

# Le moteur de workflow jBPM

---

David THIBAU – 2022

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



# Agenda

---

- **Introduction**

- Modèle de workflow
- Fonctionnalités jBPM
- Projets KIE
- Séquence de démarrage

- **Prise en main**

- Installation IDE
- API Engine et Test
- Introduction à BPMN 2.0

- **Modélisation avec jBPM**

- Types de nœuds
- Données d'instance
- Persistance et transaction

- **Intégration services externes**

- Workflow Humain
- Nœuds Service



# Introduction

---

## **Modèle Workflow**

Fonctionnalités *JBPM's*

Projets KIE

Séquence de démarrage



# Définition

---

- ❖ Un moteur de workflow est un logiciel qui exécute des processus automatisés
- ❖ Le moteur :
  - Traite des événements : Réception de signal, expiration de timeout, Fin d'une tâche humaine...
  - Réagit en suivant une spécification prédéfinie : le processus.
- ❖ Les actions déclenchées par le moteur peuvent être de toutes sortes :
  - Stocker un document dans un dépôt, mettre à jour une liste de tâches, envoyer un email, démarrer un batch, appeler un service ...



# Modélisation Graphique

---

La manière la plus naturelle pour modéliser un processus est un **graphe**.

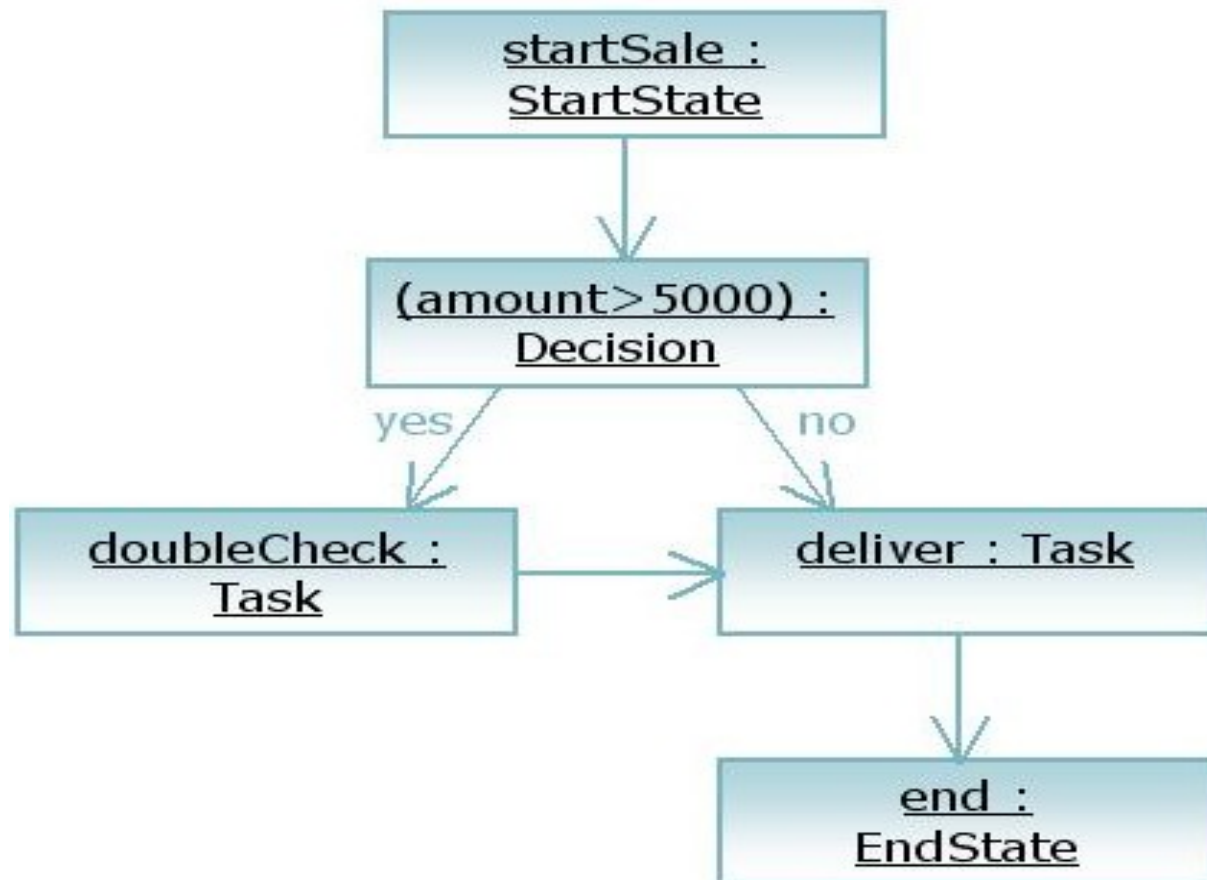
Le graphe est composé de :

- Nœuds représentant des activités
- Transitions qui représentent les séquences entre activités

Le graphe contient généralement un unique nœud de départ et au moins un nœud de fin

Les chemins représentent les différents scénariis d'exécution

# Exemple



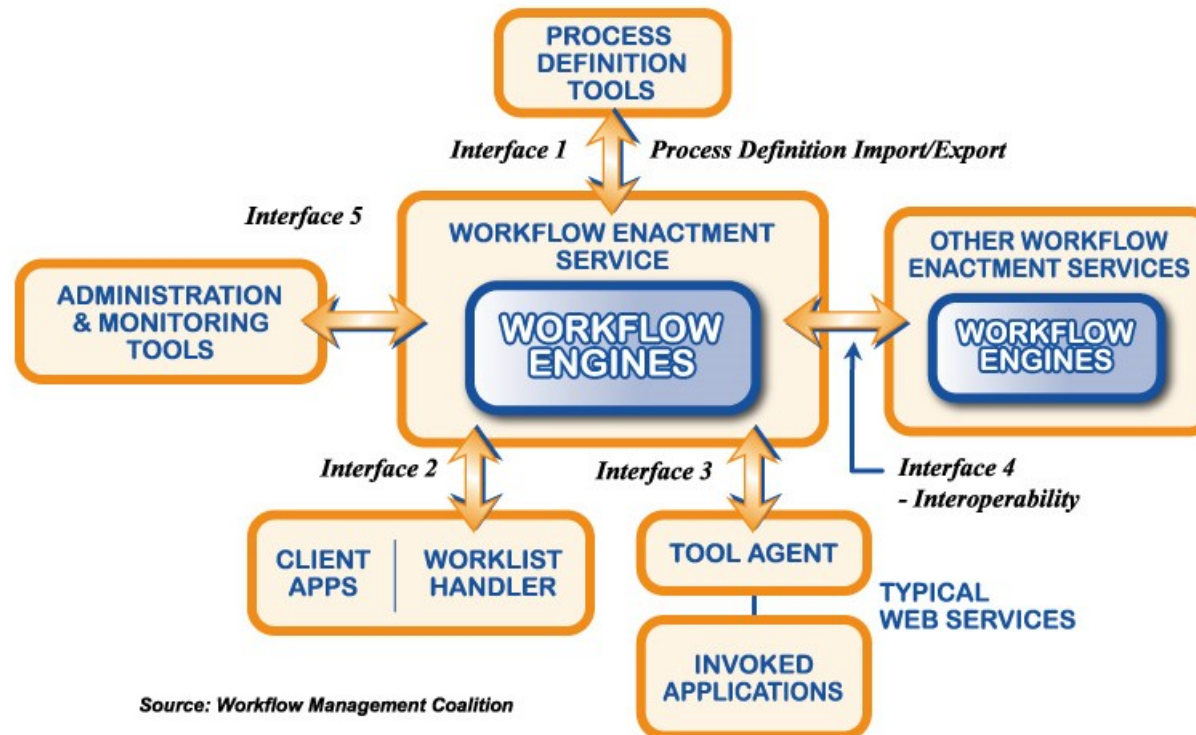


# Modèle de référence

---

- ❖ Le *Workflow Management Coalition* a publié un modèle de référence qui définit 4 composants et leur API
  - Le service de workflow
  - Les outils de modélisation
  - Les applications clientes
  - Les applications invoquées
  - Les outils d'administration et surveillance

# Workflow Reference Model (Wfmc)







# Workflow Service

---

- ❖ C'est la plate-forme fournissant un environnement d'exécution des processus
- ❖ Les processus sont instanciés à partir de de définitions de processus
- ❖ La plateforme interagit avec les autres composants afin de :
  - Exécuter les activités
  - Fournir le statut des instances



# Outils de modélisation et BPMN2.x

---

- ❖ Les outils de modélisation doivent être compris par les experts métier, les ingénieurs ... et le moteur de workflow
- ❖ **BPMN 2.0** est un format XML qui est devenu un standard pour la définition de processus
- ❖ Les éditeurs graphiques permettent de disposer les nœuds, les transitions et d'éditer leurs propriétés.
  - Le graphe résultant est fourni au moteur
- ❖ Quelquefois, il est possible de tester ou simuler le processus via l'outil de modélisation.



# Applications Clientes

---

- ❖ Elles utilisent le service de workflow via son API, afin de :
  - Démarrer, signaler, annuler un processus
  - Questionner le statut d'un process, récupérer ses données, l'activité courante, le responsable d'une tâche, ...
  - Accéder au trace et à l'historique workflow
- ❖ Généralement, les fonctionnalités workflow sont juste une partie de l'application cliente
- ❖ Généralement, la plate-forme de workflow fournit une application cliente générique à destination des administrateurs de la plateforme.



# Applications invoquées

---

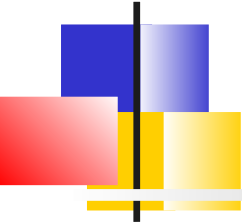
- ❖ Les applications invoquées peuvent être de toute sorte :
  - Script Shell
  - Classe Java implémentant une interface
  - Services Web
  - ...
- ❖ La plateforme doit fournir un support pour leur intégration



# Monitoring et BAM

---

- ❖ La plateforme génère des traces pour toutes les exécutions.
- ❖ Les traces peuvent être utilisées pour du BAM (Business Activity Monitoring) ou du reporting



# Domaines d'application

---

- ❖ Les moteurs de workflow sont généralement utilisés dans 2 domaines distincts :
  - Le **BPM (Business Process management)** : Un processus métier implique généralement des activités humaines et des activités système. Les tâches humaines doivent être complétées pour faire progresser le processus
  - **Service orchestration** (Architecture SOA)  
Le processus définit les relations et les séquences entre des applications exposant une API web



# Introduction

---

Modèle Workflow  
**Fonctionnalités *JBPM***  
Projets KIE  
Séquence de démarrage



# Versions jBPM

---

- \* jBPM 3 / Septembre 2009 : Langage de modélisation propriétaire jPDL, Java 1.4, Service de persistance basé sur Hibernate
- \* JBPM 4 / Juillet 2010 : Langage jPDL, Portage Java 5 et JPA. Version pas complètement aboutie
- \* JBPM 5 / Février 2011 : Standard BPMN 2.0, Fusion des projets jBPM et Drools-Flow
- \* jBPM6 / Janvier 2014 : Offre Redhat commerciale. Fait partie de la suite KIE
- \* jBPM7 / Juillet 2017 : Nouvelle API d'admin , tableaux de bord





# Pure Java

---

jBPM est 100 % Java, il peut être intégré comme :

- Une librairie standard (*.jar*)
- Un service Web offrant :
  - Une API REST pour exécuter les processus
  - Un dépôt distant stockant les définitions de processus



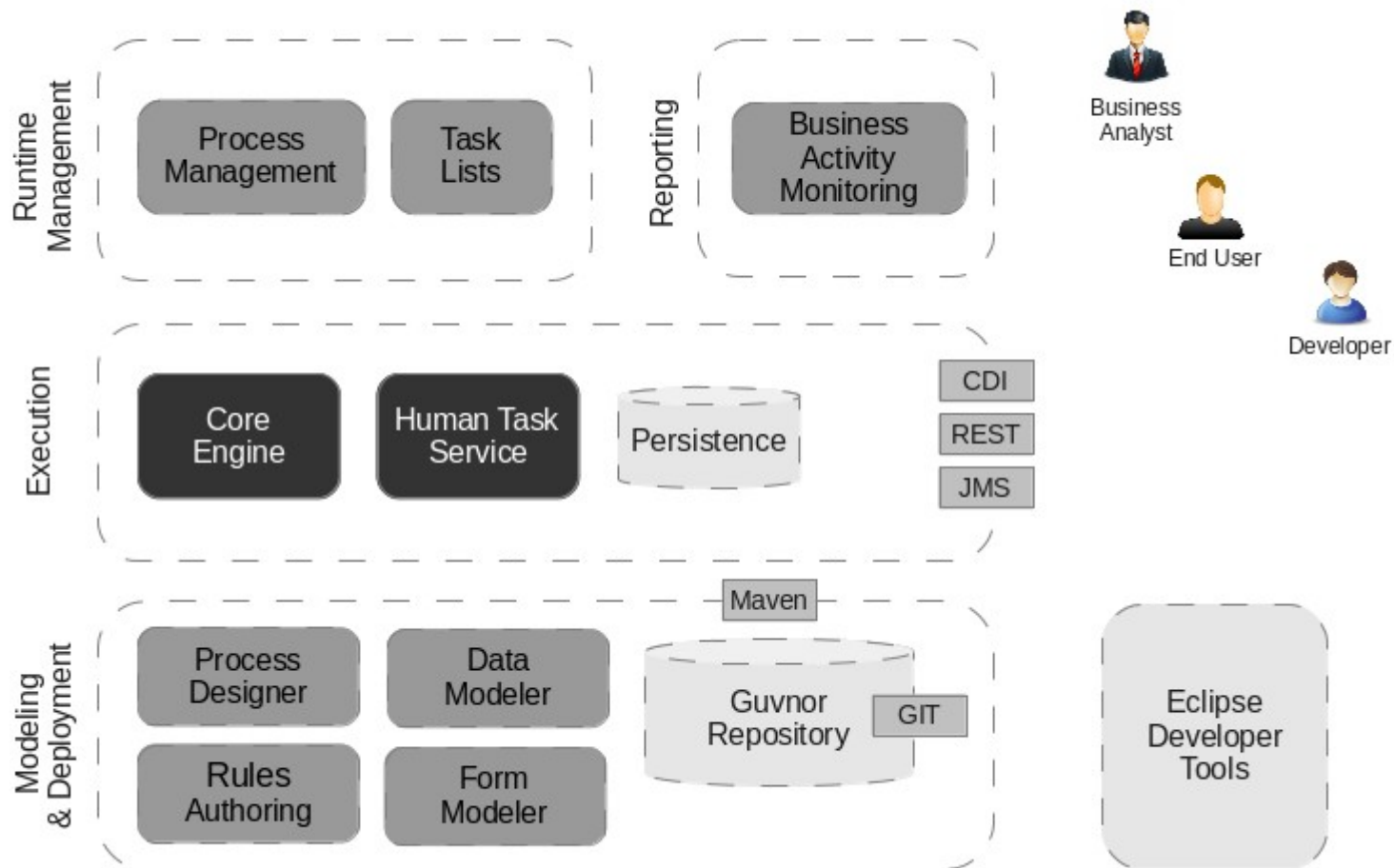
# Autres fonctionnalités

---

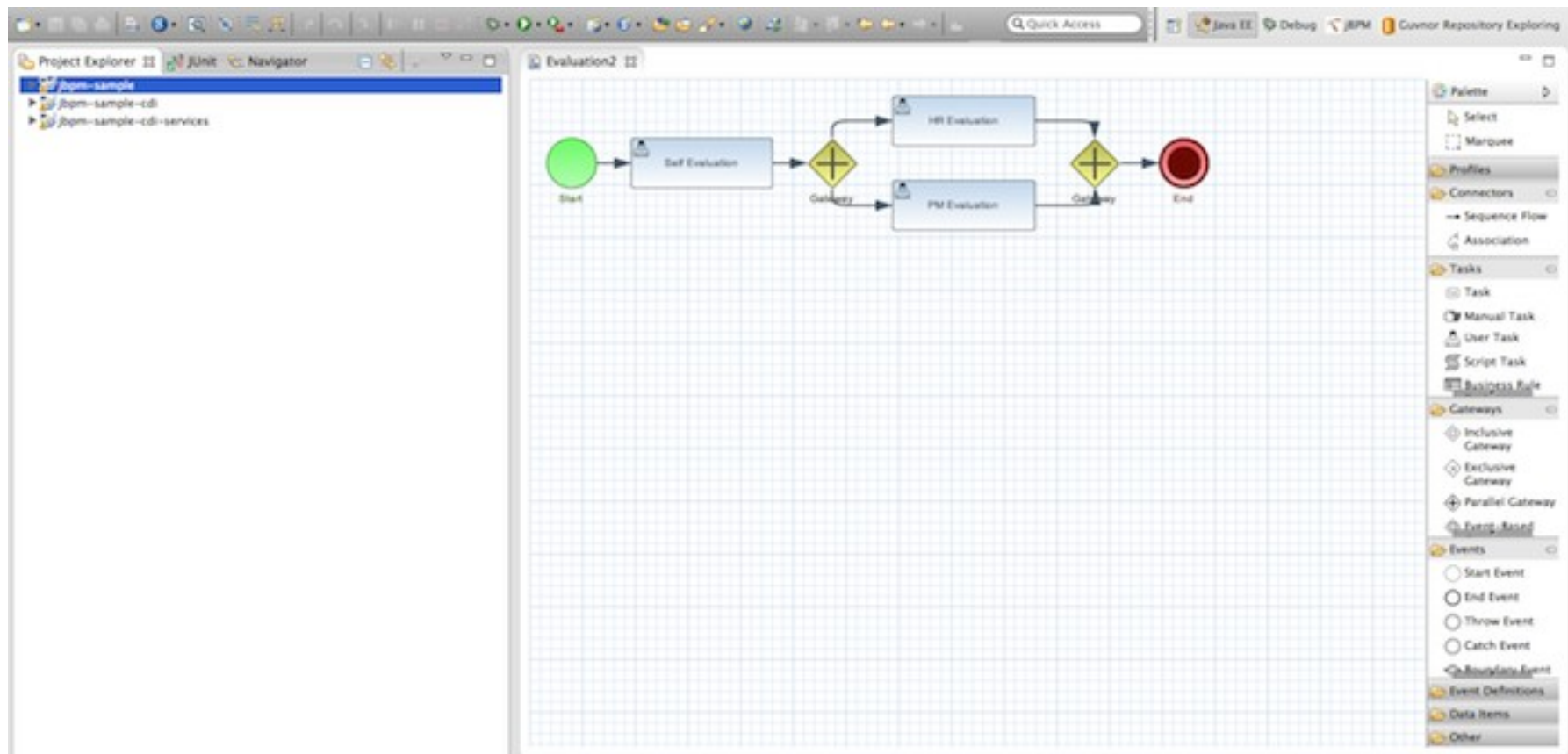
La distribution propose également :

- ✓ Des éditeurs graphiques pour modéliser les processus (plug-in Eclipse et interface web).
- ✓ Couche de persistance et de transaction adaptable basée sur JPA et JTA
- ✓ Service de tâches pour workflow humain adaptable
- ✓ Console d'administration web
- ✓ Service de journalisation d'évènements optionnel
- ✓ Intégration avec CDI, Spring, OSGI, ...
- ✓ Possibilité d'extension (définition de nouveaux nœuds)
- ✓ Intégration des règles métier via Drools
- ✓ Workflow flexible et dynamique

# Composants *jBPM*



# Éditeur Eclipse





# Plugin Eclipse

---

Le plugin *jBPM* (partie intégrante de Jboss Tools) offre :

- ✓ Des assistants pour la création de projet
- ✓ Génération de codes exemple
- ✓ L'éditeur graphique
- ✓ Validation d'une définition de processus
- ✓ Vue d'audit pour l'analyse de l'exécution



# Introduction

---

Modèle Workflow  
Fonctionnalités *JBPM*  
**Projets KIE**  
Séquence de démarrage



# Projets *KIE*

---

***KIE*** regroupe plusieurs projets RedHat liés à la logique métier

Les projets partagent la même API et les mêmes techniques de stockage d'assets et de déploiement (basées sur *Maven* et *Git*)

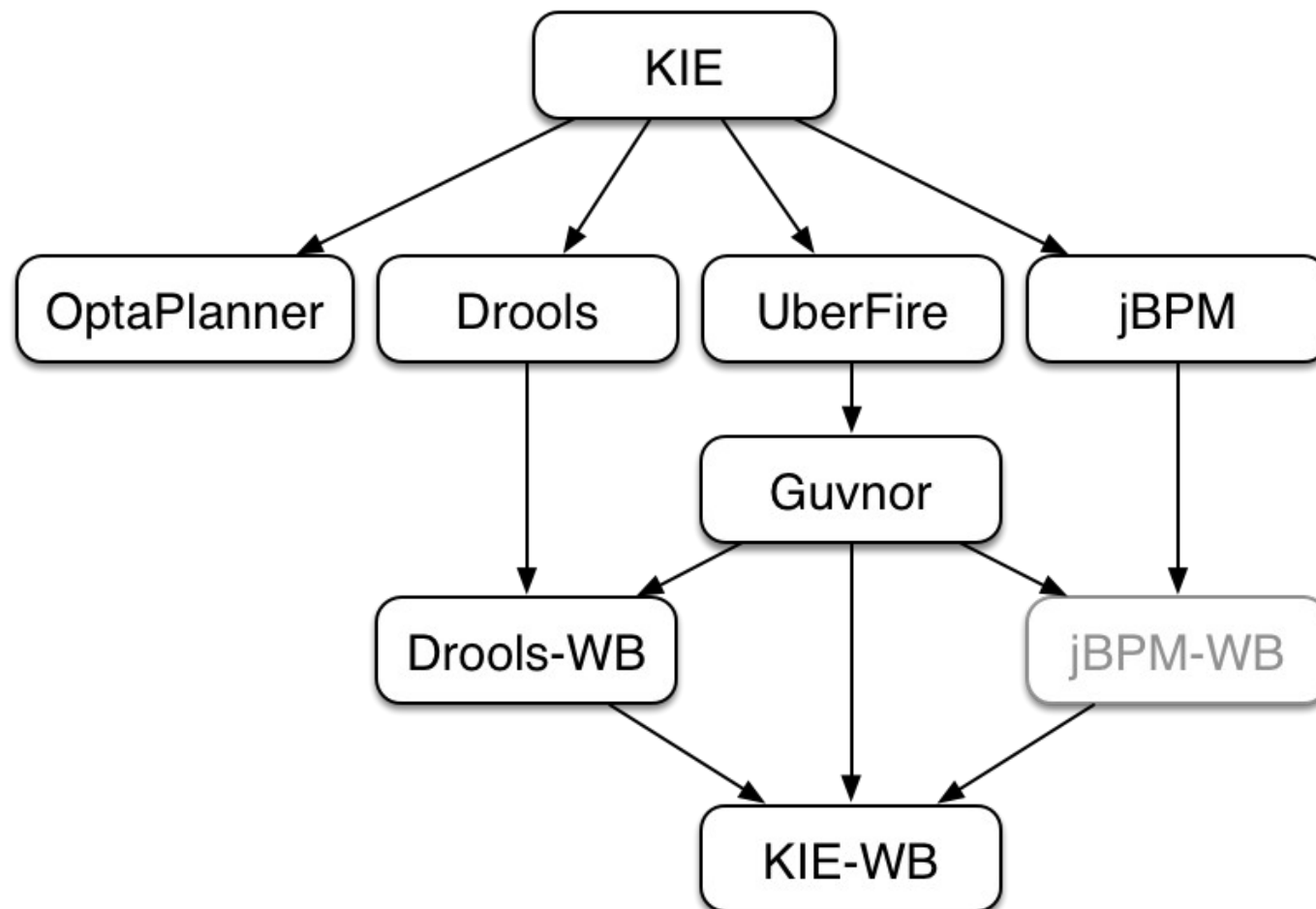
***jBPM*** : Moteur de workflow basé sur BPMN2.0

***Drools*** : Gestion des règles métier

***OptaPlanner*** (anciennement *Drools Planner*) heuristique d'optimisation

***UberFire/Guvnor*** framework de construction d'interface Web à la façon d'Eclipse. Les perspectives/vues/panneaux sont gérés par des plugins

# Projets KIE







# Cycle de vie des projets KIE

---

**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** : Test de la connaissance avant déploiement

**Déploiement** : Déploiement à un emplacement utilisable par les applications clientes (Repository **Maven**)

**Utilisation** : Le jar est chargé par un **KieContainer**. Pour interagir avec le système, une application crée une **KieSession**

**Exécution** : Appels de l'API de *KieSession*

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

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



# Kie API

---

***KieServices*** : Singleton, hub d'accès aux autres services

***KieContainer***: Permet de charger un *KieModule* et ses dépendances.

***KieModule*** : Module de connaissance. Définit les *KieBase* et *KieSession*

***KieBase*** : Base de connaissance.  
Contient des ressources comme les processus jBPM ou les règles Drools.  
Configure les *KieSessions* utilisable

***KieSession*** : Interaction avec la base de connaissance



# *KieServices*

---

**// Getting reference to kieServices**

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

**// Acces to a Container**

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

**// Acces to a Logger Factory**

```
KieLoggers loggers = KieServices.getLoggers()
```



# *KieContainer*

---

Différents ***KieContainer***, correspondant à différentes façon de charger les modules Kie, existent :

- Via le classpath
- Via un dépôt maven
- Via une API REST



# *KieModule*

---

***KieModule*** encapsule différentes :

- *KieBase* : Représente une base de connaissance. Dans le cas de jBPM contient les fichiers BPMN2.0 de définition de processus
- *KieSessions* : Représente les modes d'interactions avec le moteur. Configure la persistance, les services externes au moteur, etc..

La configuration de *KieModule* est effectuée :

- Via un descripteur *META-INF/kmodule.xml*
- Ou programmatiquement

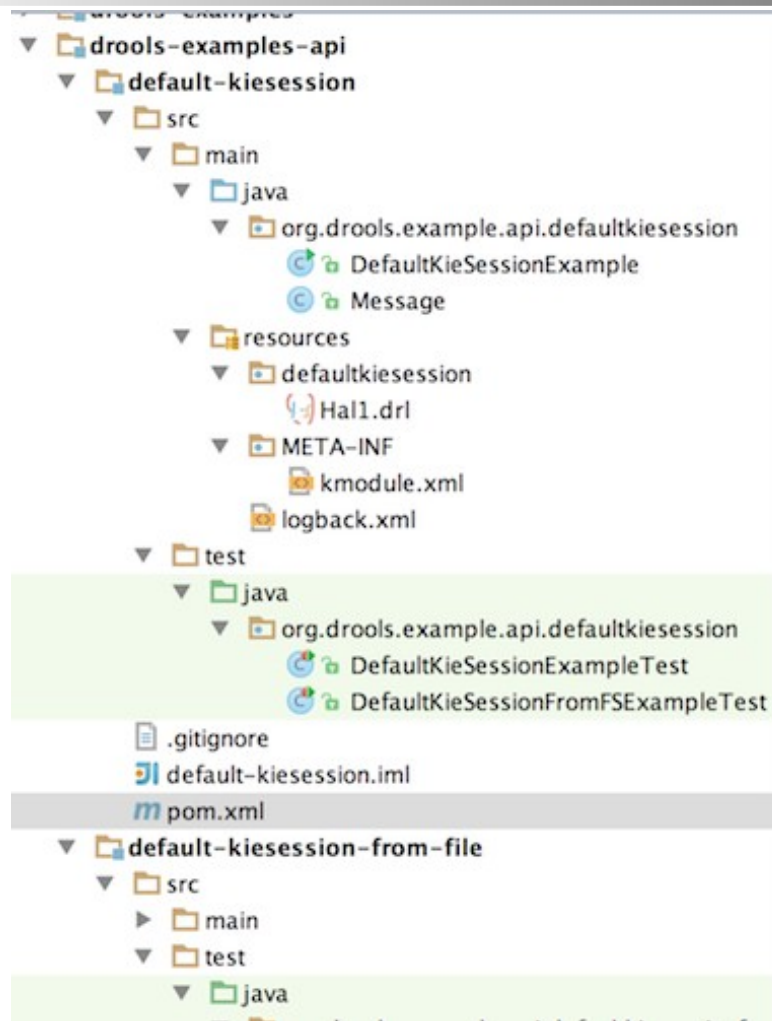


# Projet Maven

---

Typiquement, un projet KIE est un projet Maven avec :

- un ***pom.xml*** qui inclut le *KieRuntime*
- Un descripteur ***META-INF/kmodule.xml*** qui
  - Liste les définitions de processus, règles, etc.
  - Configure les bases de connaissances et les types de session qui peuvent être créées
- Un plugin Maven permet de valider les ressources durant le build



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.drools</groupId>
    <artifactId>drools-examples-api</artifactId>
    <version>6.0.0</version>
  </parent>

  <artifactId>default-kiesession</artifactId>
  <name>Drools API examples - Default KieSession</name>

  <dependencies>
    <dependency>
      <groupId>org.drools</groupId>
      <artifactId>drools-compiler</artifactId>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.drools</groupId>
        <artifactId>drools-maven-plugin</artifactId>
        <version>6.0.0</version>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>
</project>
```



# Convention versus configuration

---

La configuration la plus simple est un *kmodule.xml* vide

=> Dans ce cas, au démarrage KIE scan le classpath puis :

- Ajoute toutes les ressources métier trouvées dans la base de connaissance par défaut
- Configure une session par défaut est associée à la base de connaissance





# Packaging

---

Le projet est donc packagé sous forme de JAR ou KJAR qui contient :

- Le descripteur de module :  
*META-INF/kmodule.xml*
- Les ressources métier : business rules et processus
- Les classes modèle du domaine
- Les classes de test classes
- Le code client et le runtime Kie



# Introduction

---

Modèle Workflow  
Fonctionnalités *jBPM*  
Projets KIE  
**Séquence de démarrage**

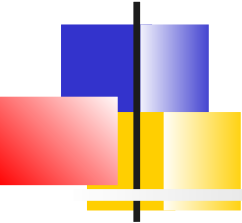


# Chargement des définitions

---

La méthode recommandée pour charger les définitions a évolué avec les releases de jBPM :

- jBPM6 : Utilisation de *KieService* et *KieContainer* pour charger les ressources à partir du classpath ou d'un dépôt distant,
- jBPM6 et 7 : Utilisation de *RuntimeManager* en spécifiant la stratégie de cycle de vie des sessions



# Création Knowledge Base (release jBPM6)

---

// Singleton of Kie services

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

// Scan of META-INF/kmodule.xml

// Scan for business resources bpmn, drl or other

// Create knowledge bases specified in kmodule.xml

```
KieContainer kContainer =  
    kieServices.getKieClasspathContainer();
```

// Get one knowledge base

```
KieBase kbase = kContainer.getKieBase("kbase");
```



# *RuntimeManager*

---

***RuntimeManager*** a été introduit pour faciliter la gestion des sessions.

Disponible depuis jBPM6, il est recommandé en jBPM7

Il permet de choisir entre 3 stratégies de gestion du cycle de vie d'une *KieSession* :

- **Singleton** : La même session est utilisée pour toutes les instances de processus
- **Request** : Une nouvelle session est créé à chaque requête
- **Process instance** : Une session est associé à une instance de processus



# Encapsulation

---

*RuntimeManager* permet d'accéder à ***RuntimeEngine*** qui encapsule entre autres :

- *KieSession*
- *TaskService* : Voir Workflow Humain
- *AuditService*<sup>1</sup> : Accès aux exécutions

L'utilisation de l'API *RuntimeManager* garantit le même procédé de chargement de session dans tout le code applicatif

1. *AuditService* est disponible dans les versions 7.x



# Séquence typique

---

- Au démarrage : Construction de *RuntimeManager* et conservation du singleton pendant toute la durée de l'application
- A la demande :
  - get *RuntimeEngine* from *RuntimeManager*
  - get *KieSession* and/or *TaskService* à partir de *RuntimeEngine*
  - Opérations sur *KieSession* ou *TaskService*. Ex : *startProcess*, *completeTask*, etc
  - Disposer le *RuntimeEngine* en utilisant *RuntimeManager.disposeRuntimeEngine* method
- A l'arrêt : fermeture de *RuntimeManager*



# Example

---

```
// first configure environment that will be used by RuntimeManager
RuntimeEnvironment environment = RuntimeEnvironmentBuilder.Factory.get()
    .newDefaultInMemoryBuilder()
    .addAsset(ResourceFactory.newClassPathResource("BPMN2-ScriptTask.bpmn2"), ResourceType.BPMN2)
    .get();

// next create RuntimeManager - in this case singleton strategy is chosen
RuntimeManager manager =
RuntimeManagerFactory.Factory.get().newSingletonRuntimeManager(environment);

// then get RuntimeEngine out of manager - using empty context as singleton does not keep track
// of runtime engine as there is only one
RuntimeEngine runtime = manager.getRuntimeEngine(EmptyContext.get());

// get KieSession from runtime runtimeEngine - already initialized with all handlers, listeners,
etc that were configured
// on the environment
KieSession ksession = runtimeEngine.getKieSession();

// add invocations to the jBPM engine here,
// e.g. ksession.startProcess(processId);

// and last dispose the runtime engine
manager.disposeRuntimeEngine(runtimeEngine) ;
```





# *RuntimeEnvironmentBuilder*

---

***RuntimeEnvironmentBuilder*** fournit des builders pré-configurés pour simplifier la création de *RuntimeManager*.

- *newEmptyBuilder()* : Builder complètement vide, la configuration doit être effectuée manuellement
- *newDefaultBuilder()* : Runtime avec un environnement par défaut
- *newDefaultInMemoryBuilder()* : Sessions non persistantes
- *newClasspathKmoduleDefaultBuilder()*  $\Leftrightarrow$  *KieClasspathContainer*, chargement des ressources à partir du classpath



# Prise en main

---

## **Installation**

API du moteur et Test  
Introduction à BPMN2



# Installation de l'IDE

---

L'IDE recommandé est le plugin Eclipse jBPM qui fournit des assistant et un éditeur graphique

Pour le mettre en place, télécharger ***jbpm-installer*** qui installe :

- Une installation Eclipse avec les plugins jBPM et un runtime jBPM pré-configuré
- Le modeleur Eclipse BPMN2
- Un serveur Wildfly



# Prise en main

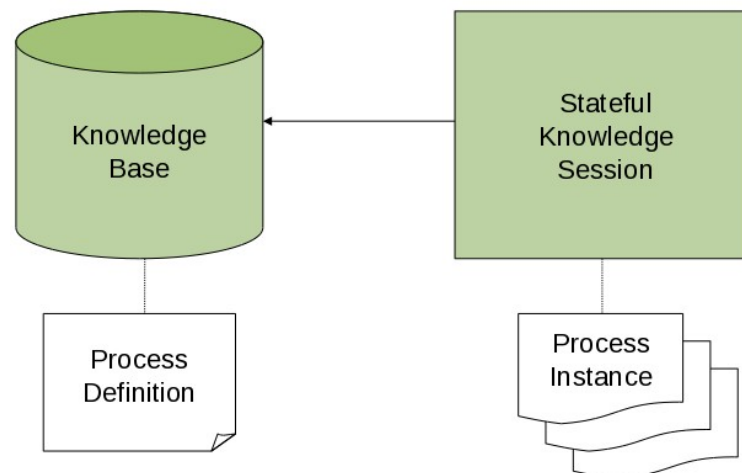
---

Installation  
**API du moteur et Test**  
Introduction à BPMN2

# Objets cœurs

L'interaction avec le moteur de workflow s'effectue via un objet ***KieSession***

L'objet session possède une référence vers une ***base de connaissance (KieBase)*** où sont stockées les définitions de processus





# Séquence typique

---

1) Créer une base de connaissance et charger les définitions de processus à partir du *classpath*, d'un dépôt Maven ou d'une URL

Effectué une fois au démarrage

2) Instancie ou récupère une session existante:

- Une fois
- Pour chaque instance de processus
- Pour chaque requête

3) Créé et démarre une instance de processus

4) Interactions avec l'instance

5) Évènement de fin du processus



# Démarrage processus

---

Pour démarrer une instance il suffit de fournir le nom de la définition de processus :

```
KieSession ksession = kbase.newKieSession();
```

```
ProcessInstance processInstance =  
    ksession.startProcess("com.sample.MyProcess");
```

Les autres APIS :

- ✓ Démarrer avec des paramètres

```
ProcessInstance startProcess(String processId,  
    Map<String, Object> parameters)
```
- ✓ Arrêter un processus

```
void abortProcessInstance(long processInstanceId)
```



# Signaler un processus

---

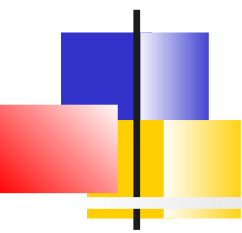
Généralement, un processus atteint un état en attente

Pour progresser, il doit être signalé avec un événement (un type et des données associées)

L'API permet de signaler un processus spécifique ou tous les processus

```
void signalEvent(String type, Object event, long processInstanceId) ;  
void signalEvent(String type, Object event);
```





# Récupérer des informations

---

- ✓ Tous les processus :

```
Collection<ProcessInstance> getProcessInstances();
```

- ✓ Un process via son id

```
ProcessInstance getProcessInstance(long  
processInstanceId);
```



# Clés de corrélation

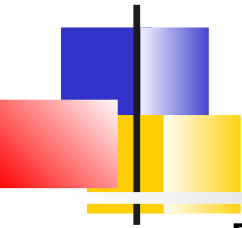
---

Il est possible d'associer un *id* métier avec une instance de processus.

Une application cliente peut alors récupérer les données d'un processus via la clé de corrélation plutôt que *l'id* du process instance

Voir :

***CorrelationAwareProcessRuntime***



# *CorrelationAwareProcessRuntime*

---

```
ProcessInstance startProcess(String processId,  
    CorrelationKey correlationKey,  
    Map<String, Object> parameters);
```

```
ProcessInstance createProcessInstance(String processId,  
    CorrelationKey correlationKey,  
    Map<String, Object> parameters);
```

```
ProcessInstance getProcessInstance(CorrelationKey  
    correlationKey);
```



# Événements et *listeners*

---

Il est possible d'enregistrer des listeners d'événements processus.

Les listeners implémentent

***ProcessEventListener*** qui permet d'exécuter du code lorsque :

- Un process démarre ou s'interrompt
- Un process entre ou sort d'un nœud
- Lorsque les données associées à un process sont mis à jour

Les méthodes prennent un *ProcessEvent* comme argument



# Listener fournit

---

*JBPM* fournit un *ProcessEventListener* dédié au debug.

Ce listener trace tous les événements processus et les écrit :

- Sur la console
- Dans un fichier XML qui peut être visualisé par le plugin Eclipse
  - Version Non-threadé : Les résultats sont visibles seulement lorsque le logger est fermé
  - Version Threadé : Les résultats sont visibles dès qu'ils sont produits



# Example

---

```
KnowledgeRuntimeLogger logger =  
KnowledgeRuntimeLoggerFactory.newFileLogger( ksession, "test" );  
  
KieRuntimeLogger logger =  
    KieServices.Factory.get().getLoggers().newThreadedFileLogger(ksess  
ion, "lab2", 1000);  
  
// add invocations to the process engine here,  
// e.g. ksession.startProcess(processId);  
  
...  
  
logger.close();
```

```
▼ RuleFlow started: ruleflow[com.sample.ruleflow]  
  ▼ RuleFlow node triggered: Start in process ruleflow[com.sample.ruleflow]  
    ▼ RuleFlow node triggered: Hello in process ruleflow[com.sample.ruleflow]  
      ▼ RuleFlow node triggered: End in process ruleflow[com.sample.ruleflow]  
        RuleFlow completed: ruleflow[com.sample.ruleflow]
```



# Support pour le test

---

jBPM fournit du support pour le test

La classe helper

***JbpmJUnitBaseTestCase*** simplifie les tests unitaires permettant de vérifier les différents chemins d'un processus

```
<dependency>  
  <groupId>org.jbpm</groupId>  
  <artifactId>jbpm-test</artifactId>  
  <version>${runtime.version}</version>  
</dependency>
```



# *JbpmJUnitBaseTestCase*

---

- Constructeur qui permet de spécifier si la persistance et une source de données sont utilisées durant les tests
- Assertions orientée processus
- Méthodes Helper pour créer *RuntimeManager*.

Par exemple :

```
createRuntimeManager(String... fileNames)
```

qui prend en arguments les noms des fichiers de définition de processus





# *JbpmJUnitBaseTestCase*

---

Assertions supplémentaires :

```
assertProcessInstanceActive(long procInstId,  
    StatefulKnowledgeSession ksession) : Test if process is active  
  
assertProcessInstanceCompleted(long procInstId,  
    StatefulKnowledgeSession ksession) : Test if process stopped without  
    error  
  
assertProcessInstanceAborted(long procInstId,  
    StatefulKnowledgeSession ksession) : Test if process has been aborted  
  
assertNodeActive(long procInstId, StatefulKnowledgeSession  
    ksession, String... name): Test if the process is in one of the nodes  
    provided by the parameters  
  
assertNodeTriggered(long procInstId, String... nodeNames) : Test if  
    all the nodes provided haven been activated  
  
getVariableValue(String name, long processInstanceId,  
    StatefulKnowledgeSession ksession) : Retrieve a variable value
```



# Exemple

---






```
public class MyProcessTest extends JbpmJUnitBaseTestCase {  
    public void testProcess() {  
        // création de session et charge le processus  
        RuntimeManager manager = createRuntimeManager("Evaluation.bpmn");  
        RuntimeEngine engine = getRuntimeEngine(null);  
        KieSession ksession = engine.getKieSession();//  
        Démarrage du processus  
        ProcessInstance processInstance =  
            ksession.startProcess("com.sample.bpmn.hello");  
        // Le processus s'est-il terminé sans erreur  
        assertProcessInstanceCompleted(processInstance.getId(), ksession);  
        // Les noeuds ont-ils été déclenchés  
        assertNodeTriggered(processInstance.getId(), "StartProcess", "Hello",  
            "EndProcess");  
    }  
}
```



# Eclipse plugin : Audit view

---

```
KieRuntimeLogger logger =  
    KieServices.Factory.get().getLoggers().newFileLogger(ksession,  
        "test");  
// add invocations to the process engine here,  
// e.g. ksession.startProcess(processId);  
...  
logger.close()
```

- ▼  RuleFlow started: ruleflow[com.sample.ruleflow]
- ▼  RuleFlow node triggered: Start in process ruleflow[com.sample.ruleflow]
- ▼  RuleFlow node triggered: Hello in process ruleflow[com.sample.ruleflow]
- ▼  RuleFlow node triggered: End in process ruleflow[com.sample.ruleflow]
-  RuleFlow completed: ruleflow[com.sample.ruleflow]



# Prise en main

---

Installation  
API du moteur et Test  
**Introduction à BPMN2**



# Introduction

---

BPMN2 est une spécification OMG tournée vers les experts métier et les experts technique

Il propose :

- une notation graphique destinée à être manipulés par les experts métiers.
- Un modèle XML permettant l'échange de définitions de processus entre systèmes
- Un mapping vers un langage d'exécution :  
WSBPEL



# Noeuds

---

Les processus BPMN sont constituées de différents types de noeuds connecté entre eux avec des *sequence flows*.

Les principaux types de nœuds sont :

- **Événements** : Nœud en attente d'un événement ou signal (Exemple : expiration d'un timer)
- **Activités** : Action devant être effectuée (Exemple : Tâche utilisateur, traitement système)
- **Gateways** : Nœud permettant de diviser ou regrouper différents chemins du graphe



# Événements

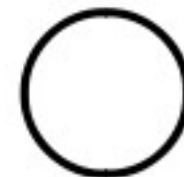
---

Un événement est quelque chose qui se produit durant l'exécution du processus. Il affecte le parcours de l'exécution et a généralement :

- Un déclencheur ou cause (trigger)
- Un impact ou résultat

Il y a trois types d'événement en fonction de leur moment d'occurrence :

- Événement de démarrage
- Événement intermédiaire
  - Événements de type catch/throw
  - Événements de type signaux
- Événement de fin



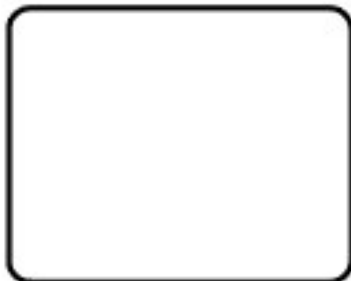


# Activité

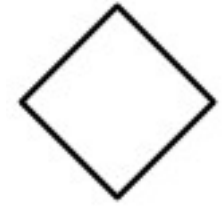
---

Une **activité** est un terme générique pour un travail qu'effectue une entreprise dans le cadre d'un processus.

Une activité peut être *atomique* (tâche) ou *composée* (sous-processus).







# Gateway

Une **gateway** est utilisée pour contrôler la divergence ou convergence des chemins du graphe.

Il permet de décrire des exécutions parallèle puis de les synchroniser

Une gateway peut être :

- Divergeant : Exclusive, Inclusive, ...
- Convergeant : Exclusive, Inclusive, ...



# Sequence flow

---

Un ***Sequence Flow*** (transition) est utilisé pour montrer l'ordre dans lequel seront exécutées les activités du processus





# Message flow

---

Un **Message Flow** est utilisé pour montrer le flux de messages entre 2 participants

Avec BPMN, les participants sont représentés par des pools (ou swimlanes)



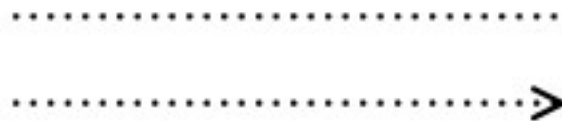


# Association

---

Une **Association** est utilisée pour relier des artefacts (informations) avec des éléments BPMN.

Par exemple des annotations peuvent être associées à des activités, gateways, ...





# Pool

---

Un **Pool** permet de classifier certaines activités du processus.

Il est sont utilisés pour grouper toues les activités d'un sous-objectif du processus

Name	
------	--



# Lane (couloir)

---

Un ***Lane*** (couloir) est une partition à l'intérieur d'un processus ou d'un pool qui s'étend sur l'ensemble du processus.

Il groupe les activités effectués par un même acteur.





# Données

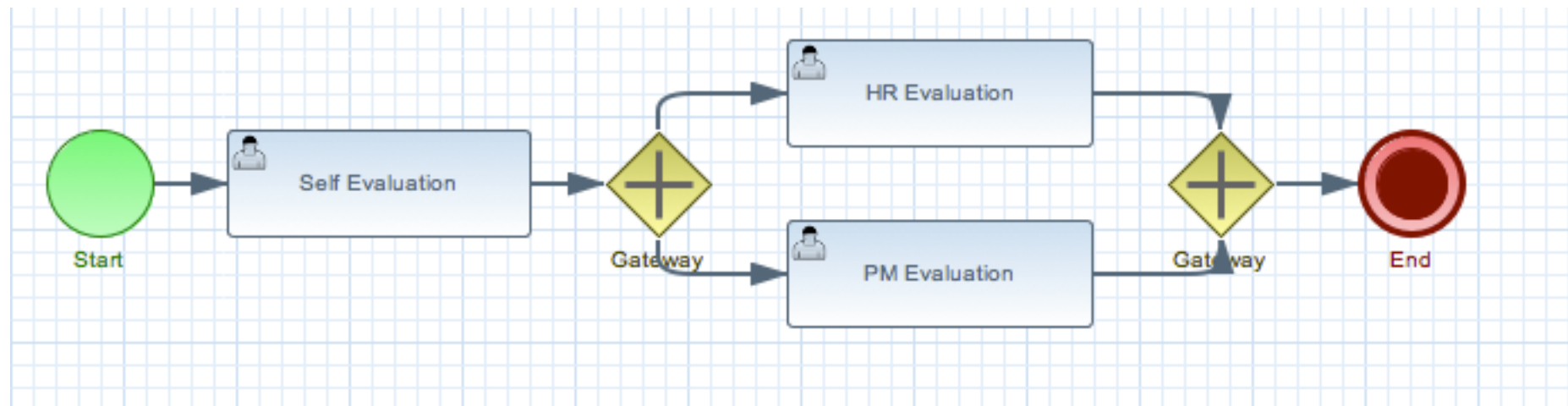
---

Les éléments ***données*** représentent les informations nécessaires pour l'exécution d'une activité ou les informations produites par une activité.

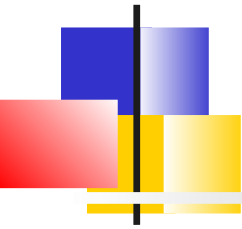
Il peut s'agir d'un type métier



# Exemple







# Modélisation avec jBPM

---

## **JPMN et BPMN2.0**

Noeuds disponibles

Données

Persistance et transactions

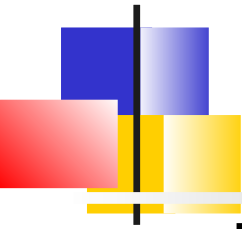


# Mise au point du graphe

---

Plusieurs éditeurs sont disponibles pour construire des graphes BPMN2.0 utilisés par *JBPM*

- L'éditeur graphique du plug-in Eclipse
- L'éditeur Web fourni par la console jbpmp
- L'édition directe du fichier XML
- L'API Java fournie par *JBPM*



# Editeur Eclipse

---

L'éditeur Eclipse utilise la syntaxe XML BPMN 2.0 (incluant BPMNDI pour les informations graphiques).

- Il supporte la plupart des constructions et des attributs BPMN 2.0 (incluant les lanes, pools, annotations et tous les types de neoud BPMN2 types).
- Il ajoute du support aditionnes pour les attributs personnalisés que jBPM a pu introduire
- Configurable : Il permet de spécifier les nœuds et éléments disponibles dans l'éditeur



# Fichier XML

---

L'éditeur graphique s'appuie sur un fichier XML respectant le schéma BPMN 2.0

<http://www.omg.org/spec/BPMN/20100524/MODEL/BPMN20.xsd>

Ce fichier est également éditable directement

Il contient 2 parties :

- ✓ Une partie processus spécifiant les nœuds, leurs connexions et leurs propriétés
- ✓ Une partie graphique (*<bpmndi:>*) qui définit les coordonnées des différents éléments graphiques



# Example

```
<definitions id="Definition" targetNamespace="http://www.jboss.org/drools" typeLanguage="http://www.java.com/javaTypes"
  expressionLanguage="http://www.mvel.org/2.0" xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" Rule Task xmlns:xsi="
  http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL BPMN20.xsd"
  xmlns:g="http://www.jboss.org/drools/flow/gpd" xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI" xmlns:dc="
  http://www.omg.org/spec/DD/20100524/DC" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
  xmlns:tns="http://www.jboss.org/drools">

  <process processType="Private" isExecutable="true" id="com.sample.hello" name="HelloProcess" >

    <!-- nodes -->

    <startEvent id="_1" name="Start" />

    <scriptTask id="_2" name="Hello" ><script>System.out.println("Hello World");</script></scriptTask>

    <endEvent id="_3" name="End" ><terminateEventDefinition/> </endEvent>

    <!-- connections -->

    <sequenceFlow id="_1_2" sourceRef="_1" targetRef="_2" />
    <sequenceFlow id="_2_3" sourceRef="_2" targetRef="_3" />

  </process>

  <bpmndi:BPMNDiagram>

    <bpmndi:BPMNPlane bpmnElement="com.sample.hello" >

      <bpmndi:BPMNShape bpmnElement="_1" > <dc:Bounds x="16" y="16" width="48" height="48" /> </bpmndi:BPMNShape>
      <bpmndi:BPMNShape bpmnElement="_2" > <dc:Bounds x="96" y="16" width="80" height="48" /> </bpmndi:BPMNShape>
      <bpmndi:BPMNShape bpmnElement="_3" > <dc:Bounds x="208" y="16" width="48" height="48" /> </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge bpmnElement="_1_2" > <di:waypoint x="40" y="40" /> <di:waypoint x="136" y="40" /> </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge bpmnElement="_2_3" > <di:waypoint x="136" y="40" /> <di:waypoint x="232" y="40" /> </bpmndi:BPMNEdge>

    </bpmndi:BPMNPlane>

  </bpmndi:BPMNDiagram>

</definitions>
```



# Process API

---

Il est possible de définir un processus en utilisant directement l'API et plus particulièrement les packages :

- ✓ *org.jbpm.workflow.core*
- ✓ *org.jbpm.workflow.core.node*

La classe *RuleFlowProcessFactory* offre les méthodes nécessaires pour ajouter des nœuds, des connexions et de spécifier leurs propriétés.



# Exemple

---

```
RuleFlowProcessFactory factory =
    RuleFlowProcessFactory.createProcess("org.jbpm.HelloWorld");
factory
// Entête
.name("HelloWorldProcess")
.version("1.0")
.packageName("org.jbpm")
// Noeuds
.startNode(1).name("Start").done()
.actionNode(2).name("Action")
.action("java", "System.out.println(\"Hello World\");").done()
.endNode(3).name("End").done()
// Transitions
.connection(1, 2)
.connection(2, 3);
RuleFlowProcess process = factory.validate().getProcess();
```



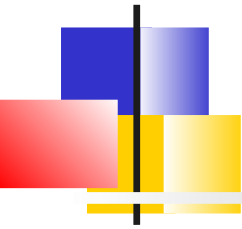
# Propriétés d'un processus

---

Les propriétés d'un processus sont

- ***Id*** : Son identifiant unique
- ***Name*** : Le nom ou libellé.
- ***Version*** : Le numéro de version
- ***Package*** : Le package (ou espace de noms) dans lequel est défini le processus
- ***Variables*** : Des variables peuvent être définies pour stocker des données liées au processus
- ***Couloirs d'activités ou swimlanes*** : Spécifie l'acteur pour l'exécution d'une tâche humaine
- ***Des gestionnaires d'exception*** : Code gérant les situations exceptionnelles





# Modélisation avec jBPM

---

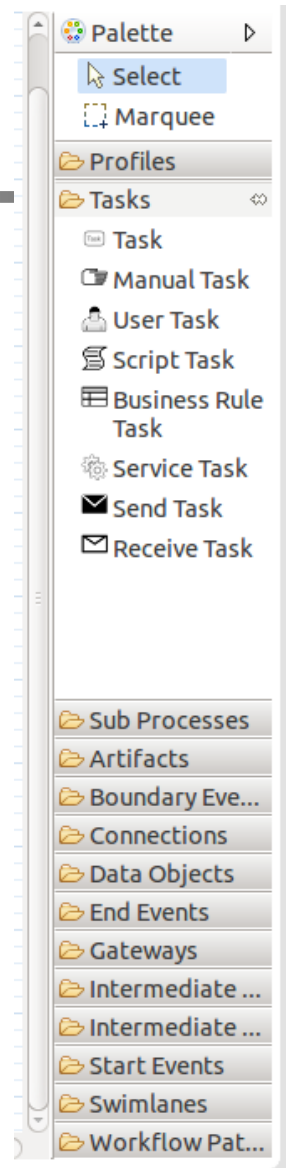
JPMN et BPMN2.0

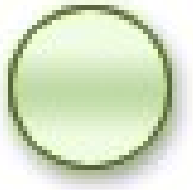
**Noeuds disponibles**

Données

Persistance et transactions

# Palette





# Noeud ou événements Start

---

## **Start node (Start event) :**

Un process a un seul nœud de départ avec aucune transition rentrante et une transition sortante.

Lorsqu'un processus est démarré, l'exécution du graphe démarre sur ce nœud et continue directement au nœud lié.

Un nœud de départ a les propriétés ***id*** et ***name***.



# Noeuds de fin (Evènements)

---

## **Nœud de fin (End event) :**

Un processus a un ou plusieurs nœud de fin.

Un nœud de fin a une seule transition entrante et aucune transition sortante.

Un nœud de fin comporte une propriété *Terminate* indiquant si ce nœud marque la fin du processus complet ou seulement du chemin courant.

Un processus est terminé si il n'y a plus de chemins actifs dans le graphe





# Événements erreur

---

**Erreur/Escalade** : Permet de modéliser une exception dans le processus.

Une transition entrante et aucune transition sortante.

Lorsque ce nœud est atteint, une erreur du même nom est lancé. Le moteur recherche un gestionnaire d'erreur capable de traiter l'exception, si aucun gestionnaire n'est trouvé, le processus s'arrête.

Un nœud erreur comporte les propriétés additionnelles suivantes :

- *Error/Escalation Type*: Le nom de l'erreur, utilisé pour chercher le gestionnaire adéquat.
- *Error/Escalation Data*: Le nom de la variable qui contient les données associées à l'erreur ou une expression. Ces données sont également fournies au gestionnaire.





# Évènements « Boundary »

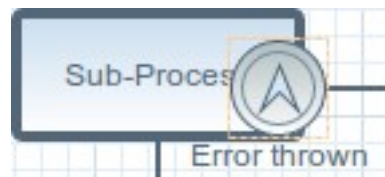
---

## Évènements de frontière :

Ces événements sont placés à cheval sur un nœud activité.

Ils n'ont qu'une transition sortante.

Ils sont utilisés pour traiter les exceptions lancées par un sous-composant.





# Événement intermédiaire : signal

---

**Signal** : Un événement signal de type catch utilisé pour répondre à un événement interne ou externe du processus.

Il a une transition entrante et une transition sortante.

Il définit le type d'événement auquel il répond et éventuellement le nom du variable qui contiendra les données associées à l'événement

Pour signaler un événement via l'API :

```
ksession.signalEvent(eventType, data,  
    processInstanceId)
```

=> déclenchera tous les nœuds « signal » associés au type d'événement *eventType*



# Événement intermédiaire : timer

**Timer** : Un timer pouvant se déclencher une ou plusieurs fois après un certain temps.

C'est un événement de type catch

Il doit avoir une transition entrante et une transition sortante.

Un timer a les propriétés additionnelles suivantes :

- Délai : [#d][#h][#m][#s][#[ms]]
- Période : 0 pour une seule fois ou [#d][#h][#m][#s][#[ms]]







# Activité script

---

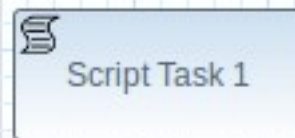
**Script** : Représente un script à exécuter.

Un nœud script a une transition entrante et une transition sortante.

La propriété **action** spécifie en java ou *mvel* ce qui doit être exécuté.

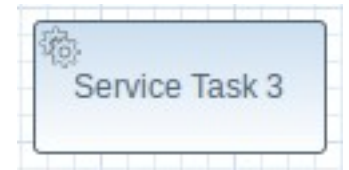
Le code de l'action a accès à toutes les variables du processus et en particulier à la variable prédéfinie *kcontext* (*ProcessContext*) qui permet de récupérer et manipuler tous les objets du processus.

Lorsqu'un nœud script est atteint, l'action correspondante est exécutée, puis le processus est placé dans le nœud sortant.





# Activité service



**Service ou unité de travail** : Représente un appel à un service externe effectuant une tâche atomique.

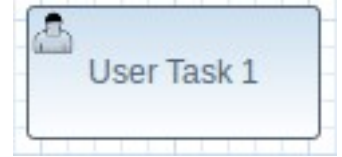
La configuration définit les paramètres d'entrée et de sortie de l'activité.

Un nœud service contient une transition entrante et une transition sortante.

Les propriétés additionnelles de ce type de nœud sont donc :

- ✓ **Mapping des paramètres** : permet de copier des variables du processus dans les paramètres d'entrée
- ✓ **Mapping du résultat** : Permet de copier le résultat du service dans des variables de processus
- ✓ **Actions** : Il est possible de définir des actions exécutées à l'entrée et à la sortie du nœud

# Tâche humaine



**Nœud tâche (Human task)** : Représente une tâche devant être exécutée par un utilisateur.

Il contient une transition entrante et une transition sortante.

Il peut être placé avec un couloir d'activité permettant de désigner le responsable de la tâche.

Un nœud tâche est une unité de travail particulière.



# Sous-processus



Call Activity 1

---

**Sous-Process (Reusable Sub-process)** : représente l'appel d'un sous-processus externe à l'intérieur du processus courant.

Le nœud a une connexion entrante et une connexion sortante.

Lorsque le nœud est atteint, le moteur démarre le sous-processus ayant l'identifiant fourni. Le nœud offre les propriétés supplémentaires suivantes :

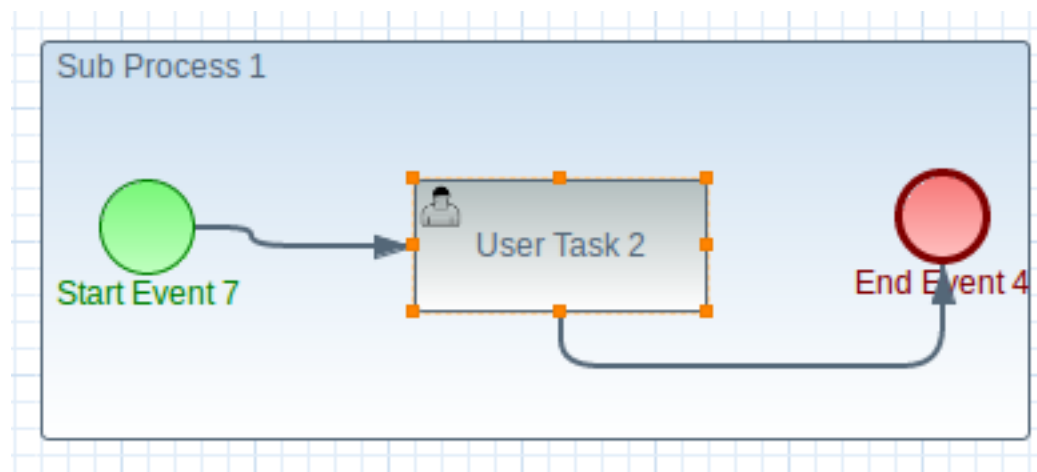
- **Wait for completion** : Indique si le processus parent doit attendre la fin du processus fils pour continuer son exécution
- **Independent** : Le sous-processus est indépendant du processus parent. La fin du processus parent ne provoque pas la fin du processus fils
- **Actions** en entrée et en sortie
- **Mapping** des paramètre d'entrée et des résultats

# Noeud composite

**Nœud Composite (Sous-processus embarqué):** Un nœud composite est un conteneur d'autres nœuds. Cela permet de définir des variables ou gestionnaire d'exception contextuelles au groupe de nœud.

Le nœud a une transition entrante et une transition sortante.

Le sous-process doit contenir un nœud de départ et au moins un nœud de fin. En général, les nœuds de fin ne sont pas terminaux car ils termineraient le processus parent.



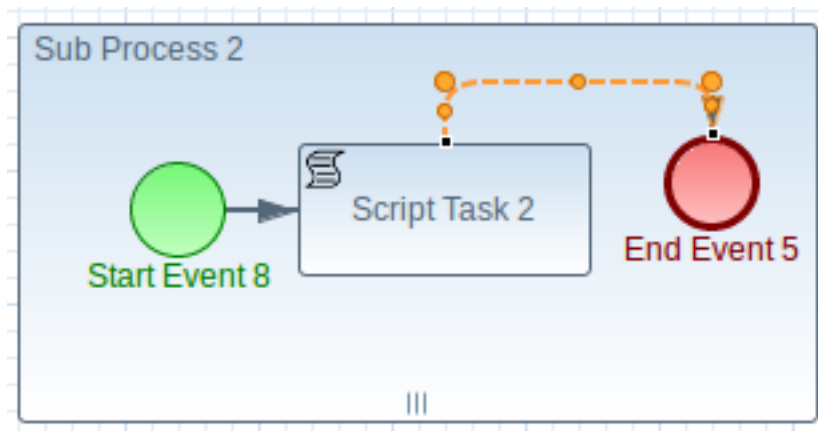
# Itération

**Itération (Multiple-instances sub process)** : Permet d'exécuter la séquence contenue plusieurs fois. Une fois pour chaque élément de la collection fournie.

Le nœud a une transition entrante et une transition sortante.

Le nœud attend l'exécution de la séquence contenue pour chaque élément de la collection avant de continuer. Le nœud offre les propriétés supplémentaires suivantes :

- *CollectionExpression* : Le nom de la variable de type collection
- *VariableName* : Le nom de la variable représentant l'élément courant de l'itération.



# Nœud règle

**Rule Task (ou RuleFlowGroup)** : Représente un ensemble de règles métier devant être évaluées.

Le nœud a une transition rentrante et une transition sortante.

Les règles exprimées dans un fichier séparé *.drl* appartiennent à un groupe spécifié par l'attribut *ruleflow-group*.

Lorsque le nœud est atteint, le moteur démarre l'exécution des règles concernées. L'exécution se propage au nœud suivant dès qu'il n'y a plus de règles actives.



# Gateway diverge

**Diverge** : Permet de créer des branches dans l'exécution du processus.

Le nœud a une transition entrante et plusieurs transitions sortantes.

3 types de nœud Diverge sont supportés :

- **AND** signifie que les branches s'exécutent en parallèle.
- **XOR** signifie qu'une seule des branches sera exécutée. La sélection de la branche se fait en évaluant les contraintes attachées à chaque transition et les priorités entre branches. Il faut s'assurer qu'il y aura toujours une contrainte satisfaite. Les contraintes peuvent être exprimées en code Java ou mvel ou en utilisant le langage de règle de Drools
- **OR** signifie que toutes les transitions dont les contraintes seront satisfaites seront franchies. Il faut également s'assurer qu'il y aura toujours une contrainte satisfaite





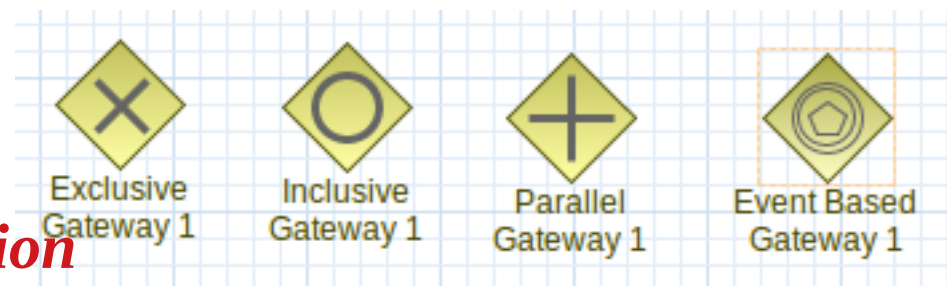
# Gateway converge

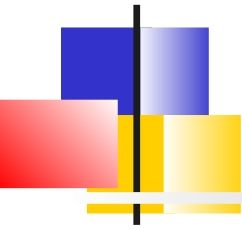
**Converge** : Permet de synchroniser plusieurs branches.

Le nœud a plusieurs transitions entrantes et une transition sortante.

3 types de *join* sont supportés :

- **AND** : Attendre toutes les branches avant de continuer.
- **XOR** signifie que le nœud suivant est activé dès qu'une branche atteint le nœud *join*. Le nœud suivant peut être activé plusieurs fois
- **OR** : Attendre tous les chemins actifs. (Démarré par un Diverge OR)





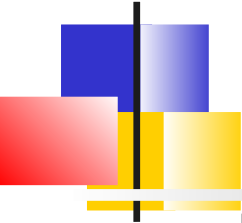
# Modélisation avec jBPM

---

*jPBM* et BPMN2.0  
Nœuds disponibles

**Données**

Persistance et transactions



# Données associées au processus

---

Des données peuvent être associées à une instance de processus par le biais de **variables**.

Durant, l'exécution les valeurs de ces variables peuvent être lues et modifiées.

Une variable est définie par :

- Un **nom**
- Un **type** (type primitif ou objet Java)
- Une **portée (scope)** : Les scopes sont hiérarchiques. Le scope de plus haut-niveau étant le processus. Les sous-processus formant des « sous-scope »

Lorsqu'une variable est accédée, elle est recherchée dans le scope courant puis dans les scopes parents

Il existe également une portée globale (la session) accessible de tous les processus



# Cas d'utilisation

---

- ✓ Les variables de processus peuvent être positionnées au démarrage du processus via une *Map*

```
ProcessInstance startProcess(String procId, Map<String, Object>  
    params)
```

- ✓ Les scripts peuvent accéder aux variables directement par leur nom. Dans ce cas, la variable est locale au script

```
// call method on the process variable "person"  
person.getAge();
```

- ✓ Pour mettre à jour une variable de processus, il faut utiliser la variable pré-définie *kcontext*

```
kcontext.setVariable(variableName, value);
```



# Cas d'utilisation

---

- ✓ Les nœuds *service*, *tâche humaine*, *sous-processus* peuvent échanger des données avec le processus appelant en définissant des associations de paramètres
- ✓ Les nœuds événements peuvent également définir une variable qui récupérera les données passées en paramètre de l'événement
- ✓ Les variables peuvent être utilisées dans une contrainte d'un nœud gateway
- ✓ Les variables globales sont elles positionnées via l'objet session

```
ksession.setGlobal(name, value);  
kcontext.getKnowledgeRuntime().setGlobal(name, value);
```



# Contraintes

---

Il existe 2 types de contraintes :

- ✓ Les contraintes « **code** » sont des expressions booléennes en Java ou MVEL. Elles utilisent en général les variables de processus.

```
return person.getAge() > 20 ;  
return person.age > 20 ;
```

- ✓ Les contraintes « **règle** » utilisent la syntaxe Drools.

Les règles font alors référence à des faits présents dans la mémoire de travail ou à des variables globales.

```
Person( age > 20 )
```



# Action script

---

Les actions script peuvent s'exécuter :

- Dans les nœuds scripts
- A l'entrée et à la sortie de plusieurs types de nœuds

Les action *script* ont accès aux variables via l'objet implicite *kcontext*



# Utilisation de *kcontext*

---

Retrouver le nœud courant :

```
NodeInstance node = kcontext.getNodeInstance();
```

Le processus (ou sous-processus courant)

```
ProcessInstance p = kcontext.getProcessInstance();  
p.signalEvent( type, eventObject ) ;
```

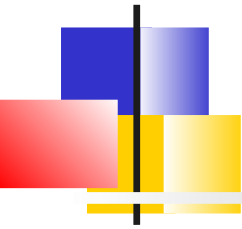
Manipuler les variables

```
Person p = kcontext.getVariable("Personne") ;
```

Accéder à la base de connaissance

```
KnowledgeRuntime kr = kcontext.getKnowledgeRuntime() ;
```





# Modélisation avec jBPM

---

*jPBM* et BPMN2.0  
Nœuds disponibles  
Données

**Persistence et transactions**



# Persistance

---

L'exécution des processus peuvent s'étaler dans le temps. Il est donc nécessaire de les stocker dans une base pour faire face à un redémarrage de serveur par exemple.

Plusieurs types d'informations peuvent être stockées (sérialisées) en base :

- L'état courant du processus (un état en attente)
- L'historique du processus

*jBPM* s'appuie sur **JPA** et **JTA** et permet donc de plugger différentes implémentations de ces APIs. Par défaut, les processus ne sont pas persistants



# Persistance binaire

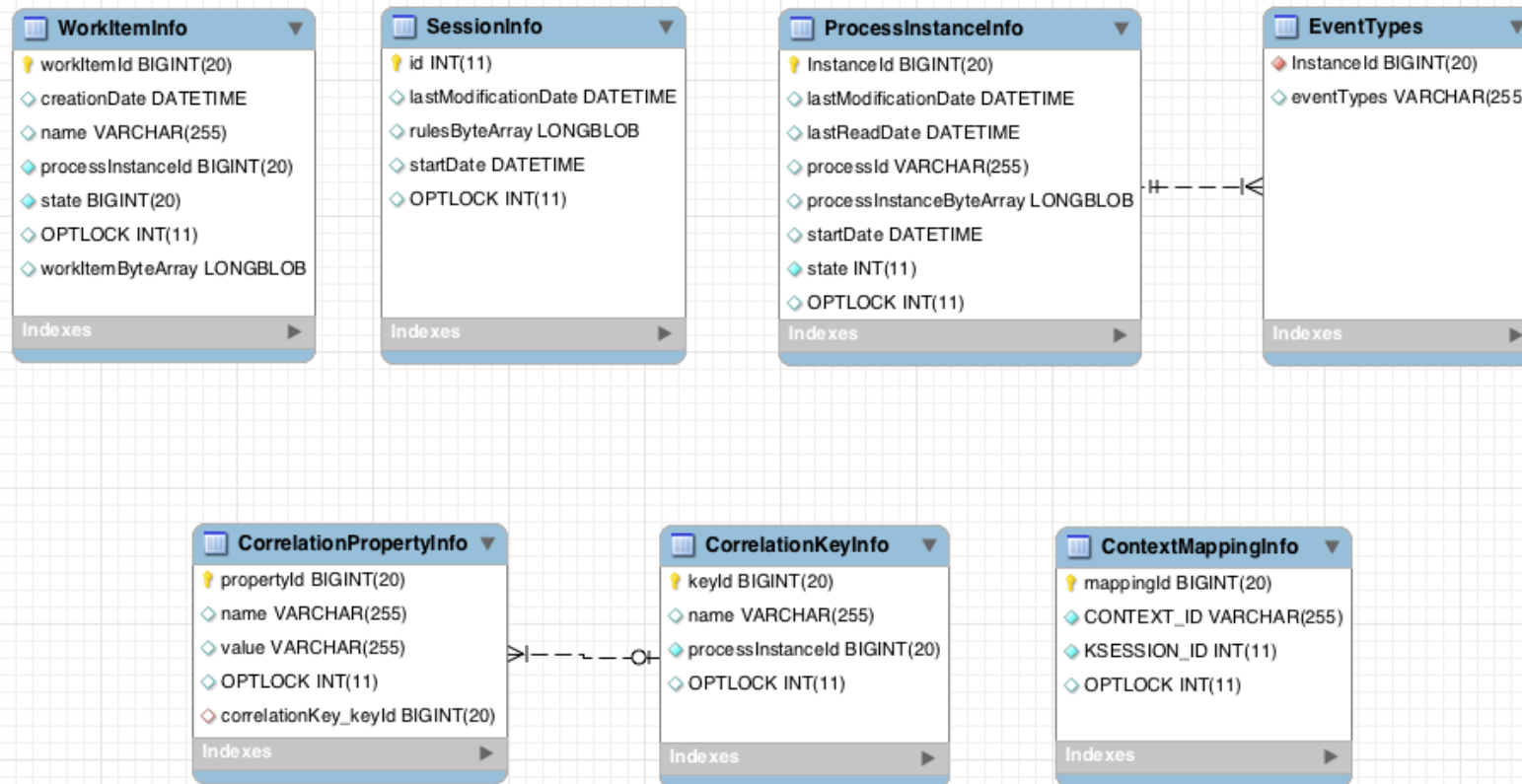
---

jBPM convertit le statut d'un process instance dans un format binaire(marshalling)

Au moment du stockage:

- L'instance est transformée en un *blob*.
- Le *blob* est stocké avec d'autre méta-données : id, définition, date de démarrage
- La session (timers, données de session) est également stockée séparément dans un autre *blob*

# Modèle JBPM





# Point de sauvegarde

---

L'état du processus est stocké en général à la fin d'une séquence d'exécution. (safe point)

Typiquement, un événement externe (signal, fin de tâche utilisateur) provoque le déclenchement d'une séquence de nœuds. Lorsque le processus (et processus fils) ne peuvent plus progresser, leur état est sauvegardé.

La séquence d'exécution correspond généralement à une transaction base de données



# Configuration de la persistance

---

Par défaut, la persistance n'est pas activée.

La configuration de la persistance consiste à :

- Ajouter les dépendances nécessaires (avant jBPM7)
- Configurer une source de données JTA
- Créer le moteur avec la persistance configurée



# Dépendances de l'implémentation par défaut

---

1. `jbpm-persistence-jpa` (`org.jbpm`)
2. `drools-persistence-jpa` (`org.drools`)
3. `persistence-api` (`javax.persistence`)
4. `hibernate-entitymanager` (`org.hibernate`)
5. `hibernate-annotations` (`org.hibernate`)
6. `hibernate-commons-annotations` (`org.hibernate`)
7. `hibernate-core` (`org.hibernate`)
8. `dom4j` (`dom4j`)
9. `jta` (`javax.transaction`)
10. `btm` (`org.codehaus.btm`)
11. `javassist` (`javassist`)
12. `slf4j-api` (`org.slf4j`)
13. `slf4j-jdk14` (`org.slf4j`)
14. `h2` (`com.h2database`)
15. `commons-collections` (`commons-collections`)



# Configuration unité de persistance

---

```
<persistence-unit name="org.jbpm.persistence.jpa" transaction-type="JTA">
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
  <jta-data-source>jdbc/jbpm-ds</jta-data-source>
  <mapping-file>META-INF/JBPMorm.xml</mapping-file>
  <mapping-file>META-INF/Taskorm.xml</mapping-file>
  <mapping-file>META-INF/TaskAuditorm.xml</mapping-file>

  <class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>
  <class>org.drools.persistence.info.SessionInfo</class>
  <class>org.drools.persistence.info.WorkItemInfo</class>

  <class>org.jbpm.process.audit.ProcessInstanceLog</class>
  <class>org.jbpm.process.audit.NodeInstanceLog</class>
  <class>org.jbpm.process.audit.VariableInstanceLog</class>

  <class>org.jbpm.persistence.correlation.CorrelationKeyInfo</class>
  <class>org.jbpm.persistence.correlation.CorrelationPropertyInfo</class>
```





# Transactions

---

Par défaut, chaque invocation du moteur est effectuée dans sa propre transaction

jBPM supporte les transactions JTA (les transactions locales ne sont possible qu'avec Spring)

Il est également possible de délimiter manuellement les frontières de la transaction en utilisant JTA



# Fournisseur JTA

---

Le fournisseur typique de JTA est Wildfly.

La classe *JBPMHelper* utilisée pour les tests fournit du support pour utiliser le gestionnaire de transactions Bitronix et la base mémoire H2



# *JBPMHelper*

---

JBPMHelper permet de démarrer le serveur H2 et de déclarer une datasource *jdbc/jbpm-ds*

```
JBPMHelper.startH2Server();
JBPMHelper.setupDataSource();
RuntimeEnvironment env = RuntimeEnvironmentBuilder.Factory.get()
    .newDefaultBuilder()
    .addAsset(ResourceFactory.newClassPathResource("myProcess.bpmn"), ResourceType.BPMN2)
    .get();
```



# Configuration de prod.

---

Sans l'utilisation de *JBPMHelper* et de h2, il faut :

- modifier le descripteur *persistence.xml* pour l'adapter à la base utilisée
- fournir le nom de *EntityManagerFactory* lors de la création du Runtime

```
RuntimeEnvironment environment =  
    RuntimeEnvironmentBuilder.Factory.get().newDefaultBuilder()  
        .entityManagerFactory(Persistence.createEntityManagerFactory("tp5db"))  
        .addAsset(ResourceFactory.newClassPathResource("tp7/subprocess.bpmn"),  
            ResourceType.BPMN2)  
        .get();
```



# Configuration manuelle du moteur

---

*JPAKnowledgeService* peut être utilisé pour créer des KIE session

```
// create the entity manager factory and register it in the environment
```

```
EntityManagerFactory emf =
```

```
    Persistence.createEntityManagerFactory( "org.jbpm.persistence.jpa" );
```

```
Environment env = KnowledgeBaseFactory.newEnvironment();
```

```
env.set( EnvironmentName.ENTITY_MANAGER_FACTORY, emf );
```

```
// create a new KIE session that uses JPA to store the runtime state
```

```
StatefulKnowledgeSession ksession =
```

```
    JPAKnowledgeService.newStatefulKnowledgeSession( kbase, null, env );
```

```
int sessionId = ksession.getId();
```

```
// invoke methods on your method here
```

```
ksession.startProcess( "MyProcess" );
```

```
ksession.dispose();
```



# Configuration avec *RuntimeEnvironmentBuilder*

---

```
EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("org.jbpm.persistence.jpa");
```

```
RuntimeEnvironment runtimeEnv =  
    RuntimeEnvironmentBuilder.Factory  
        .get()  
        .newDefaultBuilder()  
        .entityManagerFactory(emf)  
        .knowledgeBase(kbase)  
        .get();
```



# Récupérer une session persistée

---

```
ksession =  
    JPAKnowledgeService.loadStatefulKnowledgeSession(sessionId, kbase, null, env );
```



# Exemple délimitation manuelle de la transaction (*JBPM6*)

---

```
// create the entity manager factory and register it in the environment
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory( "org.jbpm.persistence.jpa" );
Environment env = KnowledgeBaseFactory.newEnvironment();
env.set( EnvironmentName.ENTITY_MANAGER_FACTORY, emf );
env.set( EnvironmentName.TRANSACTION_MANAGER,
    TransactionManagerServices.getTransactionManager() );
// create a new knowledge session that uses JPA to store the runtime state
StatefulKnowledgeSession ks = JPAKnowledgeService.newStatefulKnowledgeSession( kbase,
    null, env );
// start the transaction
UserTransaction ut = (UserTransaction) new InitialContext().lookup(
    "java:comp/UserTransaction" );
ut.begin();
// perform multiple commands inside one transaction
ks.insert( new Person( "John Doe" ) );
ks.startProcess( "MyProcess" );
// commit the transaction
ut.commit();
```





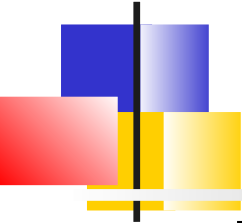
# Historique

---

Si nécessaire, il est possible de stocker l'historique d'une exécution dans une base de données (éviter d'utiliser la base applicative).

Les traces d'exécution sont générées à partir des événements processus.

Il est alors facile d'implémenter des *listeners* stockant les informations nécessaires dans la base



# *jbpm-audit*

---

Le module ***jbpm-audit*** contient un listener d'évènement qui stocke les informations relatives au process dans une BD via JPA.

Le modèle de données contient 3 entités :

- Un pour les informations du process
- Un pour les information du nœud
- Un pour les informations sur les variables.

# Modèle BAM

ProcessInstanceLog
id BIGINT(20)
duration BIGINT(20)
end_date DATETIME
externalId VARCHAR(255)
user_identity VARCHAR(255)
outcome VARCHAR(255)
parentProcessInstanceId BIGINT(20)
processId VARCHAR(255)
processInstanceId BIGINT(20)
processName VARCHAR(255)
processVersion VARCHAR(255)
start_date DATETIME
status INT(11)
Indexes

NodeInstanceLog
id BIGINT(20)
connection VARCHAR(255)
log_date DATETIME
externalId VARCHAR(255)
nodeId VARCHAR(255)
nodeInstanceId VARCHAR(255)
nodeName VARCHAR(255)
nodeType VARCHAR(255)
processId VARCHAR(255)
processInstanceId BIGINT(20)
type INT(11)
workItemId BIGINT(20)
Indexes

VariableInstanceLog
id BIGINT(20)
log_date DATETIME
externalId VARCHAR(255)
oldValue VARCHAR(255)
processId VARCHAR(255)
processInstanceId BIGINT(20)
value VARCHAR(255)
variableId VARCHAR(255)
variableInstanceId VARCHAR(255)
Indexes

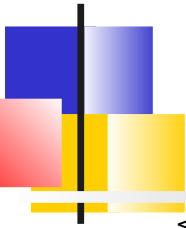


# Stocker les évènements dans la base

---

Pour stocker l'historique, il faut configurer un logger spécifique:

```
KieSession ksession = ...;  
AbstractAuditLogger auditLogger =  
    AuditLoggerFactory.newInstance(Type.JPA, ksession,  
    null);  
ksession.addProcessEventListener(auditLogger);
```



# Configuration *persistence.xml*

---

```
<persistence-unit name="org.jbpm.persistence.jpa" transaction-type="JTA">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <jta-data-source>jdbc/jbpm-ds</jta-data-source>
  <mapping-file>META-INF/JBPMorm.xml</mapping-file>
  <class>org.drools.persistence.info.SessionInfo</class>
  <class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>
  <class>org.drools.persistence.info.WorkItemInfo</class>
  <class>org.jbpm.persistence.correlation.CorrelationKeyInfo</class>
  <class>org.jbpm.persistence.correlation.CorrelationPropertyInfo</class>
  <class>org.jbpm.runtime.manager.impl.jpa.ContextMappingInfo</class>

  <class>org.jbpm.process.audit.ProcessInstanceLog</class>
  <class>org.jbpm.process.audit.NodeInstanceLog</class>
  <class>org.jbpm.process.audit.VariableInstanceLog</class>

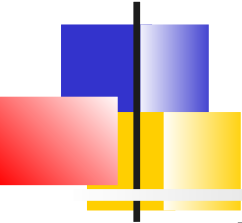
  <properties>
    <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
    <property name="hibernate.max_fetch_depth" value="3"/>
    <property name="hibernate.hbm2ddl.auto" value="update"/>
    <property name="hibernate.show_sql" value="true"/>
    <property name="hibernate.connection.release_mode" value="after_transaction"/>
    <property name="hibernate.transaction.jta.platform" value="org.hibernate.service.jta.platform.internal.JBossStandAloneJtaPlatform"/>
  </properties>
</persistence-unit>
```



# Exemple listener spécifique

---

```
public class CustomLogFactory extends SimpleRegisterableItemsFactory {
    historyEnabled = false;
    .
    @Override
    public List<ProcessEventListener> getProcessEventListeners(RuntimeEngine runtime) {
        List<ProcessEventListener> defaultListeners = new ArrayList<ProcessEventListener>();
        // register JPAWorkingMemoryDBLogger
        if(historyEnabled){
            AbstractAuditLogger logger =
                AuditLoggerFactory.newJPAInstance(runtime.getKieSession().getEnvironment());
            logger.setBuilder(getAuditBuilder(runtime));
            defaultListeners.add(logger);
        }
        // add any custom listeners
        defaultListeners.addAll(super.getProcessEventListeners(runtime));
        return defaultListeners;
    }
    .
    .
}
```



# Exemple listener spécifique (2)

---

## Enregistrement classe spécifique dans l'environnement

```
CustomLogFactory logFactory = new CustomLogFactory();  
environment = RuntimeEnvironmentBuilder.Factory.get()  
    .newDefaultBuilder()  
    .entityManagerFactory(Persistence.createEntityManagerFactory(Benchmark.persistanceUnit))  
    .registerableItemsFactory(logFactory)  
    .addAsset(ResourceFactory.newClassPathResource("process01.bpmn20.xml"), ResourceType.BPMN2)  
    .get();
```



# Services externes

---

**Workflow humain**  
Autres services externes





# Workflow humain

---

Certaines tâches dans les processus métier sont exécutées par des « humains », les acteurs du processus

*jBPM* fournit le nœud *task* représentant l'interaction avec un acteur du processus

Durant l'exécution du nœud, le moteur de workflow interagit avec un service externe : le ***TaskHandler***

*jBPM* fournit une implémentation optionnelle d'un *TaskHandler* basé sur la spécification WS-HT (*Web services Human Task*)



# Utilisation

---

Pour implémenter un workflow humain avec jBPM :

- (1) Inclure des nœuds tâches dans le processus
- (2) Intégrer un composant de gestion de tâche ou utiliser celui de jBPM
- (3) Construire une UI permettant aux utilisateurs finaux d'interagir avec le moteur : Consulter leur liste de tâches, poster des formulaire des complétion d'une tâche, etc.



# Nœud tâche

---

Un nœud tâche représente une tâche atomique effectuée par un utilisateur.

*jBPM* considère ce type de nœud comme un nœud service devant être invoqué.

Le service prévient le moteur lors de la fin de la tâche humaine

Les informations additionnelles sur un nœud tâche incluent principalement :  
les utilisateurs pouvant réaliser la tâche



# Propriétés

---

Les propriétés additionnelles de ce type de nœud sont :

**TaskName** : Le nom de la tâche

**Priority** : Entier indiquant la priorité de la tâche

**Comment** : Un commentaire.

**ActorId** : Le(s) responsable(s) de la tâche

**groupId** : Le(s) groupe(s) utilisateur responsable(s) de la tâche

**Skippable** : Indique si l'utilisateur peut ignorer la tâche

**Content** : Les données associés à la tâche

**Swimlane** : Le couloir d'activité.

**Wait for completion** : Le nœud doit attendre la réalisation de la tâche avant de continuer

**On-entry** et **on-exit** : Actions à exécuter à l'entrée et la sortie du nœud

**Parameter/result mapping** : Échange de variables avec la tâche

**ParentId** : La tâche parente.

Les valeurs des propriétés peuvent être des constantes ou des expressions MVEL



# Couloir d'activité

---

- ❖ Les **couloirs d'activité** (*swimlane*) permettent que différentes tâches soient réalisées par le **même** acteur.
  - Lorsqu'une première tâche est créée pour un couloir d'activité donné. L'acteur de la tâche est stocké et il devra réaliser toutes les autres tâches du même couloir d'activité.
  - L'affectation est définie dans le processus soit avec un identifiant d'acteur ou un identifiant de groupe d'acteurs

Les couloirs d'activité doivent être déclarés au niveau du processus



# *WorkItemHandler*

---

Pour gérer l'exécution de tout service externe (finalisation, annulation), un ***WorkItemHandler*** doit être enregistré dans la configuration.

*JBPM* fournit une implémentation basée sur la spécification WS-HumanTask.

*[http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/WS-HumanTask\\_v1.pdf](http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/WS-HumanTask_v1.pdf)*

Cette implémentation peut s'avérer trop complexe pour le besoin du projet et il est très aisé d'implémenter son propre *WorkItemHandler*



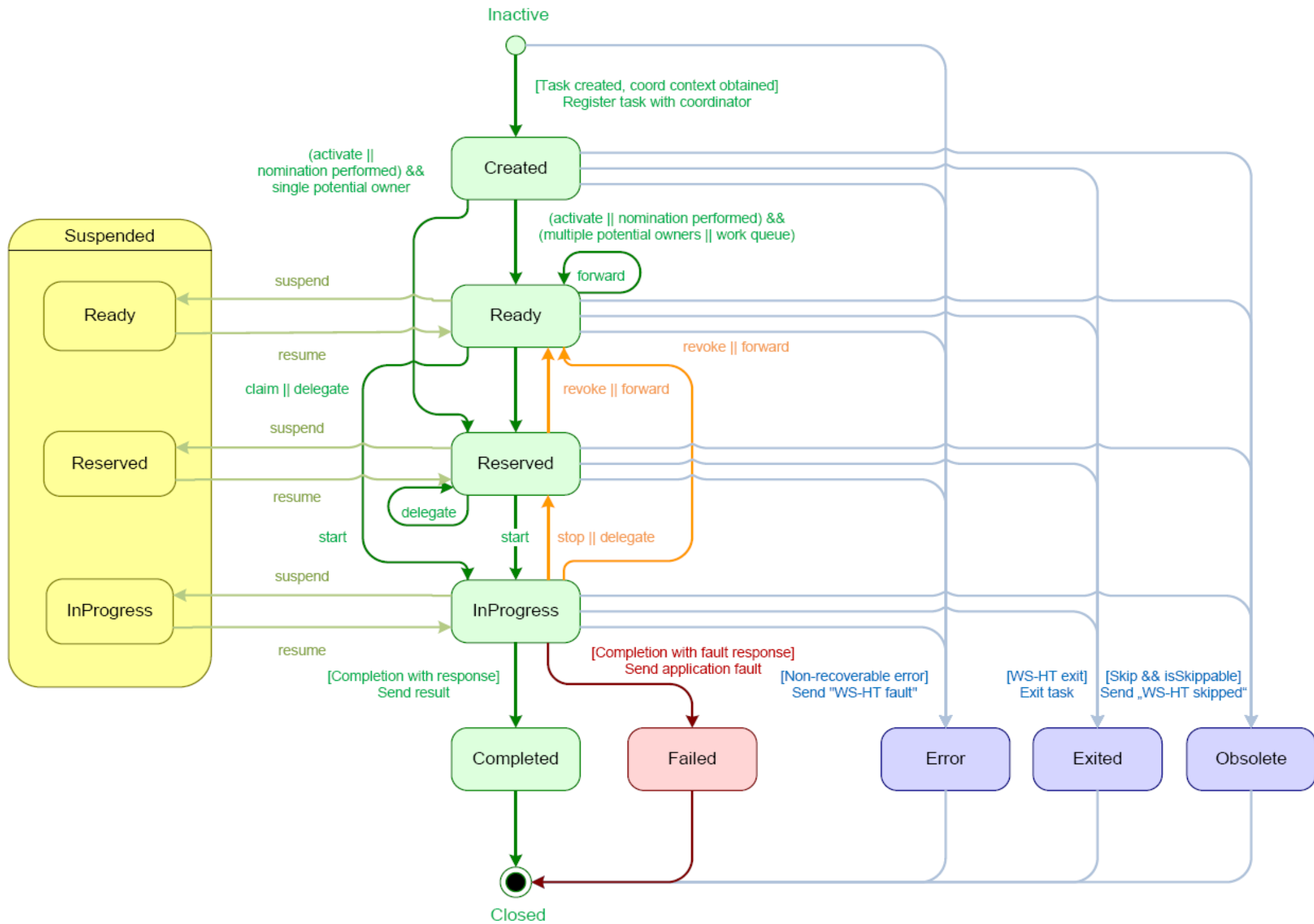
# WS-HT, séquence *normale*

---

Généralement, le process attend la réalisation de la tâche avant de progresser.

Avec WS-HT, le cycle le plus direct est :

- Création de la tâche  
=> La tâche apparaît dans la liste de tâches des acteurs autorisés. Elle est dans le statut *created*
- Un acteur réclame la tâche, la tâche est dans le statut *reserved*
- L'acteur démarre la tâche, la tâche est dans le statut *started*
- Finalement, l'acteur termine la tâche







# RuntimeManager

---

Depuis jBM6, le *RuntimeManager* facilite la mise en place du service de tâche par défaut de jBPM

- En utilisant cette approche, il n'est plus nécessaire d'enregistrer le service de tâche auprès du moteur
- Pour plugger une autre gestionnaire de tâche, il faut configurer la session via le *kmodule.xml* ou programmatiquement



# Interaction avec service de tâche (*jBPM6+*)

---

```
RuntimeEngine engine =  
    runtimeManager.getRuntimeEngine(EmptyContext.get());  
KieSession kieSession = engine.getKieSession();  
// Start a process  
kieSession.startProcess("CustomersRelationship.customers", params);  
// Do Task Operations  
TaskService taskService = engine.getTaskService();  
List<TaskSummary> tasksAssignedAsPotentialOwner =  
    taskService.getTasksAssignedAsPotentialOwner("mary", "en-UK");  
  
// Claim Task  
taskService.claim(taskSummary.getId(), "mary");  
// Start Task  
taskService.start(taskSummary.getId(), "mary");
```



# Services externes

---

Workflow humain  
**Autres services externes**



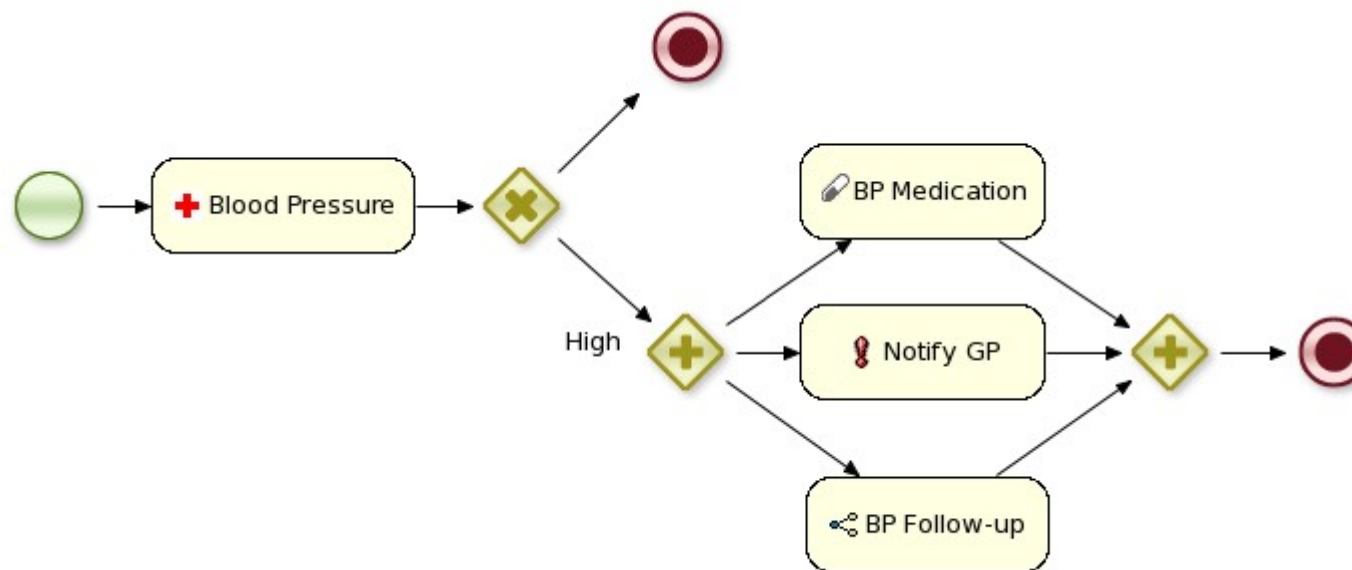
# Introduction

---

*jBPM* permet de définir des nœuds services spécifiques à un domaine d'utilisation

Ces nœuds peuvent être utilisés dans l'éditeur graphique permettant d'avoir une représentation claire du processus métier tout en cachant les détails d'implémentation

# Example





# Mise en place

---

Pour exécuter des processus utilisant des nœuds services spécifique, il faut :

- Créer les fichiers de définitions du service (optionnel)
- Enregistrer les définitions (optionnel)
- Utiliser les nouveaux nœuds dans le processus ou utiliser le nœud générique service
- Fournir une implémentation du service



# Fichiers de définition

---

Les fichiers de définitions précisent :

- ✓ Les paramètres d'entrée du service (Map)
- ✓ Les résultats (Map)
- ✓ Le nom du nœud
- ✓ Éventuellement un icône

Les fichiers doivent être placés dans le *classpath* sous le répertoire *META-INF*



# Exemple

---

```
import org.drools.process.core.datatype.impl.type.StringDataType;
[
// the Notification work item
[
"name" : "Notification",
"parameters" : [
"Message" : new StringDataType(),
"From" : new StringDataType(),
"To" : new StringDataType(),
"Priority" : new StringDataType(),
],
"displayName" : "Notification",
"icon" : "icons/notification.gif"
]
]
```





# Enregistrement des définitions

---

Pour être prise en compte, les définitions de service doivent être référencées dans un fichier ***drools.rulebase.conf*** présent également dans *META-INF*

#WorkDefinitions.conf est le fichier par défaut

#définissant les services Email et Log

```
drools.workDefinitions = MyWorkDefinitions.conf WorkDefinitions.conf
```



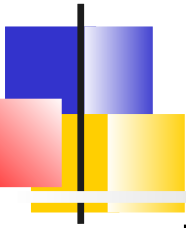
# Utilisation des nœuds

---

L'éditeur de processus fait apparaître les nouveaux nœuds dans la section  
« *Service Tasks* »

La vue propriété permet d'accéder aux propriétés définies par le service plus :

- ✓ Le mapping de paramètres et résultat



# Implémentation du service

---

Le service doit implémenter l'interface *WorkItemHandler* qui définit 2 méthodes :

```
void executeWorkItem(WorkItem workItem,  
    WorkItemManager manager) :
```

Exécute le service et notifie le manager de la fin du service

```
void abortWorkItem(WorkItem workItem,  
    WorkItemManager manager) :
```

Interrompt le service et notifie le manager de l'interruption du service



# Example

---

```
public class NotificationWorkItemHandler implements WorkItemHandler {
    public void executeWorkItem(WorkItem workItem, WorkItemManager manager) {
        String from = (String) workItem.getParameter("From");
        String to = (String) workItem.getParameter("To");
        String message = (String) workItem.getParameter("Message");
        String priority = (String) workItem.getParameter("Priority");
        // send email
        EmailService service = ServiceRegistry.getInstance().getEmailService();
        service.sendEmail(from, to, "Notification", message);
        // notify manager that work item has been completed
        manager.completeWorkItem(workItem.getId(), null);
    }
    public void abortWorkItem(WorkItem workItem, WorkItemManager manager) {
        // Do nothing, notifications cannot be aborted
    }
}
```



# Enregistrement de l'implémentation

---

Lors de l'exécution, l'implémentation du service doit être enregistrée auprès du *WorkItemManager*

```
ksession.getWorkItemManager().registerWorkItemHandler("Notification", new NotificationWorkItemHandler());
```



# Handlers fournis

---

jBPM propose de nombreuses implémentations de *WorkItemHandler* :

Par exemple, dans le package  
`org.jbpm.bpmn2.handler` :

- *ReceiveTaskHandler* (<receiveTask>)
- *SendTaskHandler* (<sendTask>)
- *ServiceTaskHandler* (<serviceTask>)

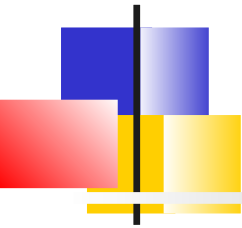
Dans le module *jbpm-workitems*, il y a des exemples d'implémentations (FTP, REST, Service Web, JMS, ...)



# Merci!!!

---

❖ MERCI DE VOTRE ATTENTION



## Annexe : persistance *jBPM6*





# *JPAKnowledgeService*

---

Ensuite, il faut créer (ou recréer) les sessions avec le service ***JPAKnowledgeService*** en ayant positionné *l'EntityManagerFactory* dans l'environnement



# Example

---

```
// create the entity manager factory and register it in the environment
EntityManagerFactory emf =
Persistence.createEntityManagerFactory( "org.jbpm.persistence.jpa" );
Environment env = KnowledgeBaseFactory.newEnvironment();
env.set( EnvironmentName.ENTITY_MANAGER_FACTORY, emf ) ;

// create a new knowledge session that uses JPA to store the runtime state
StatefulKnowledgeSession ksession =
JPAKnowledgeService.newStatefulKnowledgeSession( kbase, null, env );
int sessionId = ksession.getId() ;

// invoke methods on your method here
ksession.startProcess( "MyProcess" );
ksession.dispose();
```



# Chargement de la session

---

L'identifiant de la session est nécessaire pour pouvoir la recharger à partir de la base de données et de poursuivre l'exécution des processus.

```
// create the entity manager factory and register it in the environment
```

```
EntityManagerFactory emf =
```

```
Persistence.createEntityManagerFactory( "org.jbpm.persistence.jpa" );
```

```
Environment env = KnowledgeBaseFactory.newEnvironment();
```

```
env.set( EnvironmentName.ENTITY_MANAGER_FACTORY, emf );
```

```
ksession = JPAKnowledgeService.loadStatefulKnowledgeSession(  
    sessionId, kbase, null, env );
```



# *persistence.xml*

---

L'unité de persistance est définie de façon standard dans un fichier ***persistence.xml*** présent dans le classpath

```
<persistence-unit name="org.jbpm.persistence.jpa">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <jta-data-source>jdbc/processInstanceDS</jta-data-source>
  <class>org.drools.persistence.info.SessionInfo</class>
  <class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</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>
```



# Datasource JBoss

---

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

<datasources>

<local-tx-datasource>

<jndi-name>jdbc/processInstanceDS</jndi-name>

<connection-url>jdbc:h2:file:/NotBackedUp/data/process-instance-db</
  connection-url>

<driver-class>org.h2.jdbcx.JdbcDataSource</driver-class>

<user-name>sa</user-name>

<password>sasa</password>

</local-tx-datasource>

</datasources>
```



# jBPM6

---

```
RuntimeEnvironment env =  
    RuntimeEnvironmentBuilder.Factory.get().newDefaultBuilder().  
    entityManagerFactory(Persistence.createEntityManagerFactory("tp5db"))  
    .addAsset(ResourceFactory.newClassPathResource("tp5/subprocess.bpmn"),  
        ResourceType.BPMN2)  
    .addAsset(ResourceFactory.newClassPathResource("tp5/Main.bpmn"),  
        ResourceType.BPMN2).get();  
  
return RuntimeManagerFactory.Factory.get().newSingletonRuntimeManager(env);
```