiées sur le champ

La valeur du champ peut être de type String, numérique, date (format "dd-mmm-yyyy" par défaut), *boolean* ou *Enum* des tests sur *null* peuvent être effectués ainsi que sur des retours de méthode



# Contrainte simple valeur

---

Les opérateurs pouvant être utilisés sont : `<`, `<=`, `>`, `>=`,  
`==`, `!=`, *contains*, *not contains*, *memberof*, *not memberof*,  
*matches (expr. régulière)*, *not matches*

```
Cheese( quantity == 5 )
```

```
Cheese( bestBefore < "27-Oct-2009" )
```

```
Cheese( type == "camembert" )
```

```
Cheese( from == Enum.COW)
```

```
Cheese( type matches "(Buffalo)?\\S*Mozarella" )
```

```
CheeseCounter( cheeses contains "stilton" )
```

```
CheeseCounter( cheese memberOf $matureCheeses )
```

```
Person( likes : favouriteCheese ) Cheese( type == likes )
```

```
Person( girlAge : age, sex == "F" ) Person( age == ( girlAge + 2), sex ==  
    'M' )
```



# Contrainte valeurs multiples

---

Les opérateurs *in* et *not in*, permettent de spécifier plusieurs valeurs séparées par ","

```
Person( $cheese : favouriteCheese )
```

```
Cheese( type in ( "stilton", "cheddar", $cheese ) )
```



# Contraintes multiples

---

Les contraintes multiples permettent de spécifier plusieurs restrictions sur le champ reliées par les opérateurs '&&' ou '||' et des parenthèses

```
Person( age > 30 && < 40 )
```

```
Person( age ( (> 30 && < 40) ||  
              (> 20 && < 25) ) )
```

```
Person( age > 30 && < 40 || location == "london" )
```





# Contraintes groupes

---

La virgule **,** permet de séparer les contraintes de groupes et équivaut à un AND moins prioritaire :

```
Person( age > 50, weight > 80, height > 2 )
```

L'opérateur virgule ne peut pas être imbriqué dans une expression composite :

```
Person( ( age > 50, weight > 80 ) || height > 2 )  
// => ERREUR de compilation
```



# Autres opérateurs

---

***not*** : Il n'y a aucun fait dans la mémoire de travail correspondant à ces restrictions

***exists*** : Il y a au moins un fait dans la mémoire de travail correspondant à ces restrictions

***forall*** : Tous les faits de la mémoire de travail correspondant à la première restriction satisfont les autres restrictions

***from*** : Permet d'exprimer des contraintes sur un sous-ensemble de la mémoire de travail (Exemple requête)

***collect*** : Permet de raisonner sur une collection d'objets

***accumulate*** : Permet d'effectuer une fonction d'agrégation sur une collection d'objets



# Exemples

---

**#Il n'y a aucun bus de couleur rouge dans la mémoire**

```
not Bus(color == "red")
```

**#Il y a au moins un bus 42 de couleur rouge dans la mémoire**

```
exists ( Bus(color == "red", number == 42) )
```

**#Tous les bus anglais sont rouge**

```
forall( $bus : Bus( type == 'english')  Bus( this == $bus, color =  
    'red' ) )
```

**# Les adresses avec le bon code postal**

**# qui sont détenues par des personnes de la mémoire**

```
Person( $personAddress : address )
```

```
Address( zipcode == "23920W") from $personAddress
```



# Exemples

---

**# Construction d'une liste de maman**

**# Toutes les personnes de sexe féminin ayant des enfants**

```
$mothers : LinkedList()
```

```
from collect( Person( gender == 'F', children > 0 ) )
```

**# Toutes les commandes dont le total est supérieur à 100**

```
$order : Order()
```

```
$total : Number( doubleValue > 100 )
```

```
from accumulate( OrderItem( order == $order, $value : value ),  
    sum( $value ) )
```



# Le langage de règle : DRL

---

Principaux éléments

Attributs des règles

LHS

**RHS**

Requête



# RHS

---

La partie droite contient une liste d'actions à exécuter.

En général, pas de code conditionnel car la règle doit être « atomique » (sinon séparer en plusieurs règles)

Les opérations peuvent agir sur la mémoire de travail et donc déclencher l'inférence :

- Ajout de faits
- Suppression de faits
- Modification de faits



# Macro-méthodes

Drools propose plusieurs macro-méthodes évitant de récupérer les références des faits que l'on désire mettre à jour :

- ***set : set<field> ( <value> )***

Utilisé pour mettre à jour un champ

```
$application.setApproved ( false );
```

```
$application.setExplanation( "has been bankrupt" );
```

- ***modify : modify ( <fact-expression> ) {  
    <expression>,  
    <expression>,  
    ... }***

Utilisé pour spécifier les champs à modifier et notifier Drools du changement

```
modify( LoanApplication ) {  
    setAmount( 100 ),  
    setApproved ( true )  
}
```



# Macro-méthodes (2)

---

- **update :**  
**update ( <object, <handle> )**  
**update ( <object> )** // => Rechercher le fait correspondant  
Utilisé pour spécifier le fait à mettre à et pour notifier Drools du changement.  
`LoanApplication.setAmount( 100 );`  
`update( LoanApplication );`
- **insert: insert( new <object> );**  
Utilisé pour insérer un nouveau fait  
`insert( new Applicant() );`
- **insertLogical : insertLogical(new Something())**  
l'objet est automatiquement supprimé si la règle n'est plus valable.  
`insertLogical( new Applicant() );`





# Macro-méthodes (3)

---

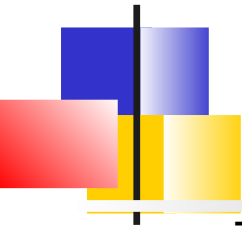
- **delete : delete( <object> )**  
Utilisé pour supprimer un objet de la mémoire de travail. Le mot-clé *retract* est également supporté  
delete( Applicant )



# *drools et RuleContext*

La variable prédéfinie **drools** de type *RuleContext* permet d'appeler d'autres méthodes utiles :

- Méthodes donnant des informations sur la règle
  - **drools.getRule().getName()** : Le nom de la règle activé
  - **drools.getMatch()** : Information sur pourquoi la règle a été activé
- Avoir une référence sur KieRuntime, on peut également utiliser
  - **drools.getKieRuntime().halt()** : Arrêter le moteur en mode actif
  - **drools.getKieRuntime().getAgenda()** : Accès à l'agenda puis aux groupes de règles  
drools.getKieRuntime().getAgenda().getAgendaGroup( "CleanUp" ).setFocus();
  - **drools.getKieRuntime().setGlobal()** : Positionner une globale
  - **drools.getKieRuntime().getQueryResults(<string> query)** : Retourner le résultat d'une requête



# *kcontext et KieRuntime API*

Toute l'API de la base de connaissance est également exposée via la variable prédéfinie **kcontext** de type *KieContext*.

Sa méthode **getKieRuntime()** retourne un objet de type *KieRuntime* :

- **getAgenda()** retourne une référence sur l'agenda de la session
- **getQueryResults(String query)** retourne le résultat d'une requête
- **addEventListener, removeEventListener** : enregistrement de listeners pour la mémoire de travail ou l'agenda.
- **getKnowledgeBase()** retourne l'objet *KnowledgeBase* .
- Gestion des variables **globales** avec *setGlobal(...)*, *getGlobal(...)* et *getGlobals()*.
- **getEnvironment()** : l'environnement d'exécution et ses propriétés



# Le langage de règle : DRL

---

Principaux éléments

Attributs des règles

LHS

RHS

**Requête**



# Introduction

---

Une requête, dans Drools, peut être considérée comme une règle sans sa section de droite.

Cependant, une différence majeure consiste en le fait qu'une requête peut prendre des arguments



# Définition de requête

---

Une **requête** permet de rechercher des faits présents dans la base de connaissance

Une requête peut être paramétrée.

Les noms des requêtes sont globales à la base de connaissance  
=> Pas de nom identique même dans des packages différents

Sa définition est identique à la partie gauche d'une règle :

```
query "people over the age of x" (int x)
```

```
    person : Person( age > x )
```

```
end
```



# Requête à la demande

---

Une requête peut être évaluée à la demande via la méthode ***getQueryResults*** de KieSession

```
public QueryResults getQueryResults(String query,  
                                     Object... arguments);
```

Il est ensuite possible d'itérer sur les lignes retournées

```
QueryResults results = ksession.getQueryResults( "people over the age of x" ,30 );  
System.out.println( "we have " + results.size() + " people over the age  of 30" );  
  
System.out.println( "These people are are over 30:" );  
for ( QueryResultsRow row : results ) {  
    Person person = ( Person ) row.get( "person" );  
    System.out.println( person.getName() + "\n" );  
}
```



# Requêtes *live*

---

Drools permet également d'attacher un listener à une requête afin d'être informé des résultats dès qu'ils sont disponibles

Ce sont les requêtes *live*, elles sont exécutées via la méthode ***openLiveQuery()***

```
public LiveQuery openLiveQuery(String query,  
                                Object[] arguments,  
                                ViewChangedEventListener listener);
```





# Interface *ViewChangeListener*

---

L'interface ***ViewChangeListener*** comporte 3 méthodes :

```
public interface ViewChangeEventListner {  
    public void rowInserted(Row row);  
    public void rowDeleted(Row row);  
    public void rowUpdated(Row row);  
}
```

L'interface permet donc d'être prévenu lors d'insertion, la mise à jour et la suppression des faits respectant la requête



# Performance

---

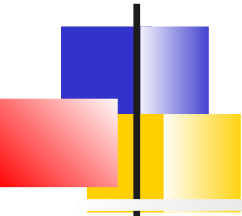
## **Réseau Rete/ PHREAK**

Backward chaining

Améliorations PHREAK

drools-metric

# Algorithme de Rete et ReteOO



Le traitement de comparer les faits aux règles est appelé le **Pattern Matching**, il existe de nombreux algorithmes de pattern matching : *Linear, Rete, Treat, Leaps*.

Drools a dans un premier temps implémenté l'algorithme de Rete dans un technologie objet : **ReteOO**

- L'algorithme est particulièrement efficace lorsque le jeu de données (les faits) change régulièrement mais dans de faibles proportions car le moteur de règle se souvient des règles ayant déjà matchée
- Il est gourmand en mémoire



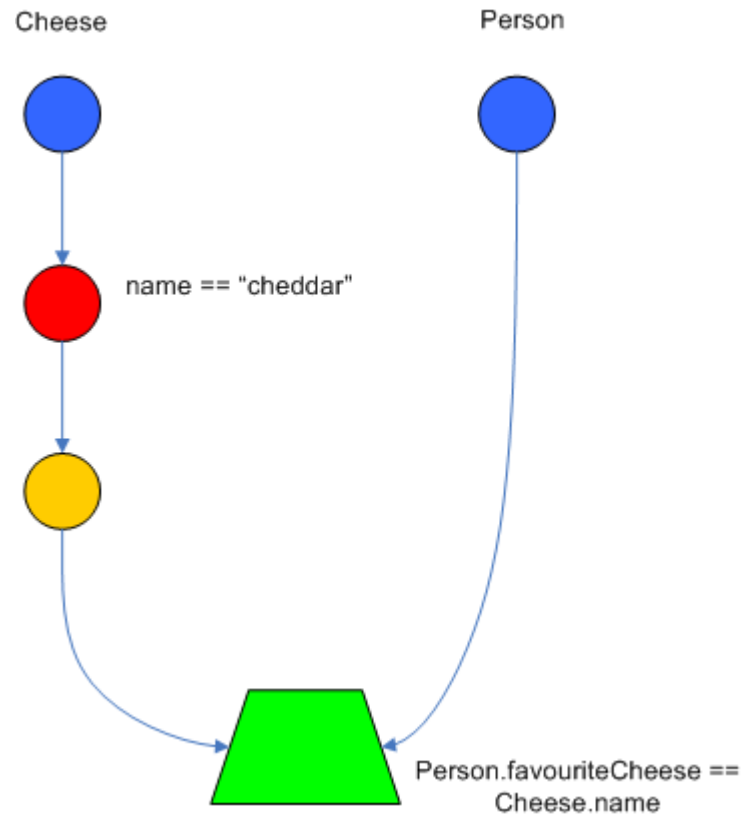
# Principes de Rete

---

L'algorithme de Rete repose sur 2 étapes :

- La compilation des règles crée un réseau discriminant constitué de différents nœuds :
  - **TypeNode** : Relatif à un type de fait
  - **Alpha** : Contrainte sur un fait
  - **Beta** : Comparaison entre des tuples d'objets et un objet unique.
  - **Règle** : Si une donnée atteint ce type nœud la règle est déclenchée
- Le pattern matching. Il s'exécute à chaque modification de la base de fait. Les faits sont introduits dans le réseau et traverse les nœuds si les conditions sont satisfaites. Certains nœuds (*Beta*) ont une mémoire associée et collecte les faits qui parviennent au nœud.

# Example





# Inconvénients de Rete

---

Rete est un algorithme *eager*

Il effectue de nombreuses actions lors des actions d'insertion, de mise à jour et de suppression afin de trouver des correspondances partielles pour toutes les règles.

=> Cela nécessite beaucoup de temps avant d'exécuter éventuellement les règles, en particulier dans les grands systèmes.



# PHREAK

---

A partir de Drools 6, l'algorithme PHREAK est utilisé

C'est un algorithme *lazy*

La mise en correspondance partielle des règles est délibérément retardée pour traiter plus efficacement de grandes quantités de données.

*PHREAK* est équivalent à *ReteOO* lorsque le nombre de règles reste modéré mais permet d'éviter des pertes de performances lorsque le nombre de règles grossit

*PHREAK* permet également des gains de performance lors de l'utilisation de groupes d'agenda et des attributs de priorité (*salience*) des règles



# Évaluations des règles avec PHREAK

---

Lorsque le moteur Drools démarre, toutes les règles sont considérées comme ***unlinked***

Les actions d'insertion, de mise à jour et de suppression de faits sont mises en file d'attente et Phreak utilise une heuristique, pour sélectionner la règle la plus susceptible d'être exécutée.

Lorsque toutes les valeurs d'entrée requises sont renseignées pour une règle (Contraintes sur les types de donnée), la règle est considérée comme étant ***linked*** aux données correspondantes.

Phreak crée ensuite un ***objectif (goal)*** représentant cette règle et le place dans une file d'attente prioritaire triée par l'attribut *salience*.

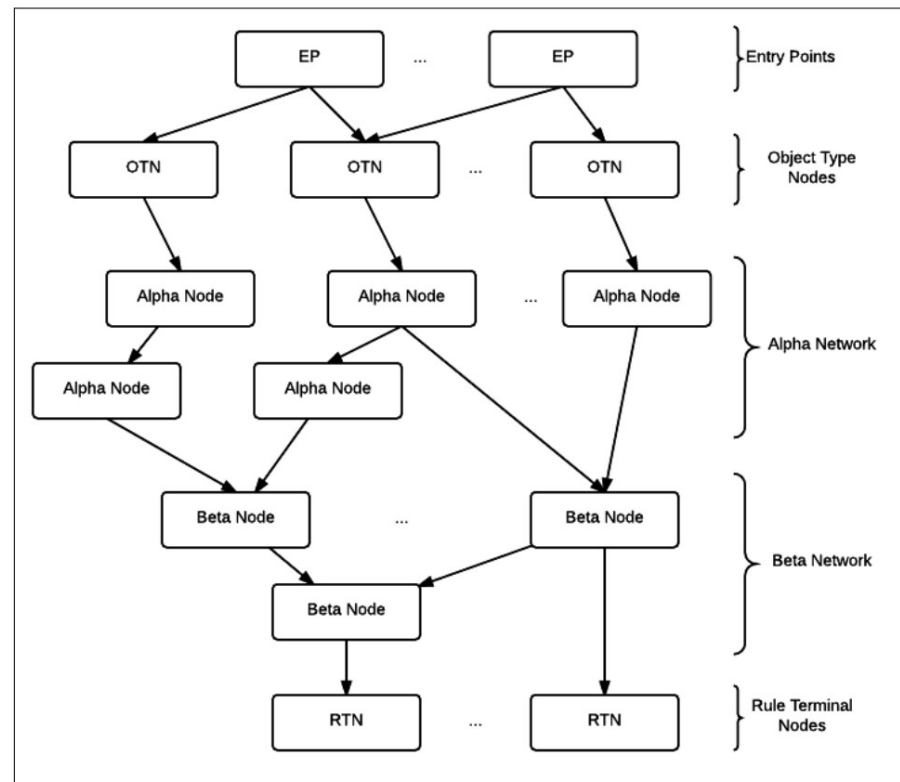
Seule la règle pour laquelle l'objectif a été créé est évaluée, et les autres évaluations de règle potentielles sont retardées.



# Réseau de règles

PHREAK construit également un réseau Rete à partir des règles

Le réseau est constitués de différents nœuds





# ObjectType Node

---

Un **Object Type Node (OTN)** est une contrainte sur le type du fait *~ instanceof*

=> Le réseau PHREAK contient autant d'OTN que de classes distinctes utilisées dans les règles

=> Les règles utilisant la même classe partagent le même nœud OTN.

=> Lorsque le nœud est évalué toutes les règles qui s'y rapportent sont évaluées en même temps

=> Le sous-réseau d'OTN est toujours présent dans PHREAK et sa profondeur est toujours de 1



# Alpha Node

---

Un pattern peut contenir plusieurs ou aucune contrainte.

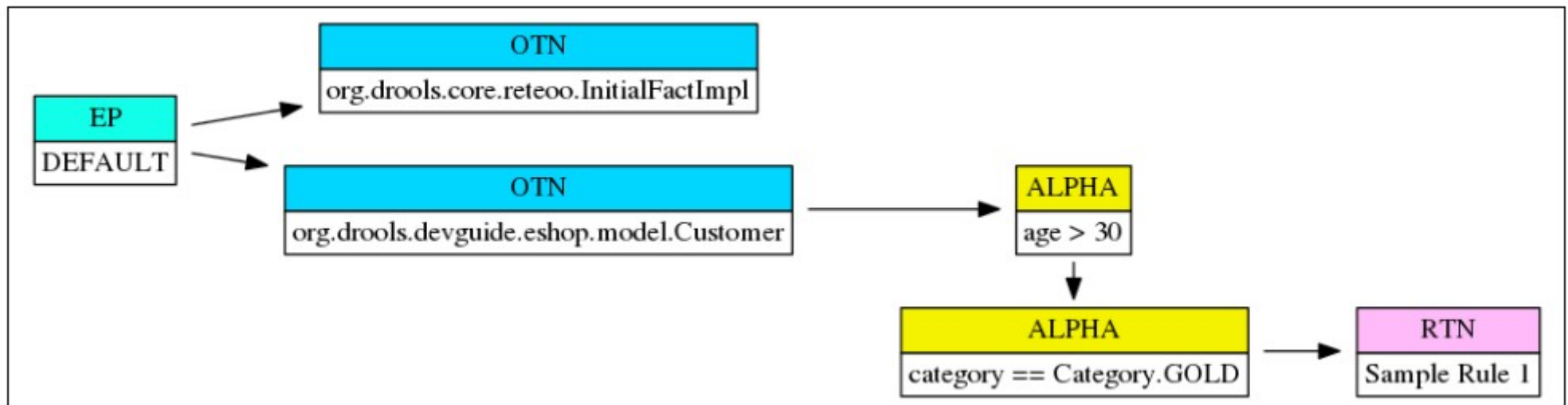
Chaque contrainte individuelle est représentée dans le réseau PHREAK par un **nœud alpha**.

Le nœud est en charge de l'évaluation de la contrainte particulière qu'il représente.

Si la contrainte est évaluée à *true* , le prochain nœud du réseau sera évalué

# Exemple *AlphaNode*

```
rule "Sample Rule 1"  
when  
    $c: Customer(age > 30, category == Category.GOLD)  
then  
    channels["customer-channel"].send($c);  
end
```





# Ordre des contraintes

---

L'ordre des Alpha Nodes dépend de l'ordre dans lequel les contraintes correspondantes sont définies dans DRL

Comme pour les OTNs, les nœuds alpha peuvent être partagés entre plusieurs règles.

Si la même contrainte est utilisée dans plusieurs pattern, Drools optimisera la création du réseau PHREAK et un seul nœud Alpha sera utilisé.

=> Attention, afin que le nœud Alpha puisse être partagé, l'ordre de la contrainte doit être le même dans tous les patterns

Rq : Voir également compilation JIT des expressions des contraintes.

Propriétés : *ConstraintJittingThresholdOption*



# Beta Nodes

---

Un **nœud Beta** représente l'opération *join* sur 2 patterns

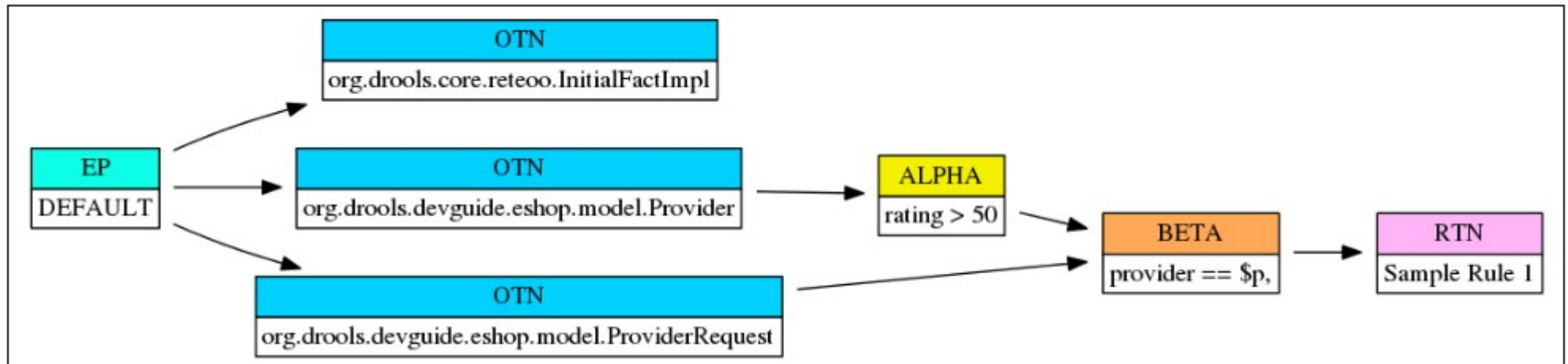
- Il a 2 entrées et 1 ou plusieurs sorties
- Il peut préciser une contrainte supplémentaire sur les 2 entrées

Les nœuds Beta peuvent également être partagés ... si l'ordre des patterns ET des contraintes sont identiques dans plusieurs règles

=> Pas toujours facile à obtenir

# Example

```
rule "Sample Rule 1"
when
    $p: Provider(rating > 50)
    $pr: ProviderRequest(provider == $p)
then
    channels["request-channel"].send($pr);
end
```





# Autres nœuds

---

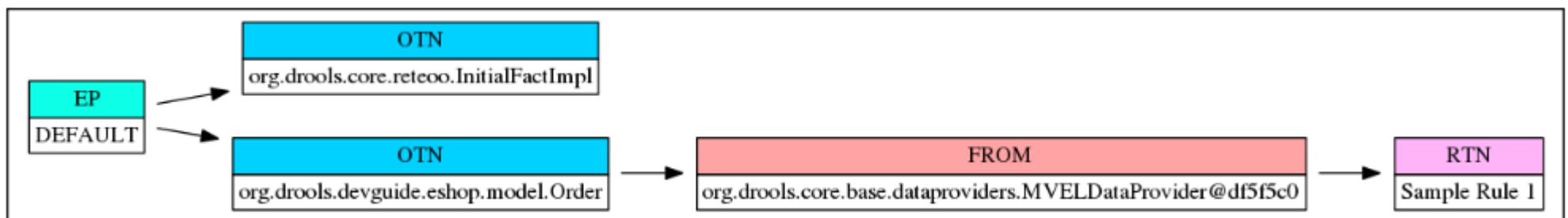
Il existe d'autres nœuds dans le réseau PHREAK correspondant aux éléments conditionnels *not*, *exists*, *accumulate* et *from*

- *NotNode*, *ExistsNode*, *AccumulateNode* : BetaNode spécialisés
- *fromNode* : Plus étrange, la partie droite et gauche de *from* sont évalués



# Exemple *fromNode*

```
rule "Sample Rule 1"
when
    $o: Order()
    $ol: OrderLine(
        item.category == Category.HIGH_RANGE,
        quantity > 10) from $o.getOrderLines()
then
    channels["audit-channel"].send($ol);
end
```





# Performance

---

Réseau Rete/ PHREAK  
**Backward chaining**  
Améliorations PHREAK  
*drools-metric*



# Chaînage avant / arrière

---

Les moteurs de règles sont à chaînage avant ou arrière :

- Le **chaînage avant** est piloté par les données : à partir d'un fait, des règles s'appliquent, se propagent et se terminent par une conclusion.
- Le **chaînage arrière** part de la conclusion pour remonter aux causes  
Dans Drools, Il est implémenté via les requêtes

Drools 5 était un moteur à chaînage avant.

Drools 6 est devenu un moteur hybride (avant et arrière)

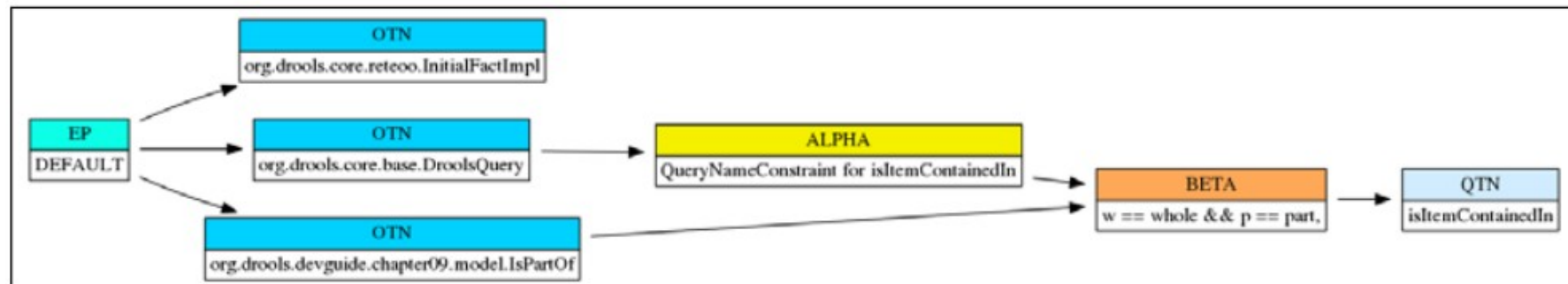
# Noeud d'une requête

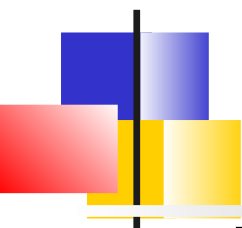
Les requêtes ont leur représentation dans le réseau PHREAK :

- Un OTN sur DroolsQuery
- Un Alpha sur le nom de la query
- Un Beta sur la condition de la requête
- Un nœud de sortie QTN

Exemple :

```
query isItemContainedIn(Item p, Item w)
  IsPartOf(whole == w, part == p)
end
```





# Requêtes comme condition

---

Les requêtes peuvent être considérées comme des objectifs ou des sous-objectifs devant être satisfaits par le moteur

=> La façon dont Drools implémente un certain degré de raisonnement en arrière consiste à utiliser des requêtes comme conditions dans une règle

Le nœud Query (Query Element Node) s'enregistre en tant que *ViewChangedEventListener* à la requête correspondante pour réagir aux nouveaux résultats ou aux modifications des résultats générés précédemment.



# Performance

---

Réseau Rete/ PHREAK  
Backward chaining  
**Améliorations PHREAK**  
drools-metric



# Introduction

---

Les améliorations consistent principalement :

- A différer l'évaluation des règles
- Propagation des faits via un ensemble permettant le traitement match et le multi-threading
- Segmentation du réseau et utilisation de bit-mask



# Évaluation différée

---

- Au départ les règles sont unliked

- Lors d'insertions/modification/suppression de faits, elles ne sont propagées qu'au sous-réseau Alpha, aucun nœud Beta n'est évalué

- Une heuristique détermine quelle règle est la plus susceptible d'aboutir et impose ainsi un ordre d'évaluation entre elles

- Lorsque tous les nœuds d'une règle ont des données à évaluer, la règle est considérée comme liée.

- Toutes les règles liées sont ajoutées à une file d'attente triée en fonction de l'importance de chaque règle. Différents groupes d'agenda ont des files d'attente différentes et seules les règles de la file d'attente du groupe d'agenda actif sont évaluées.

- => PHREAK retarde l'évaluation du sous-réseau bêta jusqu'à ce que fireAllRules() soit invoquée





# Propagation d'ensemble

---

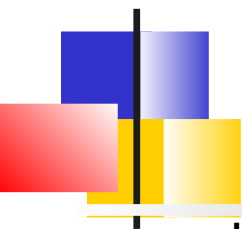
Avec ReteOO, à chaque modification de fait, le réseau était traversé.

Chaque nœud évalué dans le réseau créait un tuple qui était propagé au nœud suivant.

Avec PHREAK, les modifications mise en attente pour les nœuds Beta sont traités par lot et leurs résultats ajoutés à un ensemble.

Cet ensemble est ensuite transmis au nœud suivant où toutes les actions mises en file d'attente sont à nouveau évaluées et ajoutées au même ensemble

=> Cette propagation orientée ensemble offre des avantages de performances pour certaines règles



# Segmentation du réseau

---

Les nœuds partagés entre différentes règles forment des segments. Une règle est alors vue par PHREAK comme un chemin de segments plutôt qu'un chemin de nœuds.

Chaque nœud à l'intérieur d'un segment est affecté à un masque binaire. Chaque segment d'un chemin également.

Lorsqu'un nœud contient suffisamment de données dans son entrée pour être évalué, son bit est défini sur on . Lorsque tous les nœuds d'un segment sont activés, le segment lui-même est activé.

Une règle est alors considérée comme liée lorsque tous ses segments sont activés.

Ces masques binaire sont utilisés par Drools pour éviter la réévaluation des nœuds et segments déjà évalués, fournissant une évaluation plus efficace du réseau PHREAK



# Performance

---

Réseau Rete/ PHREAK  
Backward chaining  
Améliorations PHREAK  
***drools-metric***



# Performance

---

Quelques recommandations pour la performance :

- S'assurer que le nom de la propriété de fait se trouve à gauche de l'opérateur et que la valeur (constante ou variable) se trouve à droite.  
Person( firstName == "John" )  
plutôt que  
Person( "John" == firstName )
- Favoriser l'opérateur ==
- Commencer par les conditions de règle les plus restrictives
- Évitez d'itérer sur de grandes collections d'objets avec des clauses from  
when  
    \$c: Company();  
    Employee (salary > 100000.00, company == \$c)  
plutôt que  
when  
    \$c: Company()  
    \$e: Employee ( salary > 100000.00) from \$c.employees
- Utiliser des listener plutôt que System.out.println dans les règles pour le debug



# drools-metric

---

Le module drools-metric peut être utilisé pour identifier les règles lentes au niveau performance

org.drools : drools-metric

- Trace logging via logback :  
`<logger name="org.drools.metric.util.MetricLogUtils" level="trace"/>`
- Peut être exposé dans Micrometer  
`io.micrometer : micrometer-registry-jmx`
- 2 configurations :  
`drools.metric.logger.enabled = true`  
`drools.metric.logger.threshold` (seuil minimal pour apparaître dans les logs)



# Exemple output

```
TRACE [JoinNode(6) - [ClassObjectType class=com.sample.Order]], evalCount:1000,
elapsedMicro:5962
TRACE [JoinNode(7) - [ClassObjectType class=com.sample.Order]], evalCount:100000,
elapsedMicro:95553
TRACE [ AccumulateNode(8) ], evalCount:4999500, elapsedMicro:2172836
TRACE [EvalConditionNode(9)]:
cond=com.sample.Rule_Collect_expensive_orders_combination930932360Eval1Invoker@ee
2a6922], evalCount:49500, elapsedMicro:18787
```

***evalCount*** est le nombre d'évaluations de contraintes par rapport aux faits insérés lors de l'exécution du nœud

***elapsedMicro*** est le temps écoulé de l'exécution du nœud en microsecondes

Il est possible retrouver la règle associée au nœud via  
*ReteDumper.dumpAssociatedRulesRete(kbase) =>*  
[ AccumulateNode(8) ] : [Collect expensive orders combination]



# Projets connexes

---

**Intégration avec jBPM**  
Drools Fusion et CEP



# Introduction

---

Drools et jBPM se complètent, permettant aux utilisateurs finaux de décrire les connaissances métier à l'aide de différents paradigmes: les règles et les processus

Les projets partagent :

- La même API
- Les mêmes patterns d'intégration avec les applications clientes
- Les mêmes mécanismes de construction et de déploiement





# Accéder aux processus à partir des règles

---

Côté action, la RHS de la règle a accès à *kcontext* et *kcontext.getKieRuntime()*

Avec cette référence d'objet, il est possible de:

- créer, annuler et signaler des processus
- Accédez au *WorkItemManager* pour terminer des *WorkItems*



# Example

---

```
rule "Validate OrderLine Item's cost"
```

```
  when
```

```
    $ol: OrderLine()
```

```
  then
```

```
    Map<String, Object> params = new HashMap<String, Object>();
```

```
    params.put("requested_amount", $ol.getItem().getCost());
```

```
    kcontext.getKieRuntime().startProcess("simple", params);
```

```
end
```



# Les instances de processus comme faits

---

L'insertion d'instances de processus en tant que faits dans le moteur de règles permet d'écrire des règles sur nos processus ou groupes de processus

Le listener ***RuleAwareProcessEventLister***, fourni par jBPM, insère automatiquement les *ProcessInstances* et les met à jour chaque fois qu'une variable est modifiée

Les processus doivent inclure des activités asynchrones



# Example

---

```
rule "Too many orders for just our Managers"
  when
    List($managersCount:size > 0) from collect(Manager())
    List(size > ($managersCount * 3)) from
      collect(WorkflowProcessInstance(processId == "process-
order"))
  then
    //There are more than 3 Process Order Flows per manager.
    // Please hire more people :)
  end
```



# BPMN2 Business Rule Tasks

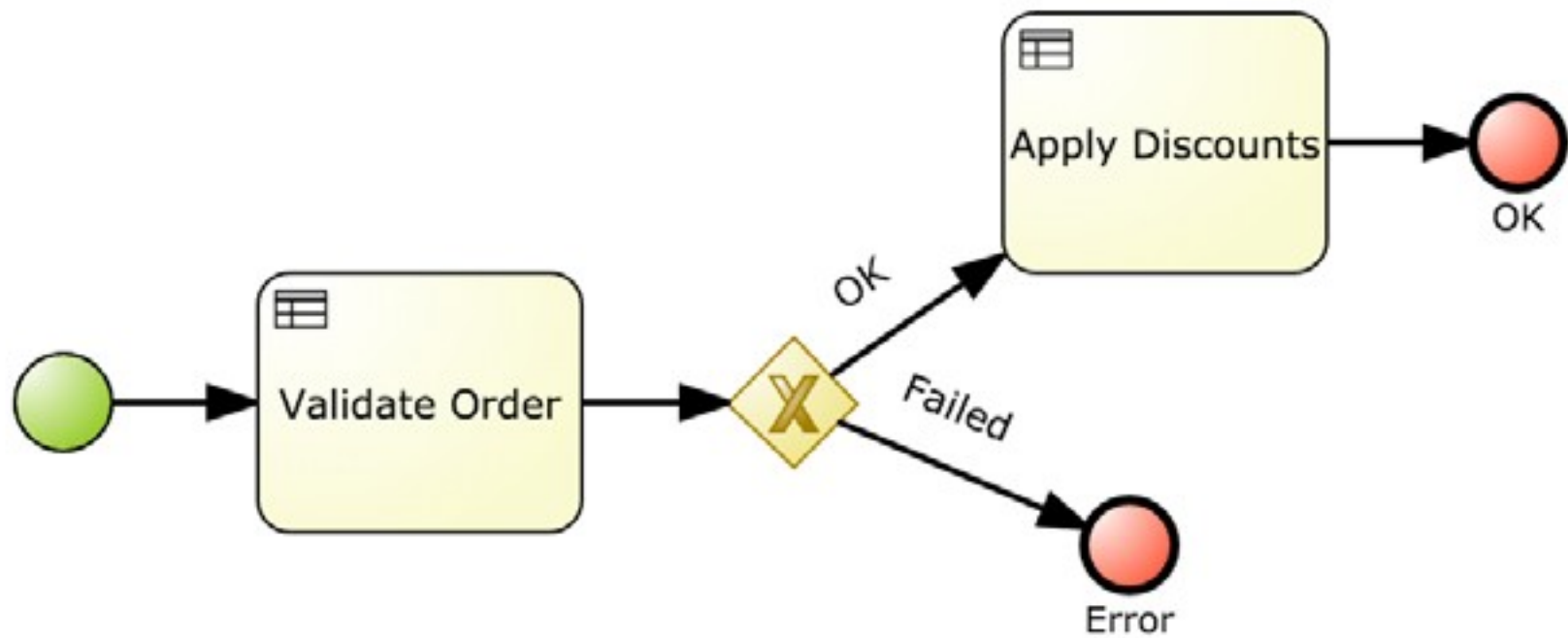
---

La spécification BPMN2 propose un type de nœud spécifique appelé ***Business Rule Task***

Il est utilisé conjointement avec l'attribut de règle ***ruleflow-group***.

- Cette propriété permet de spécifier quelles règles peuvent être déclenchées lorsque le processus atteint le nœud du même nom

# Example





# Projets connexes

---

Intégration avec jBPM  
**Drools Fusion et CEP**



# Introduction

---

Le ***Complex event processing (CEP)*** permet de prendre des décisions basées sur des relations temporelles entre les faits.

L'objectif principal du CEP est de corrélérer de petites unités de données temporelles dans un nuage de données en constante évolution afin de réagir à des situations spéciales difficiles à trouver.

Le raisonnement se fait sur des événements qui sont des faits avec une heure d'occurrence





# Caractéristiques

---

- En général, il faut traiter de nombreux événements mais seulement un petit pourcentage a un réel intérêt.
- Les événements sont généralement immuables, (on ne peut pas changer le passé ! ).
- Les règles et les requêtes travaillent sur des pattern d'événements
- Il y a de fortes relations temporelles entre les événements
- Les événements pris isolément n'ont en général peu d'importance. Le système doit détecter des patterns d'événements reliés temporellement
- Le système doit composer et agréger les événements

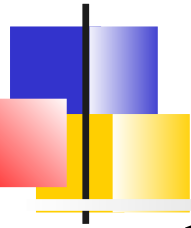


# Événement complexe

---

Un **complex event** est simplement une agrégation, une composition ou une abstraction d'autres événements

Les règles seront exprimées via ces événements complexes



# Sémantique des événements

---

2 types d'événements sont pris en compte :

- **Ponctuel**
- **Intervalle**

Tous les événements sont :

- Immuables
- Un cycle de vie géré: lorsqu'ils sont trop vieux, ils sont retirés de la session



# Déclaration des événements temporels

---

Afin de créer les règles CEP, il faut spécifier les types d'objets devant être traités comme des événements

Cela se fait en ajoutant des méta-données:

- **@Role** : *Fact* ou *Event*
- **@Timestamp** : L'attribut donnant l'estampillage si absent, le moment de l'insertion
- **@Duration** : L'attribut donnant la durée de l'événement (Optionnel)
- **@Expires** : L'attribut indiquant la date d'expiration de l'événement



# Example

---

```
@org.kie.api.definition.type.Role(Role.Type.EVENT)
@org.kie.api.definition.type.Duration("durationAttr")
@org.kie.api.definition.type.Timestamp("executionTime")
@org.kie.api.definition.type.Expires("2h30m")
public class TransactionEvent implements Serializable {
    private Date executionTime;
    private Long durationAttr;
    /* class content skipped */
}
```



## Example (2)

---

```
declare PhoneCallEvent
```

```
  @role(event)
```

```
  @timestamp(whenDidWeReceiveTheCall)
```

```
  @duration(howLongWasTheCall)
```

```
  @expires(2h30m)
```

```
  whenDidWeReceiveTheCall: Date
```

```
  howLongWasTheCall: Long
```

```
  callInfo: String
```

```
end
```



# Opérateurs ternaires

---

Il y a 13 opérateurs temporels disponibles qui permettent de corréler les événements

Par exemple :

```
declare MyEvent
@role(event)
@timestamp(executionTime)
End
```

```
rule "my first time operators example"
when
    $e1: MyEvent()
    $e2: MyEvent(this after[5m] $e1)
Then
    System.out.println("We have two events" + " 5 minutes apart");
end
```

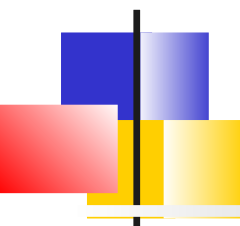


# Exemple : délai d'activation

---

```
rule "Sound the alarm"  
when  
  $f : FireDetected( )  
  not( SprinklerActivated( this after[0s,10s] $f ) )  
then  
  // sound the alarm  
end
```





Operator		Point - Point	Point - Interval	Interval - Interval
A before B	A	●	●—●	●—●
	B			●—●
A after B	A			●—●
	B	●	●—●	●—●
A coincides B	A	●		●—●
	B	●		●—●
A overlaps B	A			●—●
	B			●—●
A finishes B	A		●—●	●—●
	B		●	●—●
A includes B	A		●—●	●—●
	B		●	●—●
A starts B	A		●—●	●—●
	B		●	●—●
A finishedby B	A		●	●—●
	B		●—●	●—●
A startedby B	A		●	●—●
	B		●—●	●—●
A during B	A		●	●—●
	B		●—●	●—●
A meets B	A		●—●	●—●
	B		●	●—●
A metby B	A		●	●—●
	B		●—●	●—●
A overlappedby B	A			●—●
	B			●—●



# Exemple d'agrégation

---

```
rule "Sound the alarm in case temperature rises above  
    threshold"  
when  
    TemperatureThreshold( $max : max )  
    Number( doubleValue > $max ) from accumulate(  
    SensorReading( $temp : temperature ) over window:time( 10m ),  
    average( $temp ) )  
then  
    // sound the alarm  
end
```



# Autre exemple

---

```
rule "More than 10 transactions in an hour from one client"
when
    $t1: TransactionEvent($cId: customerId)
    Number(intValue >= 10) from accumulate(
        $t2: TransactionEvent(this != $t1,
            customerId == $cId, this meets[1h] $t1),
        count($t2) )
    not (SuspiciousCustomerEvent(customerId == $cId,
        reason == "Many transactions"))
then
    insert(new SuspiciousCustomerEvent($cId,
        "Many transactions"));
end
```



# Cycle de vie des événements

---

L'exécution en mode STREAM permet au moteur de détecter les événements qui ne peuvent plus matcher aucune règle et donc de les supprimer de la mémoire de travail

L'identification des événements à supprimer se fait :

- En se basant sur leur attribut d'expiration

- En analysant les contraintes temporelles des règles



# Fenêtres glissantes

---

Les **fenêtres glissantes** permettent de grouper des événements dans une fenêtre temporelle glissante.

Les fenêtres peuvent être exprimées via :

Un **temps** : Prise en compte des événements survenus lors des x dernières unités de temps

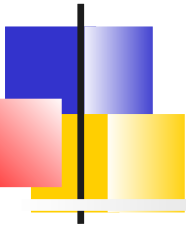
`StockTick() over window:time( 2m )`

Une **longueur** : Prise en compte des x derniers événements

`StockTick( company == "IBM" ) over window:length( 10 )`

Les fenêtrage peuvent être définies :

- A l'intérieur d'une règle
- Ou à l'extérieur pour la réutilisation



# Exemple basé sur la longueur

---

```
rule "last 6 transactions are more than 100 dollars"
When
    Number(doubleValue > 100.00) from accumulate(
        TransactionEvent($amount: totalAmount)
        over window:length(6),
        sum($amount))
then
    //... TBD
end
```



# Exemple basé sur le temps

---

```
rule "obtain last five hours of operations"
when
    $n: Number() from accumulate(
        TransactionEvent($a: totalAmount)
        over window:time(5h),
        sum($a)
    )
Then
    System.out.println("total = " + $n);
end
```



# Déclaration de fenêtre glissante

---

```
declare window Beats
```

```
  @doc("last 10 seconds heart beats")
```

```
  HeartBeat() over window:time( 10s )
```

```
    from entry-point "heart beat monitor"
```

```
end
```

```
rule "beats in the window"
```

```
  when
```

```
    accumulate(HeartBeat() from window Beats,
```

```
      $cnt : count(1))
```

```
  then
```

```
    // there has been $cnt beats over the last 10s
```

```
end
```





# Mise en place

---

Les Bases de connaissance et les Sessions nécessite une configuration spécifique :

- KieBase doivent utiliser le mode ***stream***
- Il faut choisir sur le mode de déclenchement des règles : *discret* ou *continu*
- L'horloge interne de la *KieSession* doit être positionnée



# *KieBase*

---

Kie Base doit être configurée pour utiliser le **STREAM** event processing mode.

Cette configuration informe le runtime qu'il doit gérer des événements et de les garder en interne ordonné via leur timestamp

Avec *kmodule.xml*

```
<kbase name="cepKbase" eventProcessingMode="stream"
packages="chapter06.cep">
<ksession name="cepKsession"/>
</kbase>
```



# Discrete vs Continuous

---

Le déclenchement des règles peut se faire :

- A des moments spécifiques.  
Après l'insertion d'un fait, on peut appeler *fireAllRules*
- Continuellement avec *fireUntilHalt*

Si l'absence d'événements doit déclencher une règle, il faut utiliser la façon continue

Si la seule chose pouvant déclencher une règle est l'insertion d'un fait, la méthode discrète est suffisante



# Horloge de la session

---

Par défaut, KieSessions utilise l'horloge de la machine sur laquelle elle s'exécute

Pour les tests, on peut utiliser une *pseudo-clock*

Pour des scénarios distribués, on peut utiliser une horloge synchronisée



# Exemple Pseudo-clock

---

## Configuration :

```
<kbase name="cepKbase" eventProcessingMode="stream"  
packages="chapter06.cep">  
  <ksession name="cepKsession" clockType="pseudo"/>  
</kbase>
```

## Usage dans un cas de test :

```
SessionPseudoClock clock = ksession.  
getSessionClock();  
clock.advanceTime(2, TimeUnit.HOURS);  
clock.advanceTime(5, TimeUnit.MINUTES);
```



# Intégrer Kie avec les applications

---

## **Considérations**

Patterns d'intégration  
Dépôts de connaissances Maven  
KAAS  
KieWorkbench



# Usage de Drools

---

Drools peut alors interagir avec toutes les couches de de l'application

- Avec l'UI pour fournir des validations complexes de formulaire
- Avec les sources de données pour charger une donnée persistente lorsque l'évaluation d'une règle le nécessite
- Avec les services externe, soit pour charger des informations complexes dans les règles soit pour envoyer des messages aux services externes en fonction de l'issue de la règle



# Communication entre le moteur et l'application cliente

---

Différents mécanismes peuvent être utilisés :

- Les variables globales
- Les Entry points : Partitionnement de faits ou de flux d'événements
- Les Channels pour envoyer des informations au monde extérieur
- Les Listeners ou marshallers (sérialisation de session)





# Mode synchrone

---

Si nous devons exécuter nos règles métier de manière synchrone

- Si aucun état doit être conservé entre 2 exécutions, utiliser des KieSessions stateless et créer autant de sessions que de requêtes
- Utiliser des variables globales pour stocker des informations d'exécution de règles



# Mode asynchrone

---

Le mode asynchrone est nécessaire si l'absence d'entrée provoque le déclenchement de règle.

Il peut être également utilisé si les composants interagissent avec le moteur de manière asynchrone

Dans ce cas :

- Les sessions Kie doivent pouvoir être partagées entre différents threads. Certaines d'entre elles peuvent insérer de nouvelles informations et d'autres peuvent prendre soin de déclencher des règles (fireUntilHalt).
- Enregistrez des listener spécifiques qui veillent à informer les autres composants des situations spéciales détectées par les règles
- Les entry points deviennent un composant très utile lorsque plusieurs sources insèrent des informations dans une seule session Kie



# Intégrer Kie avec les applications

---

Considérations

**Patterns d'intégration**

Dépôts de connaissances Maven

KAAS

KieWorkbench



# Scenarios d'intégration

---

Différentes mécanisme d'intégration peuvent être envisagés :

- Embarquer les règles et le runtime Drools dans l'application cliente
- Externaliser les règles dans un dépôt Maven séparé
- Accéder au règle via un service web :  
Knowledge As A Service

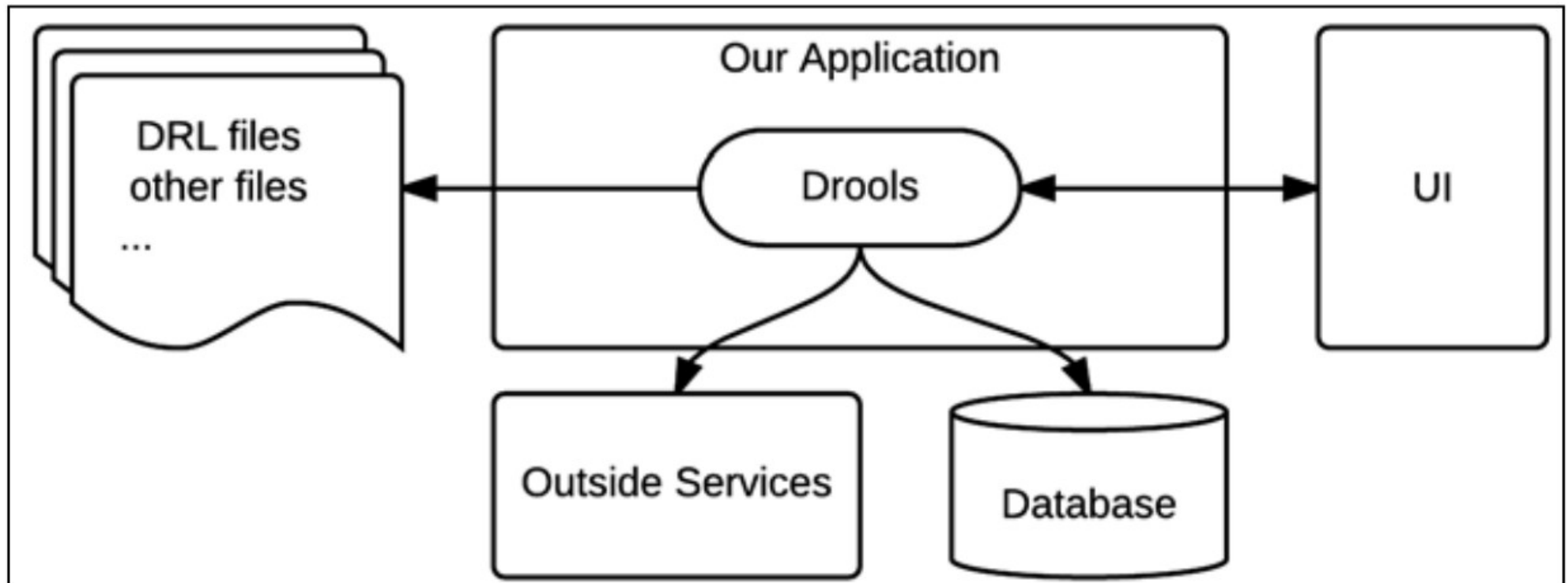
Pour tous ces scénarios, Drools propose du support pour CDI, Spring et Camel



# Architecture :

## *Drools et Ressources embarquées*

---





# Repository Maven distinct

---

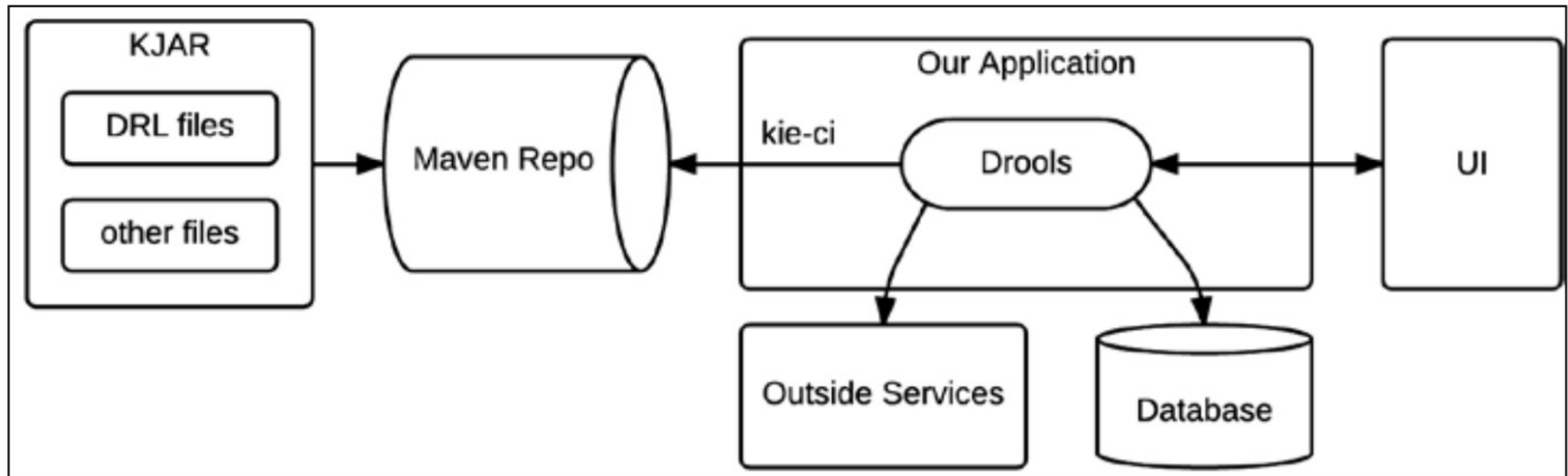
Les règles peuvent être séparées du reste de l'application en utilisant un dépôt Maven :

- Les règles métier sont déployées dans le dépôt sous forme de KJAR
- Les règles peuvent être chargées dynamiquement dans l'application (qui contient le runtime Drools)

=> Les règles peuvent être développées, et déployées autant de fois que nécessaire sans avoir à redéployer les applications.

# Architecture

## *Dépôt Maven*





# Chargement des règles

---

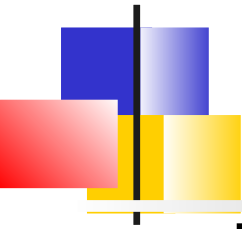
## Avec CDI

```
@Inject @KSession
@KReleaseId(groupId = "org.drools.devguide", artifactId =
    "chapter-11-kjar", version = "0.1-SNAPSHOT")
KieSession kSession;
```

## Avec un scanner

```
KieServices ks = KieServices.Factory.get();
KieContainer kContainer = ks.newKieContainer(
    ks.newReleaseId("org.drools.devguide",
        "chapter-11-kjar", "0.1-SNAPSHOT"));
KieScanner kScanner = ks.newKieScanner(kContainer);
kScanner.start(10_000);
KieSession kSession = kContainer.newKieSession();
```





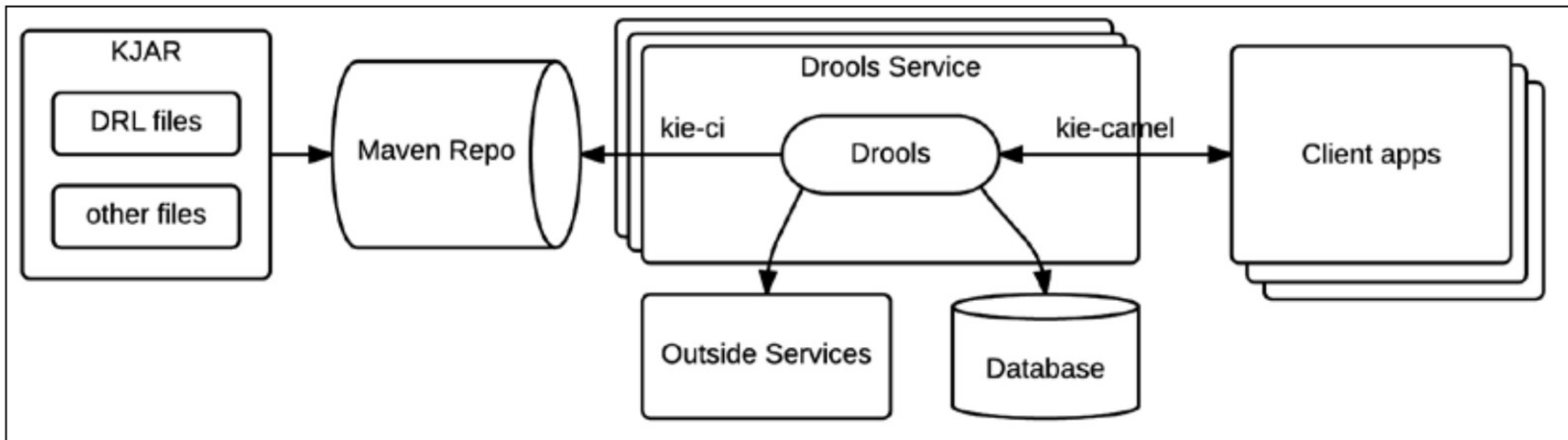
# Knowledge As A Service

---

La dernière approche consiste à externaliser le runtime Drools dans un service externe.

- Ce service expose une API (REST ou autre)
- Plusieurs applications peuvent accéder au service
- Le service peut être répliqué si nécessaire
- Le cycle de vie des règles est complètement indépendant des applications clientes

# Architecture KAAS





# Intégrer Kie avec les applications

---

Considérations  
Patterns d'intégration  
**Dépôts de connaissances Maven**  
KAAS  
KieWorkbench



# Dépôt de connaissance Maven

---

**Construire un kjar**  
Mise à jour des règles métier



# Introduction

---

Un des premiers objectifs lors de l'utilisation d'un moteur de règles est de décorréler les règles métier de l'application qui l'utilise.

Drools s'appuie sur Maven pour déployer les règles métier dans un dépôt.

Les applications peuvent alors :

- Soit déclarer une dépendance Maven vers l'artefact des règles métier enfin de charger les règles via le *classpath*
- Soit accéder à un dépôt distant dynamiquement (http, Scanner)



# Construction d'un *kjar* avec Maven

---

Un projet « règles métier » est typiquement un projet Maven avec en plus un fichier *kmodule.xml* configurant les différentes bases de connaissances

Le cycle de build est enrichi avec le plugin Kie qui précompile les règles et s'assurent de leur validité

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



# Configuration par défaut

---

*kmodule.xml* définit les bases de connaissances et les sessions pouvant être utilisées dans le projet

Un fichier vide donne une configuration par défaut :

- Une unique base de connaissance comprenant toutes les règles/processus/etc. trouvés dans le répertoire **resources** du jar
- Une session *stateful* et une session *stateless*



# Exemple *kmodule.xml*

```
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="KBase1" default="true" eventProcessingMode="cloud" equalsBehavior="equality">
    <ksession name="KSession2_1" type="stateful" default="true"/>
    <ksession name="KSession2_1" type="stateless" default="false" 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="stateful" default="false" clockType="realtime">
      <fileLogger file="drools.log" threaded="true" interval="10"/>
      <workItemHandlers>
        <workItemHandler name="name" type="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>
```





# Attributs de *kbase*

nom	défaut	autorisé	description
<b><i>name</i></b>	aucune		Obligatoire. Utilisé pour récupérer la base à partir du conteneur
<b><i>includes</i></b>	aucune	Liste d'item séparés par des virgules	Les bases de connaissance à inclure
<b><i>packages</i></b>	all	Liste d'item séparés par des virgules	Liste les ressources à compiler
<b><i>default</i></b>	false	true/false	Il n'est pas nécessaire de préciser le nom pour la base de connaissance par défaut
<b><i>equalsBehaviour</i></b>	identity	identity/equality	Test de présence d'un fait dans la base de connaissance
<b><i>eventProcessingMode</i></b>	cloud	cloud/stream	Stream pour drools-fusion
<b><i>declarativeAgenda</i></b>	disabled	disabled/enabled	Agenda déclaratif activé ou non expérimental, les règles peuvent agir directement sur les activations présentes dans l'agenda



# Attributs de *ksession*

nom	défaut	autorisé	description
<b><i>name</i></b>	aucun	N'importe	Obligatoire. Nom utilisé pour récupérer une session du conteneur
<b><i>type</i></b>	stateful	stateless/ stateful	
<b><i>default</i></b>	false	true/false	Il n'est pas nécessaire d'indiquer le nom pour la session par défaut
<b><i>clockType</i></b>	realtime	realtime/ pseudo	Indique si les timestamp des événements est fourni par le système ou par l'application
<b><i>beliefSystem</i></b>	simple	simple/jtms/ defeasible	Système de gestion des faits



# Éléments de ksession

---

Un élément *<ksession>* peut également définir différents sous-éléments :

- Un file ***logger*** permet de définir un fichier de trace qui consignera tous les évènements Drools dans un fichier
- Des ***WorkItemHandlers*** permet de définir des gestionnaires de tâche qui sont associés à des nœuds spécifiques de jBPM (ex : Tâche humaine)
- Des ***listeners*** d'événements Drools :  
*ruleRuntimeEventListener*, *agendaEventListener*  
ou *processEventListener*



# Maven et Release ID (Drools6)

---

Si le projet KIE est un projet Maven, il est possible d'utiliser les coordonnées Maven pour instancier le conteneur.

```
KieServices kieServices = KieServices.Factory.get();  
ReleaseId releaseId = kieServices.newReleaseId( "org.acme", "project", "1.0" );  
KieContainer kieContainer = kieServices.newKieContainer( releaseId ) ;
```



# Dépôt de connaissance Maven

---

Construire un kjar  
**Mise à jour des règles métier**



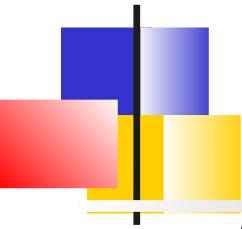
# Déploiement

---

Le déploiement consiste à mettre à jour le dépôt Maven.

L'application cliente peut alors:

- Être mise à jour manuellement pour utiliser une nouvelle version du kjar
- Être dépendante de la dernière version
- Être dépendante d'une famille de version



# *KieScanner*

---

***KieScanner*** permet de scruter en permanence le repository Maven afin de vérifier si une nouvelle release du projet est disponible

Dans ce cas, la nouvelle release est déployée dans le *KieContainer* et les nouvelles règles sont prises en compte

L'utilisation de *KieScanner* nécessite la présence de ***kie-ci.jar*** dans le classpath



# Enregistrement du scanner

---

```
KieServices kieServices = KieServices.Factory.get();

ReleaseId releaseId = kieServices.newReleaseId( "org.acme",
    "myartifact", "1.0-SNAPSHOT" );

KieContainer kContainer =
    kieServices.newKieContainer( releaseId );

KieScanner kScanner =
    kieServices.newKieScanner( kContainer );

// Démarrage de KieScanner
// qui scrute le repository toutes les 10 secondes
kScanner.start( 10000L );
```





# Mise à jour

---

La mise à jour automatique est effective si :

- la version de l'artifact est suffixée avec SNAPSHOT, LATEST RELEASE ou un intervalle de version
- le *KieScanner* trouve une mise à jour dans le dépôt Maven

La mise à jour consiste à télécharger la nouvelle version et d'effectuer un build incrémental. Les KieBases et KieSessions contrôlé par le KieContainer sont automatiquement mises à jour et toutes les nouvelles KieBases ou KieSessions utilisent la nouvelle version du projet.



# Intégrer Kie avec les applications

---

Considérations  
Patterns d'intégration  
Dépôts de connaissances Maven  
**KAAS**  
KieWorkbench



# Introduction

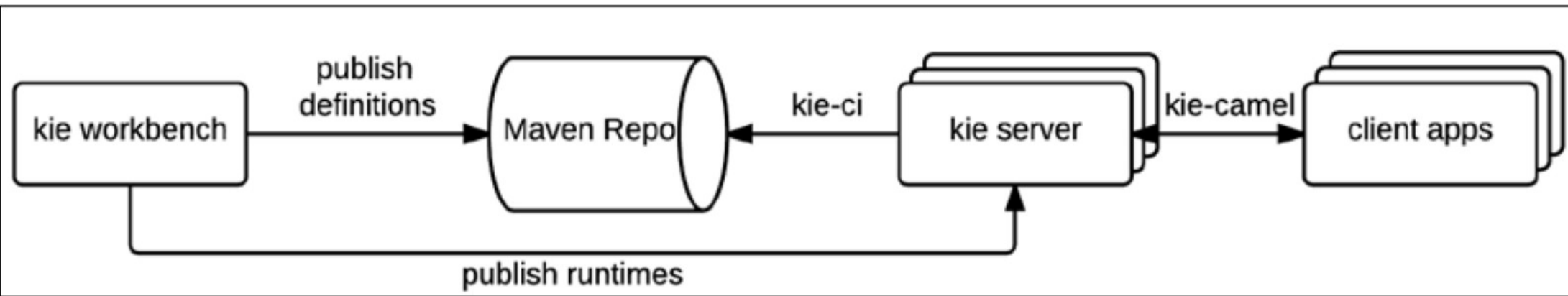
---

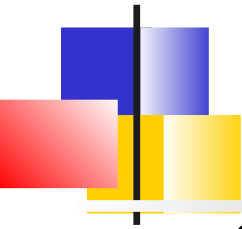
KIE fournit une application web appelée ***KieServer***, qui permet d'héberger des bases de connaissance et qui fournit une API Rest pour interagir avec le moteur

La solution peut être déployée

- En mode standalone, déploiement d'un war sur un serveur
- En mode géré, un contrôleur permet d'instancier et de démarrer des KieServer. Typiquement via un *KieWorkbench*

# Architecture mode géré





# Kie Server

---

Composant serveur autonome et modulaire utilisé pour exécuter des règles et des processus, packagé sous forme de WAR

- Disponible pour les conteneurs Web et les conteneurs d'applications JEE6 et JEE7
- Peut être facilement déployé dans des environnements cloud
- Chaque instance de serveur peut gérer de nombreux KieContainer
- Mécanisme d'extension via *Kie Server Extensions* permettant d'activer ou non *Drools*, *jBPM*, *OptaPlanner*, ...



# Mise en place

---

Distributions : archive zip ou Image docker

Utilise un serveur jBoss wildfly dans la configuration full.

Définir un utilisateur avec le rôle *kie-server*

L'API REST est alors disponible à :

<http://localhost:8080/kie-server/services/rest/server/>

Une documentation swagger est disponible à :

<http://localhost:8080/kie-server/docs/>

Les réponses sont au format XML ou JSON



# Exemple de réponse

---

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<response type="SUCCESS" msg="Kie Server info">
  <kie-server-info>
    <capabilities>KieServer</capabilities>
    <capabilities>BRM</capabilities>
    <capabilities>BPM</capabilities>
    <capabilities>CaseMgmt</capabilities>
    <capabilities>BPM-UI</capabilities>
    <capabilities>BRP</capabilities>
    <capabilities>DMN</capabilities>
    <capabilities>Swagger</capabilities>
    <location>http://localhost:8230/kie-server/services/rest/server</location>
    ...
  </kie-server-info>
</response>
```



# Client REST Natif

---

L'interaction avec le *kie-server* peut être effectuée par n'importe quel client REST (curl, postman, Spring RestTemplate,...)

Mais KIE propose un client natif adapté à ce type d'interaction.

Dépendance Maven :

**org.kie:kie-server-client**





# Configuration d'une requête cliente

---

Les requêtes vers KIE Server doivent spécifier:

- Les crédeniels pour un utilisateur de *kie-server*
- Emplacement du serveur KIE,  
Ex : *http://localhost:8080/kie-server/services/rest/server*
- Format de marshaling pour les demandes et réponses API  
(JSON, JAXB ou XSTREAM)
- Les objets *KieServicesConfiguration* et *KieServicesClient*, qui servent de point d'entrée pour démarrer la communication avec le serveur
- Un objet *KieServicesFactory* définissant le protocole REST et l'accès utilisateur
- Tout autre service client utilisé, tel que *RuleServicesClient*, *ProcessServicesClient* ou *QueryServicesClient*



# Types de client

---

Il existe différents types de clients, chacun étant spécialisé dans un aspect de KIE:

- ***RuleServicesClient*** : Utilisé pour insérer / enlever des faits et pour déclencher les règles
- ***ProcessServicesClient*** : Utilisé pour démarrer, signaler annuler des processus ou des tâches du processus
- ***QueryServicesClient***: Interrogation des processus
- ***UserTaskServicesClient*** : Exécution des tâches d'un processus
- ...



# Exemple (1)

---

```
public static void main(String[] args) {  
    initializeKieServerClient();  
    initializeDroolsServiceClients();  
}
```



# Example (2)

---

```
public static void initializeKieServerClient() {
    conf = KieServicesFactory.newRestConfiguration(URL, USER,
PASSWORD);
    conf.setMarshallingFormat(FORMAT);
    kieServicesClient =
KieServicesFactory.newKieServicesClient(conf);
}

    public static void initializeDroolsServiceClients() {
        ruleClient =
kieServicesClient.getServicesClient(RuleServicesClient.class);
        dmnClient =
kieServicesClient.getServicesClient(DMNServicesClient.class);
    }
```



# Exemple Utilisation de RuleService

---

```
public void executeCommands() {  
  
    String containerId = "hello";  
    System.out.println("== Sending commands to the server ==");  
    RuleServicesClient rulesClient = kieServicesClient.getServicesClient(RuleServicesClient.class);  
    KieCommands commandsFactory = KieServices.Factory.get().getCommands();  
  
    Command<?> insert = commandsFactory.newInsert("Some String OBJ");  
    Command<?> fireAllRules = commandsFactory.newFireAllRules();  
    Command<?> batchCommand = commandsFactory.newBatchExecution(Arrays.asList(insert, fireAllRules));  
  
    ServiceResponse<String> executeResponse = rulesClient.executeCommands(containerId, batchCommand);  
  
    if(executeResponse.getType() == ResponseType.SUCCESS) {  
        System.out.println("Commands executed with success! Response: ");  
        System.out.println(executeResponse.getResult());  
    } else {  
        System.out.println("Error executing rules. Message: ");  
        System.out.println(executeResponse.getMsg());  
    }  
}
```



# Intégrer Kie avec les applications

---

Considérations  
Patterns d'intégration  
Dépôts de connaissances Maven  
KAAS  
**KieWorkbench**



# Workbenches

---

Les projets Kie fournissent des workbenches qui permettent de créer, construire et déployer des processus et/ou règles dans des serveurs Kie

- Ces applications sont construites avec le framework UberFire.
- Ce sont des applications web déployées sur un serveur d'applications (Jboss wildfly)



# Installation

---

Télécharger la distribution ***kie-wb-\*.war***

Deployer le war dans un serveur applicatif

- Drools 7 disponible uniquement pour Wildfly 14

L'application est disponible à:  
`http://<server>:<port>/`





# Données du workbench

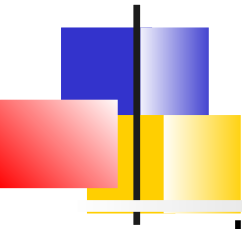
---

Les données internes du workbench sont stockées dans :

***\$WORKING\_DIRECTORY/.niogit*** ,

par exemple *wildfly-8.0.0.Final/bin/.niogit*

C'est le répertoire à sauvegarder en production



# Gestion des utilisateurs

---

Le WB utilise les rôles suivants :

- ***admin*** : Administrateur, gère les utilisateurs, les repositories, ...
- ***developer*** : Quasiment les mêmes droits qu'admin sauf le clonage de repository. Gérer les règles, les modèles, les processus, formulaires, .... Crée, construit et déploie les projets
- ***analyst*** : Idem *developer* avec droit restreint (pas de déploiement par exemple)
- ***user*** : Participe aux processus métier et réalise des tâches
- ***manager*** : Accède au reporting

Dans la configuration simple, les utilisateurs sont créés via  
`$JBOSS_HOME/bin/add-user.sh`



# Démarrage

---

Actions typiques pour démarrer :

1. Créer un utilisateur avec le rôle admin
2. Ajouter un dépôt
3. Ajouter un projet
4. Fournir le modèle métier
5. Créer les règles
6. Construire et déployer

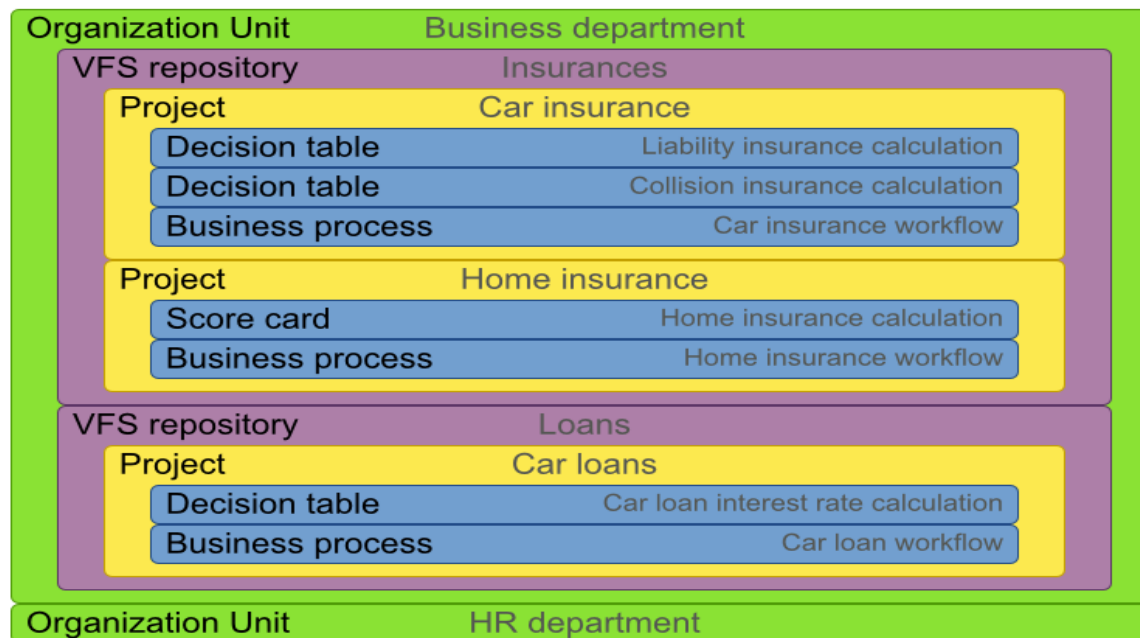
Ces actions peuvent être faites par l'interface web, par un outil de commande en ligne (*kie-config-cli.sh*) ou via une interface REST



# Organisation

Un workbench est structuré avec des unités organisationnelles, des dépôts et des projets

Workbench structure overview





# Explorateur projet

---

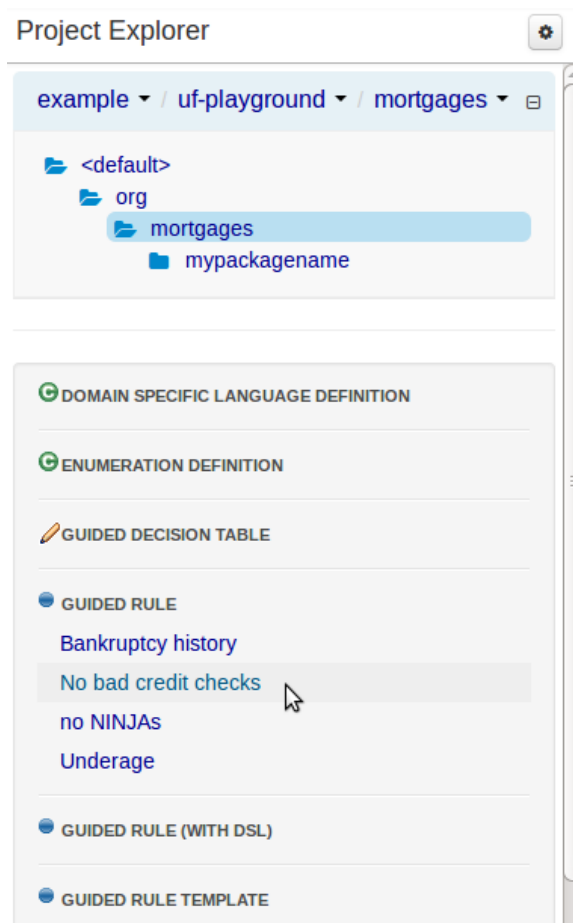
L'explorateur projet permet de parcourir les différentes unités organisationnelles, les dépôts, les projets et les ressources d'un projet

L'utilisateur, après avoir sélectionné un projet, peut accéder à ses packages et voir leur contenu

2 vues sont disponibles :

- Vue projet : certains fichiers sont cachés
- Vue dépôt: tous les fichiers sont affichés

# Example





# Éditeur projet

---

L'éditeur projet centralise toutes les configurations projet disséminés dans différents fichiers

- **Projet : *pom.xml***
  - Général : Coordonnées Maven
  - Dépendances
  - Méta-données
- **Base de connaissance : *kmodule.xml***
  - Bases de connaissance et sessions
  - Méta-données
- **Imports : *project.imports***
  - Suggestions : les imports suggérés dans l'éditeur d'assets
  - Méta-données

Le bouton Build & Depoy construit le projet et déploie le KJAR dans le dépôt Maven interne du workbench



# Modèle de donnée

---

Il est possible de définir son modèle de donnée via Drools-WB

Un modèle de données est toujours associé à un projet

La vue **Data Modeller tool** permet de définir de nouvelles entités

Une entité contient :

- Des attributs modelleur : identifiant, label, description, package
- Des attributs applicatifs : types de base Java ou autre entité



# Data Modeller

Purchases

Create

Identifier

Purchase Order

Purchase Order Header

Purchase Order Line

Purchase Order

Create new field

\*Id

Insert a valid Java identifier

Label

Insert a label

\*Type

Create

Purchase Order (org.jbpm.examples.purchases.PurchaseOrder)

Position	Identifier	Label	Type	
0	description	Description	String	
1	header	Header	Purchase Order Header	
2	lines	Lines	Purchase Order Line [0..N]	

Data object

Field

Identifier

description

Label

Description

Description

Type

String

Equals

Position

0



# Génération de code

---

Le modeleur est un outil visuel qui lorsque le bouton « *Save* » est actionné, génère une classe Java POJO (Plain Old Java Object).

La classe contient un constructeur sans argument, des getters/setters pour ses attributs applicatifs, des constantes annotées pour les attributs *modeleur*, les méthodes *equals()* et *hashCode()*



# Modèle externe

---

Les projets peuvent utiliser des modèles définies dans des classes Java créées à l'extérieur du workbench.

La dépendance vers le jar contenant les classes est indiquée :

- Soit par rapport au repository Maven local (Bouton *Add*)
- Soit par rapport au repository Guvnor M2 du workbench (Bouton *Add from repository*)



# Asset

Drools-WB regroupe les différents ressources qu'il gère sous le terme **d'asset**.

Un *asset* est une ressource stockée et versionnée dans le repository

- Assets métier : Tables décisionnelles, Règles métier en DSL
- Assets techniques : Règles DRL, Énumérations, Processus.
- Fonctions :
- DSL: Domain Specific Languages
- Modèle : Les classes ou les types de faits du modèle métier
- WorkingSets : Sous-ensemble du modèle sur lequel des contraintes peuvent s'appliquer globalement.
- Des cas de test

Les *assets* peuvent être taggés.



# Packages

---

La configuration des packages est généralement effectuée une seule fois par une personne ayant une expertise des règles et des modèles

- Un package est créé en spécifiant son nom
- Tout *asset* appartient à un seul package qui agit comme un espace de noms



# Formats de règles

Le workbench prend en charge plusieurs formats de règles et éditeurs:

- Format BRL avec BRL Editor (également présent dans Eclipse)
- DSL
- Tables de décision dans le fichier à télécharger
- Tables de décision éditées directement dans l'interface Web
- Processus de flux de règles à télécharger
- DRL
- les fonctions
- Configuration des listes de valeurs (Enum)
- Gabarit de règles alimenté par des tableaux de données

Il est toujours possible de consulter la source en drl

***Nouvelle règle → nom, catégorie et format***

=> Un assistant démarre



# Validation

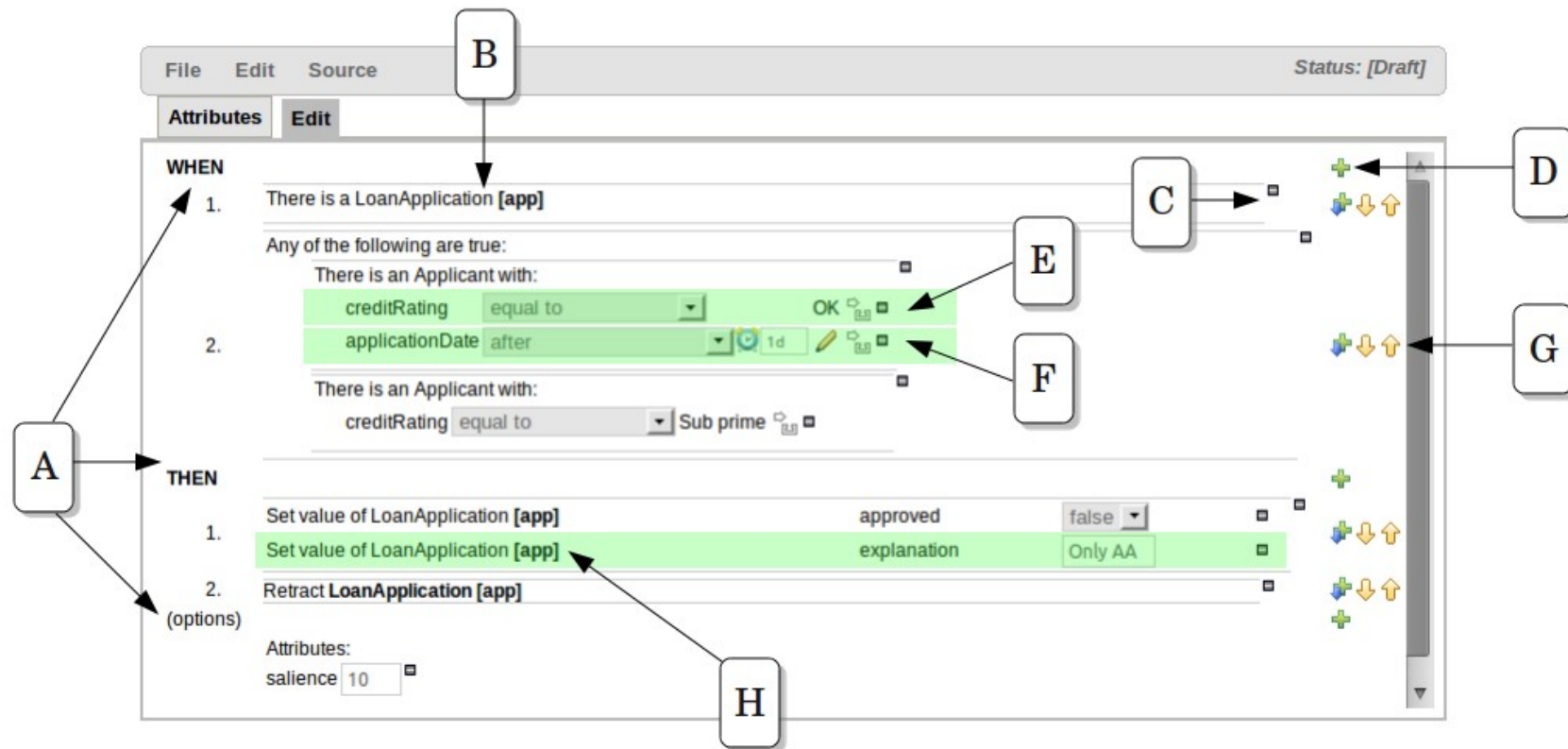
---

Drools-WB permet de valider les règles créées

Le panneau ***Problem*** affiche les validations en temps réel des différents assets

Tous les éditeurs permettent une validation avant l'enregistrement du fichier

# Éditeur guidé de règles métier







# Éditeur guidé de gabarit

Guided Template [t1]

---

**EXTENDS** None selected ▼

---

**WHEN** +

There is an Applicant with:

	age	less than ▼	\$max_age	⊞ ⊞	
1.	age	greater than or equal to ▼	\$min_age	⊞ ⊞	⊞ + ↓ ↑
	creditRating	equal to ▼	\$scr	⊞ ⊞	

---


**THEN** +

(show options...)



# Clés des gabarits

Field value



Field value





Literale value: Literal value

Template key: Template key

Advanced options:

A formula: New formula




















Expression editor: Expression editor

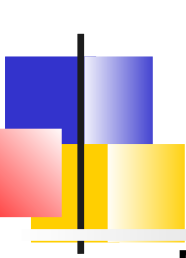


# Données du gabarit

Guided Template [t1]

Add row...

	\$max_age	\$min_age	\$scr
			
 	25	20	AA
 			OK
 			Sub prime
 	35	25	AA
 			OK
 			Sub prime
 	45	35	AA
 			OK
 			Sub prime



# Éditeur de tables de décision

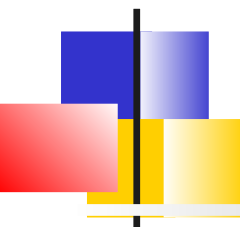
---

Drools-WB propose un éditeur pour les tables de décision.

L'éditeur propose les faits et les champs disponibles dans le contexte du projet

2 types de tables peuvent être créées :

- **Entrées étendues** : les définitions de colonnes ne spécifient pas de valeur. Les valeurs sont alors indiquées dans le corps de la table. Elles peuvent cependant être restreintes par un intervalle.
- **Entrées limitées** : les définitions de colonnes spécifient une valeur. Le corps du tableau contient des cases à cocher




















# Table étendue

+ Decision table					
	#	Description	Age	Make	Premium
			Applicant [\$a]	Vehicle [\$v]	
			age [<]	make [==]	
	1		35	BMW	1000
	2		35	Audi	1000

# Table limitée

+ Decision table

	#	Description	Age < 35	BMW	Audi	Premium 1000
			Applicant [\$a]	Vehicle [\$v]		
			age [<35]	make [==BMW]	make [==Audi]	
 	1		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
 	2		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
 	3		<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
 	4		<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
 	5		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
 	6		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
 	7		<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
 	8		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>



# Scénario de test

---

Les scénarios de Test permettent de valider le fonctionnement des règles et éviter les bugs de régression.

Un scénario de test définit plusieurs sections :

- **Given** liste les faits de test
- **Expected** liste les changements et actions attendus

# Scenario de test

Test Scenario [ Good credit history only ]

SaveDeleteRenameCopyx

Run scenario

+ GIVEN

Insert 'LoanApplication'[app]  
approved: false

Insert 'IncomeSource'[incomeSource]  
Add a field

Insert 'Applicant'[a]  
creditRating: OK

+ CALL METHOD

Add input data and expectations here.

+ EXPECT

Use real date and time

LoanApplication 'app' has values:

approved: equals false

More...

(configuration)  
All rules may fire

+ (globals)

Test Scenario ConfigMetadataAll Test Scenarios

Reporting

Success

Text





# Éditeur DSL

---

File Edit Source

Attributes Edit

```
[when]When the credit rating is {rating:ENUM:Applicant.creditRating} = applicant:Applicant(creditRating=="{rating}")  
[when]When the applicant dates is after {dos:DATE:default} = applicant:Applicant(applicationDate>"{dos}")  
[when]When the applicant approval is {bool:BOOLEAN:checked} = applicant:Applicant(approved=={bool})  
[when]When the ages is less than {num:1?[0-9]?[0-9]} = applicant:Applicant(age<{num})  
[then]Approve the loan = applicant.setApproved(true);  
[then]Set applicant name to {name} = applicant.setName("{name}");
```



# Liste de valeur

---

**Enum Editor [credit ratings]**

SaveDeleteRenameCopyValidate✕▼

Add enum

	Fact	Field	Context
<input type="checkbox"/>	Applicant	creditRating	['AA', 'OK', 'Sub prime']
<input type="checkbox"/>	Person	age	['20','25','30','35']



# Accès à la base du Workbench

---

```
public class MainKieTest {  
    public static void main(String[] args) {  
        // works even without -SNAPSHOT versions  
        String url = "http://localhost:8080/kie-drools/maven2/de/test/Test/1.2.3/Test-1.2.3.jar";  
  
        // make sure you use "LATEST" here!  
        ReleaseIdImpl releaseId = new ReleaseIdImpl("de.test", "Test", "LATEST");  
  
        KieServices ks = KieServices.Factory.get();  
  
        ks.getResources().newUrlResource(url);  
  
        KieContainer kieContainer = ks.newKieContainer(releaseId);  
  
        // check every 5 seconds if there is a new version at the URL  
        KieScanner kieScanner = ks.newKieScanner(kieContainer);  
        kieScanner.start(5000L);  
        // alternatively:  
        // kieScanner.scanNow();  
  
        Scanner scanner = new Scanner(System.in);  
        while (true) {  
            runRule(kieContainer);  
            System.out.println("Press enter in order to run the test again....");  
            scanner.nextLine();  
        }  
  
        private static void runRule(KieContainer kieKontainer) {  
            StatelessKieSession kSession = kieKontainer.newStatelessKieSession("testSession");  
            kSession.setGlobal("out", System.out);  
            kSession.execute("testRuleAgain");  
        }  
    }  
}
```



# Merci!!!

---

❖ MERCI DE VOTRE ATTENTION



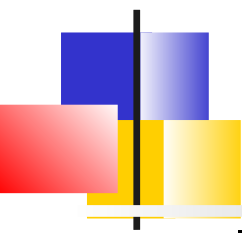
# Annexes

---

Construction programmatique d'un *kModule*  
DSL  
Table de décision  
Gabarit de règles  
Marshaller et persistance de session  
CEP et Event Driven Architecture  
Intégration Spring



# Construction programmatique d'un kModule



# Alternative programmatique

---

Il est possible de définir les bases de connaissance et les sessions programmatiquement (comme avec Drools 5)

Il est alors nécessaire de créer un ***KieFileSystem*** et d'y ajouter les ressources



# API

---

***KieFileSystem*** est un système de fichiers mémoire utilisé pour définir programmatiquement les ressources d'un *KieModule*

***KieBuilder*** est un compilateur pour les ressources contenus dans un *KieModule*

***KieRepository*** : Singleton représentant un dépôt Maven permettant d'accéder aux *KieModules* via leurs coordonnées





# Mise en place

---

- 1) Obtenir un *KieFileSystem* à partir du singleton *KieServices*
- 2) Optionnellement, écrire dans le *KieFileSystem* le *kmodule.xml*
- 3) Optionnellement, écrire dans le *KieFileSystem* le *pom.xml* (définissant les coordonnées Maven du module)
- 4) Ajouter une à une les ressources (.drl, .bpmn, ...)
- 5) Faire appel au builder pour compiler les ressources et vérifier les erreurs
- 6) Obtenir un container à partir du *KieFileSystem*



# Création d'un kModule

---

```
KieServices kieServices = KieServices.Factory.get();
KieModuleModel kieModuleModel =
    kieServices.newKieModuleModel();
KieBaseModel kieBaseModel1 =
    kieModuleModel.newKieBaseModel( "KBase1 " )
    .setDefault( true )
    .setEqualsBehavior( EqualityBehaviorOption.EQUALITY )
    .setEventProcessingMode( EventProcessingOption.STREAM );
KieSessionModel ksessionModel1 =
    kieBaseModel1.newKieSessionModel( "KSession1" )
    .setDefault( true )
    .setType( KieSessionModel.KieSessionType.STATEFUL )
    .setClockType( ClockTypeOption.get("realtime") );
KieFileSystem kfs = kieServices.newKieFileSystem();
kfs.writeKModuleXML(kieModuleModel.toXML());
```



# Ajout des ressources

---

Les ressources peuvent être ajoutées :

- En indiquant leur chemin
- En utilisant la classe *Ressource* de Drools

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



# Construction

---

La construction du *kieModule* consiste à fournir le *KieFileSystem* à un ***KieBuilder***.

Si la compilation s'effectue correctement un *kModule* est stocké dans le *KieRepository* avec ses identifiants Maven :

- Ceux du *pom.xml*
- Ou identifiant par défaut si pas de *pom.xml*



# Exemple

---

```
KieServices kieServices = KieServices.Factory.get();
KieFileSystem kfs = ...
kieServices.newKieBuilder( kfs ).buildAll();
assertEquals( 0,
    kieBuilder.getResults().getMessages( Message.Level.E
RROR ).size() );
KieContainer kieContainer =
    kieServices.newKieContainer(kieServices.getRepository()
        .getDefaultReleaseId()) ;
```



# Internal *KieHelper*

---

```
KieHelper kieHelper = new KieHelper();
kieHelper.addResource(ResourceFactory.newClassPathResource("some/file.drl"), ResourceType.DRL);
//add more resources if needed
Results results = kieHelper.verify();
if (results.hasMessages(Message.Level.WARNING,
    Message.Level.ERROR)){
    //fail
}
KieBase kieBase = kieHelper.build()
```



# Alternatives au DRL

---

DSL  
Tables de décision



DSL





# Introduction

---

Les ***Domain Specific Languages*** (DSL) permettent d'étendre le langage de règles en l'adaptant au langage métier.

C'est une couche d'abstraction dédiée aux experts métier non technique qui est traduite dans le langage de règle au moment de la compilation

Ils peuvent également être utilisés comme gabarits de conditions ou d'action, permettant ainsi de mutualiser certaines parties de règles



# Syntaxe

---

Le format d'un DSL est tout simplement un fichier texte qui traduit des clés en langage « *naturel* » en des expressions *drl*

Chaque ligne commence par indiquer un **scope**, puis la traduction du langage étendu dans le langage de règle

```
[when]This is "{something}"=Something(something=="{something}")
```

```
[then]Log "{message}"=System.out.println("{message}") ;
```

Il est possible d'utiliser également le scope **[keyword]** qui permet de redéfinir un mot-clé :

```
[keyword] quand = when
```

Les phrases définies sont en fait des expressions régulières. Les wildcards peuvent donc être utilisés.



# Examples

---

```
[when]There is a Person with name of  
    "{name}"=Person(name=="{name}")
```

```
[when]Person is at least {age} years old and lives in  
    "{location}"=Person(age > {age}, location=="{location}")
```

```
[then]Log "{message}"=System.out.println("{message}");
```

```
There is a Person with name of "kitty"  
    ---> Person(name="kitty")
```

```
Person is at least 42 years old and lives in "atlanta"  
    ---> Person(age > 42, location="atlanta")
```

```
Log "boo"  
    ---> System.out.println("boo");
```



# Exemple avec regexp

---

```
[when][]is less than or equal to<=  
[when][]is less than=<  
[when][]is greater than or equal to>=  
[when][]is greater than=>  
[when][]is equal to===  
[when][]equals===  
[when][]There is a Cheese with=Cheese()  
[when][]- {field:\w*} {operator} {value:\d*}={field} {operator} {value}
```

-----

```
There is a Cheese with  
- age is less than 42  
- rating is greater than 50  
- type equals 'stilton'
```

-----

```
Cheese(age<42, type=='stilton', rating>50)
```



# Mise en place

---

1. Nommer son fichier DRL avec l'extension *.dslr*

2. Faire référence au fichier DSL dans le fichier de règle avec le mot-clé *expand*  
`expand your-expand.dsl`

3. Passer le DSL au moment de la compilation (Drools 5) :

```
PackageBuilder builder = new PackageBuilder() ;  
builder.addPackageFromDrl( sourceReader,  
dslReader );
```



# Tables de décision



# Présentation

---

Les tables de décisions sont un moyen efficace et compact pour représenter de la logique conditionnelle, elles sont adaptées aux experts métier

Les données saisies dans un tableur permettent de générer les règles.

=> l'expert métier bénéficie alors de son outil préféré : *Excel*

Pour chaque ligne de la table décisionnelle, les données sont combinées avec un gabarit pour générer une règle.

Les tables décisionnelles permettent d'encapsuler les règles et d'isoler le modèle objet. Seuls les paramètres des règles pouvant être modifiés sont exposés.

# Exemple

	B	C	D	E
7				
8				
9		RuleSet	Some business rules	
10		Import	org.drools.decisiontable.Cheese, org.drools.dec	
11		Sequential	true	
12				
13		RuleTable Cheese fans		
14		CONDITION	CONDITION	ACTION
15		Person	Cheese	list
16	(descriptions)	age	type	add(* \$param*)
17	Case	Persons age	Cheese type	Log
18	Old guy	42	stilton	Old man stilton
19	Young guy	21	cheddar	Young man cheddar
20				
21		Variables	java.util.List list	





# Syntaxe des gabarits

---

Les tables décisionnelles comportent 2 types de colonnes :

- Colonnes de **condition**  $\Leftrightarrow$  LHS, la syntaxe de contrainte doit être utilisée
- Colonnes **d'action**  $\Leftrightarrow$  RHS, la syntaxe de code doit être utilisée

*\$param* est utilisé pour indiquer où seront insérées les données des cellules (*\$1* peut être utilisé)

Si la cellule contient une liste de valeurs séparées par des virgules les symboles *\$1*, *\$2*, etc. peuvent être utilisés.

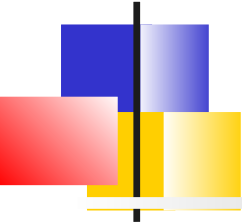
La fonction *forall(DELIMITER){SNIPPET}* peut être utilisée pour effectuer une boucle sur toutes les valeurs disponibles.



# Condition

Le rendu d'une condition dépend de la présence d'une déclaration d'un type d'objet dans une ligne au-dessus. Si le type est précisé, une contrainte de type est créée. Si la cellule contient juste un attribut, la contrainte sera une contrainte d'égalité, sinon la cellule inclus un opérateur .

13	RuleTable Cheese fans	
14	CONDITION	CONDITION
15	Person	
16	age	type
17	Persons age	Cheese type
18	42	stilton



# Conséquences

Le résultat d'une cellule action dépend de la présence d'une entrée sur la ligne immédiatement au dessus.

- Si il n'y a rien, la cellule est interprétée telle quelle
- Si il y a une variable, le contenu de la cellule est ajouté à la variable (Appel de méthode)

ACTION
list.add("\$param");
Log
Old man stilton



# Mots-clés

---

Avant le mot-clé *RuleTable*, les mots-clés suivants peuvent être présent et conditionnent leur cellule immédiatement à droite:

- ***RuleSet*** : Spécification du nom du groupe de règles (package), si vide c'est le groupe par défaut
- ***Sequential*** : La cellule à droite contient *true* ou *false*. Si *true*, la propriété *salience* est utilisé pour garantir l'ordre
- ***Import*** : Liste de classes java à importer
- ***Functions*** : Déclaration de fonctions
- ***Variables*** : Déclaration de variables globales
- ***Queries*** : Déclaration de requêtes



# Example

---

RuleSet	Control Cajas[1]
Import	foo.Bar, bar.Baz
Variables	Parameters parametros, RulesResult resultado, EvalDate fecha
Functions	<pre>function boolean isRango(int iValor, int iRangoInicio, int iRangoFinal) {     if (iRangoInicio &lt;= iValor &amp;&amp; iValor &lt;= iRangoFinal)         return true;     return false; }  function boolean esIgualTipo(TipoVO tipoVO, int p_tipo, boolean isNull) {     if (tipoVO == null)         return isNull;     return tipoVO.getSecuencia().intValue() == p_tipo; }</pre>



# *RuleTable*

---

Une cellule avec ***RuleTable*** indique le début de la définition d'une table de règle.

La table démarre avec la ligne suivante.  
Elle est lue de gauche à droite et de bas en haut jusqu'à une ligne blanche.



# Mots-clés dans la table de règles

---

**CONDITION** : Indique une colonne condition

**ACTION** : Indique une colonne action

**PRIORITY** : Indique une colonne utilisé pour l'attribut *salience*

**DURATION** : Indique l'attribut *duration* de la règle

**NAME** : Le nom de la règle (optionnel)

**NO-LOOP** : Attribut *no-loop* (*true* ou *false*)

**ACTIVATION-GROUP** : Attribut *activation-group*

**AGENDA-GROUP** : Attribut *agenda-group* de la règle

**RULEFLOW-GROUP** : Attribut *ruleflow-group* de la règle



# Intégration

---

L'intégration d'une table décisionnelle nécessite la librairie

***drools-decisiontables.jar***

La classe principale

***SpreadsheetCompiler*** prend en entrée un fichier *csv* ou *excel* et génère les règles en DRL

Les règles peuvent être alors manipulées indépendamment





# Exemple : utilisation SpreadSheet Compiler

---

```
@SuppressWarnings("restriction")  
  
public static void main(String[] args) {  
  
    String fileName="/org/formation/dtables/assurance.xls";  
  
    if ( args.length > 0 )  
        fileName = args[0];  
  
    SpreadsheetCompiler spc = new SpreadsheetCompiler();  
    String drl = spc.compile(fileName, InputType.XLS);  
    System.out.println("DRL\n"+drl);  
}
```



# Scénario de mise en place

---

1. L'expert métier démarre à partir d'un gabarit de table décisionnelle
2. Il renseigne les paramètres des règles et les actions avec des descriptions métier
3. Ils saisit les lignes correspondant aux règles
4. La table décisionnelle est reprise par un technicien qui fait correspondre le langage métier à des scripts
5. L'expert métier et le technicien revoient ensemble les modifications effectuées.
6. L'expert métier peut éditer les règles selon ses besoins.
7. Le technicien peut écrire des cas de tests vérifiant les règles



# Gabarit de règles



# Gabarits de règle

---

Les **gabarits de règle** utilisent des sources de données tabulaires (Tableur, CSV ou autres) pour générer de nombreuses règles.

C'est une technique finalement plus puissante que les tables de décisions :

- Les données peuvent être stockées dans une base de données
- La génération de règle peut être conditionnée par les données
- Les données peuvent être utilisées dans n'importe quelle partie des règles (opérateur, nom d'une classe, nom d'une propriété)
- Plusieurs gabarits peuvent être exécutées sur les mêmes données



# Structure gabarit

---

## Le fichier texte

- commence par une entête **template header** .
- Ensuite, la liste des colonnes des données tabulaires
- Une ligne vide pour marquer la fin des définitions de colonne
- L'entête standard DRL (package, import, global, fonctions)
- Le mot clé **template** marque le début d'un gabarit de règle, il peut en avoir plusieurs dans le fichier.
- Le gabarit utilise la syntaxe **@{token\_name}** pour les substitutions (ex : @{row.rowNumber}
- Le mot clé **end template** marque la fin du gabarit.



# Exemple

---

**template header**

age  
type  
Log

```
package org.drools.examples.templates;
```

```
global java.util.List list;
```

**template** "cheesefans"

```
rule "Cheese fans_{row.rowNumber}"
```

```
when
```

```
Person(age == @{age})
```

```
Cheese(type == "{@type}")
```

```
then
```

```
list.add("{@log}");
```

```
end
```

**end template**



# *kmodule*

---

Le gabarit doit ensuite être inclus avec le fichier de donnée associés dans la définition du *kmodule*

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://drools.org/xsd/kmodule">
  <kbase name="TemplatesKB" packages="org.drools.examples.templates">
    <ruleTemplate
      dtable="org/drools/examples/templates/ExampleCheese.xls"
      template="org/drools/examples/templates/Cheese.drt"
      row="2" col="2"/>
    <ksession name="TemplatesKS"/>
  </kbase>
</kmodule>
```



# Exemple BD

---

```
// Get results from your DB query...
resultSet = preparedStmt.executeQuery();
// Generate the DRL...
resultSetGenerator = new ResultSetGenerator();
String drl = resultSetGenerator.compile(resultSet,
    new
    FileInputStream("path/to/template.drt"));
```





# Syntaxe



# Élément conditionnel *and*

---

Les éléments conditionnels peuvent être combinés avec les opérateurs ***and*** (opérateur implicite)

L'opérateur *and* peut-être utilisé en préfixe ou en infixe

Implicite

```
Cheese( cheeseType : type )
```

```
Person( favouriteCheese == cheeseType)
```

Préfixe

```
(and Cheese( cheeseType : type )
```

```
Person( favouriteCheese == cheeseType ) )
```

Infixe

```
(Cheese( cheeseType : type ) and
```

```
Person( favouriteCheese == cheeseType ) )
```



# Élément conditionnel *or*

---

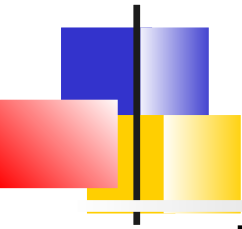
L'élément conditionnel ***or*** a pour effet de créer plusieurs sous-règles distinctes qui deviennent complètement indépendantes. Il peut s'employer en préfixe ou en infixé.

```
(or Person( sex == "f", age > 60 )
```

```
Person( sex == "m", age > 65 )
```



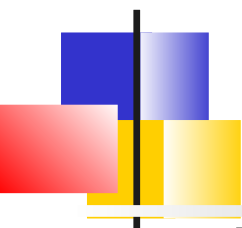
# Marshaller et persistance de session



# *KieMarshallers*

Les ***KieMarshallers*** sont utilisés pour sérialiser/désérialiser les *KieSessions*.

```
ByteArrayOutputStream baos = new  
    ByteArrayOutputStream();  
  
Marshaller marshaller =  
    KieServices.Factory.get().getMarshallers().newMarshal  
        ler( kbase );  
  
marshaller.marshall( baos, ksession );  
  
baos.close();
```



# Persistance et transactions

---

La persistance avec Drools est implémenté via **JPA**.

Il est cependant nécessaire de disposer d'une implémentation de **Java Transaction API** (JTA)

Pour les classes de test et le développement l'implémentation *Bitronix Transaction Manager* est suggérée pour un environnement de production *JBoss Transactions* est recommandé.



# Exemple

---

```
KieServices kieServices = KieServices.Factory.get();
Environment env = kieServices.newEnvironment();
env.set( EnvironmentName.ENTITY_MANAGER_FACTORY,
Persistence.createEntityManagerFactory( "emf-name" ) );
env.set( EnvironmentName.TRANSACTION_MANAGER,
    TransactionManagerServices.getTransactionManager() );
KieSession ksession =
kieServices.getStoreServices().newKieSession( kbase, null, env );
int sessionId = ksession.getId();
UserTransaction ut =
    (UserTransaction) new InitialContext().lookup( "java:comp/UserTransaction" );
ut.begin();
ksession.insert( data1 );
ksession.insert( data2 );
ksession.startProcess( "process1" );
ut.commit();
```



# Chargement d'une session

---

Pour récupérer une session stockée en base :

```
KieSession ksession =  
kieServices.getStoreServices().loadKieSession(  
    sessionId, kbase, null, env );
```





# Chargement d'une session

---

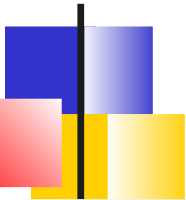
Pour récupérer une session stockée en base :

```
KieSession ksession =  
kieServices.getStoreServices().loadKieSession(  
    sessionId, kbase, null, env );
```



# *persistence.xml*

```
<persistence-unit name="org.drools.persistence.jpa" transaction-type="JTA">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <jta-data-source>jdbc/BitronixJTADatasource</jta-data-source>
  <class>org.drools.persistence.info.SessionInfo</class>
  <class>org.drools.persistence.info.WorkItemInfo</class>
  <properties>
    <property name="hibernate.dialect"
value="org.hibernate.dialect.H2Dialect"/>
    <property name="hibernate.max_fetch_depth" value="3"/>
    <property name="hibernate.hbm2ddl.auto" value="update" />
    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.transaction.manager_lookup_class"
value="org.hibernate.transaction.BTMTransactionManagerLookup" />
  </properties>
</persistence-unit>
```



# JTA Datasource avec Bitronix

---

```
PoolingDataSource ds = new PoolingDataSource();
ds.setUniqueName( "jdbc/BitronixJTADatasource" );
ds.setClassName( "org.h2.jdbcx.JdbcDataSource" );
ds.setMaxPoolSize( 3 );
ds.setAllowLocalTransactions(
ds.getDriverProperties().setProperty("driverClassName",
    "org.h2.Driver");
ds.getDriverProperties().setProperty("url",
    "jdbc:h2:tcp://localhost/JPADroolsFlow");
ds.getDriverProperties().setProperty("user", "sa");
ds.getDriverProperties().setProperty("password", "");
ds1.init();
```

## **jndi.properties**

```
java.naming.factory.initial=bitronix.tm.jndi.BitronixInitialContextFactory
```



# CEP et Event Driven Architecture



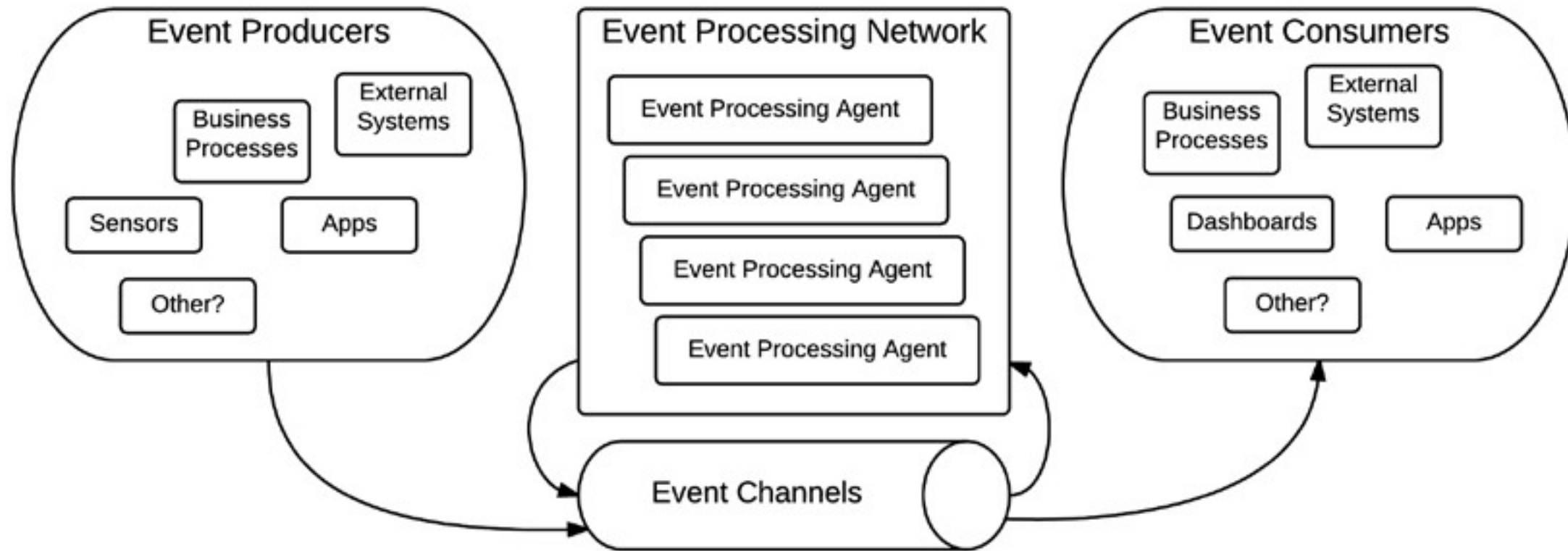
# Event-driven architecture

---

The idea of event-driven architecture (EDA) is to classify the components in the following four different categories:

- Event Producer : Creators of events, for example a sensor
- Event Consumer : Final output architecture which point the produced value. For example a dashboard
- Event Channels : Communication protocols between all the other components. For example JMS
- Event Processing Agents: Group the events to detect and process complex events. The Drools rules

# Event-driven architecture





# Exemple

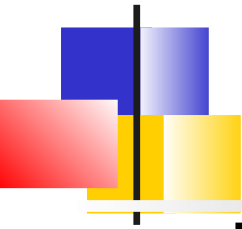
---

```
// Insert event from one entry-point to another
rule "Routing transactions from small resellers"
  when
    $t: TransactionEvent() from
      entry-point "small resellers"
  then
    entryPoints["Stream Y"].insert(t);
  end
```



# Intégration Spring





# kie-spring

---

Utilisation d'un kmodule / Bean configuration

utilisation de balise de namespace kie

Le préfixe a changé de 'drools:' pour 'kie:'

Dépendance :

```
<groupId>org.kie</groupId>
```

```
<artifactId>kie-spring</artifactId>
```

```
<version>6.2.0.Final</version>
```



# kie-spring

---

`<kie:kmodule>`

id requis l'id du bean  
ne peut contenir que

`<kie:kbase>`

name : requis

packages

includes

default

scope

eventProcessingMode

equalsBehavior

declarativeAgenda



# kie-spring

---

ne peut contenir que kie:ksession

<kie:ksession>

name : requis

type

default

clockType : REALTIME / PSEUDO

listeners-ref

Déclaration nécessaire au fonctionnement de kie-spring Impératif:

<bean id="kiePostProcessor"

class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>

ou pour les annotations

<bean id="kiePostProcessor"

class="org.kie.spring.annotations.KModuleAnnotationPostProcessor"/>



# kie-spring

---

exemple

```
<kie:kmodule id="sample-kmodule">  
  <kie:kbase name="drl_kiesample3" packages="drl_kiesample3">  
    <kie:ksession name="ksession1" type="stateless"/>  
    <kie:ksession name="ksession2"/>  
  </kie:kbase>  
</kie:kmodule>  
  
<bean id="kiePostProcessor"  
  class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```



# kie-spring

---

`<kie:releasesId>`

id : bean id, requis

groupId : requis

artifactId : requis

version : requis

exemple

```
<kie:releasesId id="beanId" groupId="org.kie.spring"
    artifactId="named-artifactId" version="1.0.0-SNAPSHOT"/>
```

`<kie:import />`

releasesId

enableScanner

scannerInterval



# kie-spring

---

Utilisation avec releaseId

```
<kie:import releaseId-ref="namedKieSession"  
    enableScanner="true" scannerInterval="1000"/>
```

```
<kie:releaseId id="namedKieSession" groupId="org.drools"  
    artifactId="named-kiesession" version="6.3.0-SNAPSHOT"/>
```

Le scanner peut être récupéré via

```
KieScanner releaseIdScanner =  
    context.getBean("namedKieSession#scanner", KieScanner.class);  
releaseIdScanner.scanNow();
```



# kie-spring

---

## Listeners

Drools supporte trois types de listeners `AgendaListener`, `WorkingMemoryListener`, `ProcessEventListener`

Ils peuvent être déclaré respectivement via

```
<kie:agendaEventListener/>
```

```
<kie:processEventListener/>
```

```
<kie:ruleRuntimeEventListener/>
```

`DebugAgendaEventListener`

`DebugRuleRuntimeEventListener`

`DebugProcessEventListener`

Qui sortent les traces sur `System.err`



# kie-spring

---

ref : référence optionnelle au bean  
avec référence ils utilisent l'implémentation fournie

```
<bean id="mock-agenda-listener" class="mocks.MockAgendaEventListener"/>
<bean id="mock-rr-listener" class="mocks.MockRuleRuntimeEventListener"/>
<bean id="mock-process-listener" class="mocks.MockProcessEventListener"/>
```

```
<kie:kmodule id="listeners_kmodule">
  <kie:kbase name="drl_kiesample" packages="drl_kiesample">
    <kie:ksession name="ksession2">
      <kie:agendaEventListener ref="mock-agenda-listener"/>
      <kie:processEventListener ref="mock-process-listener"/>
      <kie:ruleRuntimeEventListener ref="mock-rr-listener"/>
    </kie:ksession>
  </kie:kbase>
</kie:kmodule>
```

```
<bean id="kiePostProcessor"
      class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```





# kie-spring

---

On peut sortir les listener de la session et en faire un groupe disponible pour tous

```
<bean id="mock-agenda-listener" class="mocks.MockAgendaEventListener"/>
<bean id="mock-rr-listener" class="mocks.MockRuleRuntimeEventListener"/>
<bean id="mock-process-listener" class="mocks.MockProcessEventListener"/>
```

```
<kie:kmodule id="listeners_module">
  <kie:kbase name="drl_kiesample" packages="drl_kiesample">
    <kie:ksession name="statelessWithGroupedListeners" type="stateless"
      listeners-ref="debugListeners"/>
  </kie:kbase>
</kie:kmodule>
```

```
<kie:eventListeners id="debugListeners">
  <kie:agendaEventListener ref="mock-agenda-listener"/>
  <kie:processEventListener ref="mock-process-listener"/>
  <kie:ruleRuntimeEventListener ref="mock-rr-listener"/>
</kie:eventListeners>
```



# kie-spring

---

## **Logger**

se définit dans la session

### **Console logger**

```
<kie:consoleLogger/>
```

### **File logger**

```
<kie:fileLogger/>
```

ID : indentifiant unique, requis

file : chemin complet ou fichier, requis

threaded

interval

exemple :

```
<kie:fileLogger id="tfl_logger" file="#{ systemProperties['java.io.tmpdir'] }/log2"
  threaded="true" interval="5"/>
```

## **Il faut fermer le logger programmatiquement**

```
LoggerAdaptor adaptor = (LoggerAdaptor) context.getBean("fl_logger");
adaptor.close();
```



# kie-spring

---

## **<kie:batch>**

A placer dans kie:session  
contient

### **insert-object**

ref = String (optional)

### **set-global**

identifier = String (required)

reg = String (optional)

### **fire-all-rules**

max : n

### **fire-until-halt**

### **start-process**

identifier = String (required)

ref = String (optional)

### **signal-event**

ref = String (optional)

event-type = String (required)

process-instance-id =n (optional)



# kie-spring

---

exemple

```
<kie:batch>  
  <kie:insert-object  
ref="person2"/>  
  <kie:set-global  
identifier="persons" ref="personsList"/>  
  <kie:fire-all-rules max="10"/>  
</kie:batch>
```



# kie-spring

---

## **Persistence**

Il faut définir kie:jpa-persistence  
puis un transaction manager  
puis un entity manager factory



# kie-spring

---

## **exemple complet:**

```
<kie:kstore id="kstore" /> <!-- provides KnowledgeStoreService implementation -->
```

```
<bean id="myEmf"
```

```
    class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
```

```
        <property name="dataSource" ref="ds" />
```

```
        <property name="persistenceUnitName"
```

```
            value="org.drools.persistence.jpa.local" />
```

```
</bean>
```

```
<bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
```

```
    <property name="entityManagerFactory" ref="myEmf" />
```

```
</bean>
```



# kie-spring

---

```
<kie:kmodule id="persistence_module">
  <kie:kbase name="drl_kiesample" packages="drl_kiesample">
    <kie:ksession name="jpaSingleSessionCommandService">
      <kie:configuration>
        <kie:jpa-persistence>
          <kie:transaction-manager ref="txManager"/>
          <kie:entity-manager-factory ref="myEmf"/>
        </kie:jpa-persistence>
      </kie:configuration>
    </kie:ksession>
  </kie:kbase>
</kie:kmodule>

<bean id="kiePostProcessor"
      class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```



# kie-spring

---

## Spring Profiles

On peut définir dans le même fichier des configurations suivant les environnements

en encadrant les tags kmodule par

```
<beans profile="environnement">
```

```
<kie:kmodule
```

```
..
```

```
</beans>
```

à spécifier ensuite dans le system.properties ou équivalent

```
spring.profiles.active="production"
```

ou par option

```
-Dspring.profiles.active="development"
```