Architecture

This document describes the basics of the jBPM architecture. This represents the technical strategy on how we jBPM aims to serve the use cases defined in seven.forms.of.bpm.

Table of Contents

Basics

A process definition is an executable graph based on nodes and transitions. It's a static description which can have many executions.

An execution is a path of execution. It holds the runtime state of one execution, including a pointer to the position in the process graph.

An execution that is executing, will interprete the process definition diagram.

To cope with concurrent paths of execution, an execution can have children. The root of the tree is also called the process instance.

Execution modes

There are basically three process execution modes: object, persistent and embedded. For the persistent and embedded execution modes, the process execution has to participate in a transaction. In that case, the process execution has to take place inside of an Environment. The environment will be used to bind process execution updates to a transaction in the application transaction. The environment can be used to bind to e.g. a JDBC connection, JTA, BMT, Spring transactions and so on.

Object execution mode

Object execution mode is the simplest form of working with the Process Virtual Machine. This means working with the process definition and execution objects directly through the client API. Let's show this by an example. We start by creating a

`ClientProcessDefinition` that looks like this:

Object execution mode is the simplest form of working with the Process Virtual Machine. This means working with the process definition and execution objects directly through the client API. Let's show this by an example. We start by creating a `ClientProcessDefinition` that looks like this:

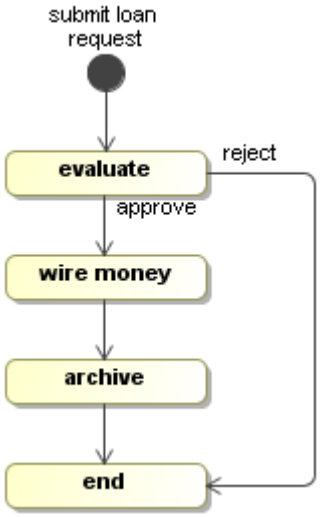


Figure: The loan process

```
ClientProcessDefinition processDefinition = ProcessFactory.build("loan")
  .node("submit loan request").initial().behaviour(AutomaticActivity.class)
    .transition().to("evaluate")
  .node("evaluate").behaviour(WaitState.class)
    .transition("approve").to("wire money")
    .transition("reject").to("end")
  .node("wire money").behaviour(AutomaticActivity.class)
    .transition().to("archive")
  .node("archive").behaviour(WaitState.class)
    .transition().to("end")
  .node("end").behaviour(WaitState.class)
.done();
```

The `ProcessFactory` is a helper class that provides convenience for building an object graph that represents a process definition. `AutomaticActivity` is a pass-through activity without anything happening and `WaitState` will wait until an external signal is given. Both activity implementations will be covered in more depth later.

The `processDefinition` object serves as a factory for process instance objects. A process instance represents one execution of the process definition. More precise, the process instance is the main path of execution.

```
ClientExecution execution = processDefinition.startProcessInstance();
```

A process instance itself is also an `Execution`. Potentially, an execution can have child executions to represent concurrent paths of execution.

The `execution` can be seen as a state machine that operates as described in the process definition. Starting a process instance means that the initial node of the process definition is executed. Since this is an automatic activity, the execution will

proceed to the `evaluate` node. The `evaluate` node is a wait state. When the execution arrived at the evaluate node, the method `startProcessInstance` will return and waits until an external signal is provided with the `signal` method. So after the `startProcessInstance` we can verify if the execution is positioned in the evaluate node.

```
assertEquals("evaluate", execution.getNodeName());
```

To make the process execute further, we provide an external trigger with the `signal` method. The result of the evaluation will be given as the signalName parameter like this:

```
execution.signal("approve");
```

The `WaitState` activity implementation will take the transition that corresponds to the given signalName. So the execution will first execute the automatic activity `wire money` and then return after entering the next wait state `archive`.

```
assertEquals("archive", execution.getNodeName());
```

When the execution is waiting in the archive node, the default signal will make it take the first unnamed transition.

```
execution.signal();
assertEquals("end", execution.getNodeName());
```

The process has executed in the thread of the client. The `startProcessInstance` method only returned when the `evaluate` node was reached. In other words, the `ClientProcessDefinition.startProcessInstance` and `ClientExecution.signal` methods are blocking until the next wait state is reached.

Persistent execution mode

The Process Virtual Machine also contains the hibernate mappings to store the process definitions and executions in any database. A special session facade called `ExecutionService` is provided for working with process executions in such a persistent environment.

Two configuration files should be available on the classpath: an environment configuration file and a `hibernate.properties` file. A basic configuration for persistent execution mode in a standard Java environment looks like this:

environment.cfg.xml:

```
<contexts xmlns="http://jbpm.org/pvm/1.0/wire">

  <environment-factory>

    <deployer-manager>
      <language name="api">
        <check-version />
        <create-id />
        <save-process />
      </language>
    </deployer-manager>

    <process-service />
```

```
    <execution-service />
    <management-service />

    <command-service>
      <retry-interceptor />
      <environment-interceptor />
      <standard-transaction-interceptor />
    </command-service>

    <hibernate-configuration>
      <properties resource="hibernate.properties" />
      <mappings resource="org/jbpm/pvm/pvm.hibernate.mappings.xml" />
      <cache-configuration resource="org/jbpm/pvm/pvm.cache.xml"
                           usage="nonstrict-read-write" />
    </hibernate-configuration>

    <hibernate-session-factory />

    <id-generator />
    <variable-types resource="org/jbpm/pvm/pvm.types.xml" />
    <job-executor auto-start="false" />

  </environment-factory>

  <environment>
    <hibernate-session />
    <transaction />
    <pvm-db-session />
    <job-db-session />
    <message-session />
  </environment>

</contexts>
```

And next to it a hibernate.properties like this

```
hibernate.properties:
hibernate.dialect                       org.hibernate.dialect.HSQLDialect
hibernate.connection.driver_class       org.hsqldb.jdbcDriver
hibernate.connection.url                jdbc:hsqldb:mem:.
hibernate.connection.username           sa
hibernate.connection.password
hibernate.hbm2ddl.auto                  create-drop
hibernate.cache.use_second_level_cache true
hibernate.cache.provider_class
org.hibernate.cache.HashtableCacheProvider
# hibernate.show_sql                      true
hibernate.format_sql                    true
hibernate.use_sql_comments              true
```

Then you can obtain the services from the environment factory like this:

```
EnvironmentFactory environmentFactory = new
PvmEnvironmentFactory("environment.cfg.xml");

ProcessService processService = environmentFactory.get(ProcessService.class);
ExecutionService executionService =
environmentFactory.get(ExecutionService.class);
```

```
ManagementService managementService =
environmentFactory.get(ManagementService.class);
```

The responsibility of the `ProcessService` is to manage the repository of process definitions. Before we can start a process execution, the process definition needs to be deployed into the process repository. Process definitions can be supplied in various formats and process definition languages. A deployment collects process definition information from various sources like a ZIP file, an XML file or a process definition object. The method `ProcessService.deploy` will take a deployment through all the deployers that are configured in the configuration file.

In this example, we'll supply a process definition programmatically for deployment.

```
ClientProcessDefinition processDefinition = ProcessFactory.build("loan")
  .node("submit loan request").initial().behaviour(AutomaticActivity.class)
    .transition().to("evaluate")
  .node("evaluate").behaviour(WaitState.class)
    .transition("approve").to("wire money")
    .transition("reject").to("end")
  .node("wire money").behaviour(AutomaticActivity.class)
    .transition().to("archive")
  .node("archive").behaviour(WaitState.class)
    .transition().to("end")
  .node("end").behaviour(WaitState.class)
.done();

Deployment deployment = new Deployment(processDefinition);
processService.deploy(deployment);
```

Now, a version of that process definition is stored in the database. The `check-version` deployer will have assigned version 1 to the stored process definition. The `create-id` deployer will have distilled id `loan:1` from the process name and the assigned version.

Deploying that process again will lead to a new process definition version being created in the database. But an incremented version number will be assigned. For the purpose of versioning, process definitions are considered equal if they have the same name.

It is recommended that a user provided key reference is supplied for all process executions. Starting a new process execution goes like this:

```
Execution execution = executionService.startExecution("loan:1",
"request7836");
```

The return value is an execution interface, which prevents navigation of relations. That is because outside of the service methods, the transaction and hibernate session is not guaranteed to still be open. In fact, the default configuration as given above will only keep the transaction and session open for the duration of the service method. So navigating the relations outside of the service methods might result into a hibernate `LazyInitializationException`. But the current node name can still be verified:

```
assertEquals("evaluate", execution.getNodeName());
```

Also very important is the generated id that can be obtained. The default `id-generator` will use the process definition id and the given key to make a unique id for the process execution like this:

```
assertEquals("loan:1/request7836", execution.getId());
```

That id must be when providing the subsequent external triggers to the process execution like this:

```
executionService.signalExecution("loan:1/request7836", "approve");
```

More information about service interfaces to run in persistent mode can be found in package `org.jbpm.pvm` of the api docs.

Embedded execution mode

Embedded execution mode means that the state of a process is stored as a string column inside a user domain object like e.g. a loan.

```
public class Loan {

  /** the loan process definition as a static resource */
  private static final ClientProcessDefinition processDefinition =
createLoanProcess();

  private static ClientProcessDefinition createLoanProcess() {
    ClientProcessDefinition processDefinition = ProcessFactory.build("loan")
      .node("submit loan
request").initial().behaviour(AutomaticActivity.class)
        .transition().to("evaluate")
      .node("evaluate").behaviour(WaitState.class)
        .transition("approve").to("wire money")
        .transition("reject").to("end")
      .node("wire money").behaviour(AutomaticActivity.class)
        .transition().to("archive")
      .node("archive").behaviour(WaitState.class)
        .transition().to("end")
      .node("end").behaviour(WaitState.class)
    .done();

    return processDefinition;
  }

  /** exposes the process definition to the execution hibernate type */
  private static ClientProcessDefinition getProcessDefinition() {
    return processDefinition;
  }


  long dbid;
  String customer;
  double amount;
  ClientExecution execution;

  /** constructor for persistence */
  protected Loan() {
  }
```

```
  public Loan(String customer, double amount) {
    this.customer = customer;
    this.amount = amount;
    this.execution = processDefinition.startProcessInstance();
  }

  public void approve() {
    execution.signal("approve");
  }

  public void reject() {
    execution.signal("reject");
  }

  public void archiveComplete() {
    execution.signal();
  }

  public String getState() {
    return execution.getNodeName();
  }

  ...getters...
}
```

If you ignore the bold parts for a second, you can see that this is a POJO without anything fancy. It's just a bean that can be stored with hibernate. The bold part indicate that implementation part of the class that is related to process and execution. Not that nothing of the process definition or execution is exposed to the user of the Loan class.

Each `Loan` object corresponds to a `loan` process instance. Some methods of the `Loan` class correspond to the external triggers that need to be given during the lifecycle of a `Loan` object.

Next we'll show how to use this class. To get started we need a `hibernate.cfg.xml`:

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

<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>

    <property
name="hibernate.dialect">org.hibernate.dialect.HSQLDialect</property>
    <property
name="hibernate.connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="hibernate.connection.url">jdbc:hsqldb:mem:.</property>
    <property name="hibernate.connection.username">sa</property>
    <property name="hibernate.connection.password"></property>
    <property name="hibernate.hbm2ddl.auto">create</property>
    <property name="hibernate.show_sql">true"</property>
    <property name="hibernate.format_sql">true"</property>
    <property name="hibernate.use_sql_comments">true"</property>
```

```
    <mapping resource="Loan.hbm.xml"/>

  </session-factory>
</hibernate-configuration>
```

And a

`Loan.hbm.xml`:

```
<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC
          "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.jbpm.pvm.api.db.embedded" default-
access="field">

  <typedef name="execution"
class="org.jbpm.pvm.internal.hibernate.ExecutionType" />

  <class name="Loan" table="LOAN">

    <id name="dbid">
      <generator class="sequence"/>
    </id>

    <property name="execution" type="execution" />
    <property name="customer" />
    <property name="amount" />

  </class>

</hibernate-mapping>
```

Then you can use the Loan class like this in a test

```
Configuration configuration = new Configuration();
configuration.configure();
SessionFactory sessionFactory = configuration.buildSessionFactory();

// start a session/transaction
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

Loan loan = new Loan("john doe", 234.0);
session.save(loan);
assertEquals("evaluate", loan.getState());

// start a new session/transaction
transaction.commit();
session.close();
session = sessionFactory.openSession();
transaction = session.beginTransaction();

loan = (Loan) session.get(Loan.class, loan.getDbid());
assertEquals("evaluate", loan.getState());
loan.approve();
```

```
assertEquals("archive", loan.getState());

// start a new session/transaction
transaction.commit();
session.close();
```

After executing this code snippet, this is the loan record in the DB:

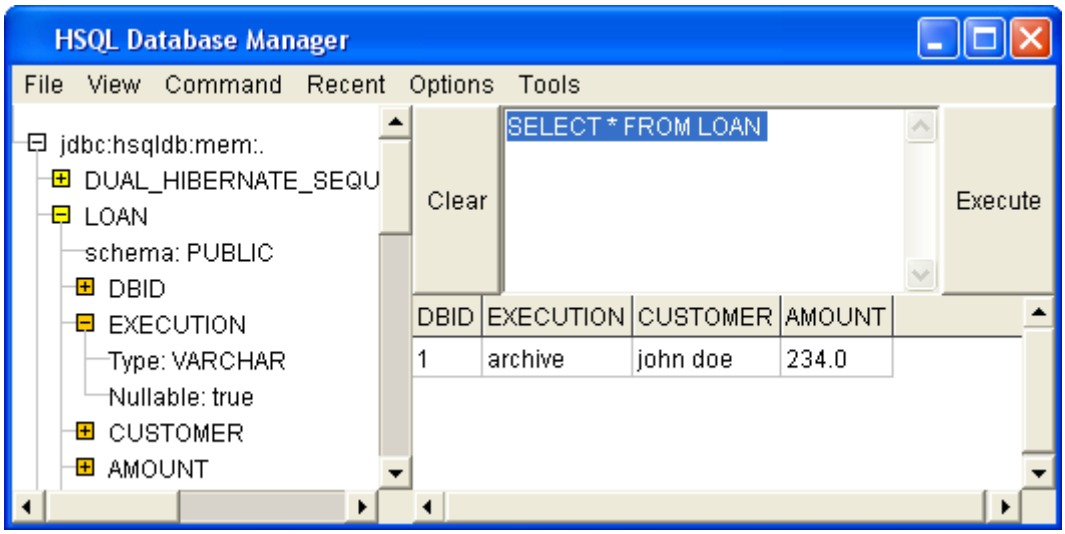


Figure: The loan record in the DB

APIs

The Process Virtual Machine has 4 integrated API's that together offer a complete coverage of working with processes in the different execution modes. Each of the APIs has a specific purpose that fits within the following overall architecture.

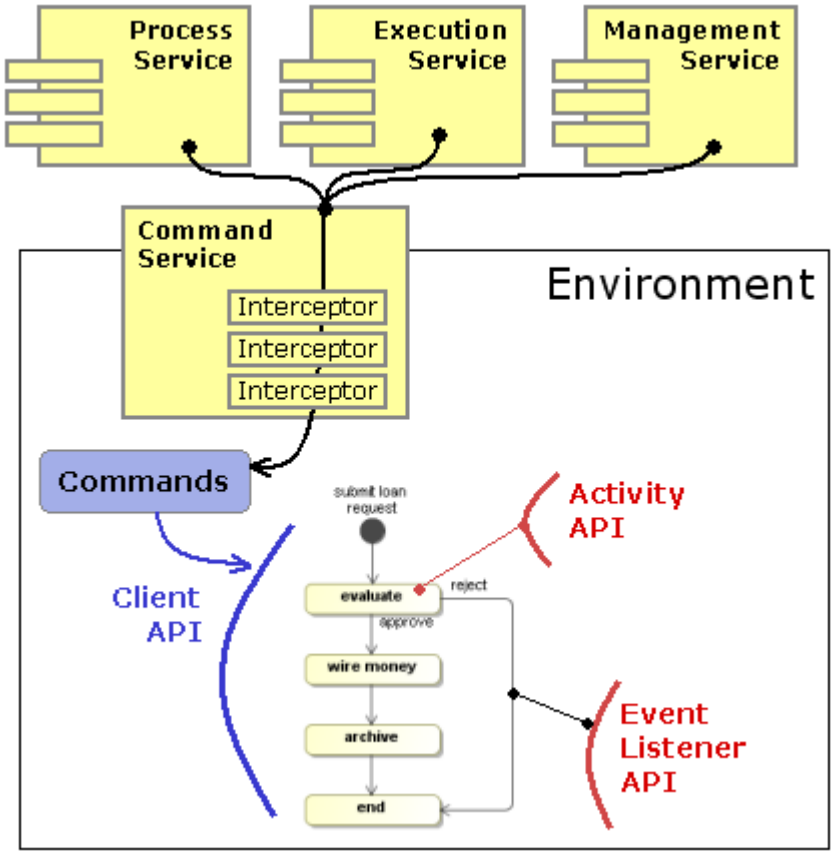


Figure: The 4 API's of the Process Virtual Machine

The services interfaces should be used from application code that wants to interact with the Process Virtual Machine which runs in transactional persistent mode, backed by a database. This is the most typical way how users interact with the PVM as a workflow engine.

To execute processes without persistence, the client API can be used to work with process and execution objects directly. The client API expose the methods of the core model objects.

The activity API is used to implement the runtime behaviour of activities. So a activity type is in fact a component with at the core an implementation of the `Activity` interface. Activity implementations can control the flow of execution.

The event listener API serves to write pieces of Java code that should be executed upon process events. It's very similar to the activity API with that exception that event listeners are not able to control the flow of execution.

Activity API

The activity API allows to implement the runtime activity behaviour in Java.

```
public interface Activity extends Serializable {
  void execute(ActivityExecution execution) throws Exception;
}
```

An activity is the behaviour of the node to which it is associated. The provided execution is the execution that arrives in the node. The interface `ActivityExecution`

exposes special methods to control the execution flow.

```
public interface ActivityExecution extends OpenExecution {

  void waitForSignal();
  void take(String transitionName);
  void execute(String nodeName);

  ...

}
```

Event listener API

The event listener API allows for listeners to be developed in Java code and that are invoked on specific process events like entering a node or leaving a node. It is very similar to the activity API, but the difference is that the propagation of the execution flow cannot be controlled. E.g. when an execution is taking a transition, a listener to that event can be notified, but since the transition is already being taking, the execution flow cannot be changed by the event listeners.

```
public interface EventListener extends Serializable {

  void notify(EventListenerExecution execution) throws Exception;

}
```

Client API

The client API was already introduced above in the object execution mode and embedded execution mode. It's an interface that exposes the methods for managing executions on the plain process definition and execution objects directly.

At a minimal, the client API and the activity API are needed to create some a process definition with activities and execute it.

Environment

In the persistent execution mode, the first purpose of the environment is to enable processes to be executed in different transactional environments like standard Java, enterprise Java, SEAM and Spring.

The PVM code itself will only use transactional resources through self-defined interfaces. For example, the PVM itself has interfaces for some methods on the hibernate session, a async messaging session and a timer session.

The environment allows to configure the actual implementations, lazy initialization of the services on a request-basis and caching the service objects for the duration of the transaction.

An environment factory is static and one environment factory can serve all the threads in an application.

```
EnvironmentFactory environmentFactory = new
PvmEnvironmentFactory("environment.cfg.xml");
Environment blocks can surround persistent process operations like this:
Environment environment = environmentFactory.openEnvironment();
try {
```

```
  ... inside the environment block...

} finally {
  environment.close();
}
```

The PVM itself will fetch all it's transactional resources and configurations from the environment. It's recommended that `Activity` implementations do the same.

Commands

Commands encapsulate operations that are to be executed within an environment block. The main purpose for commands is to capture the logic of

```
public interface Command<T> extends Serializable {

  T execute(Environment environment) throws Exception;

}
```

Services

There are three services: `ProcessService`, `ExecutionService` and `ManagementService`. In general, services are session facades that expose methods for persistent usage of the PVM. The next fragments show the essential methods as example to illustrate those services.

The `ProcessService` manages the repository of process definitions.

```
public interface ProcessService {

  ProcessDefinition deploy(Deployment deployment);

  ProcessDefinition findLatestProcessDefinition(String
processDefinitionName);

  ...

}
```

The `ExecutionService` manages the runtime executions.

```
public interface ExecutionService {

  Execution startExecution(String processDefinitionId, String executionKey);

  Execution signalExecution(String executionId, String signalName);

  ...

}
```

The `ManagementService` groups all management operations that are needed to keep the system up and running.

```
public interface ManagementService {

  List<Job> getJobsWithException(int firstResult, int maxResults);

  void executeJob(String jobId);

  ...

}
```

The implementation of all these methods is encapsulated in `Command`s. And the three services all delegate the execution of the commands to a `CommandService`:

```
public interface CommandService {

  <T> T execute(Command<T> command);

}
```

The `CommandService` is configured in the environment. A chain of CommandServices can act as interceptors around a command. This is the core mechanism on how persistence and transactional support can be offered in a variety of environments.

From the default configuration which is included in full above, here is the section that configures the services

```
<contexts xmlns="http://jbpm.org/pvm/1.0/wire">

  <environment-factory>

    <process-service />
    <execution-service />
    <management-service />

    <command-service>
      <retry-interceptor />
      <environment-interceptor />
      <standard-transaction-interceptor />
    </command-service>

    ...
```

The three services `process-service`, `execution-service` and `management-service` will look up the configured `command-service` by type. The `command-service` tag corresponds to the default command service that essentially does nothing else then just execute the command providing it the current environment.

The configured `command-service` results into the following a chain of three interceptors followed by the default command executor.

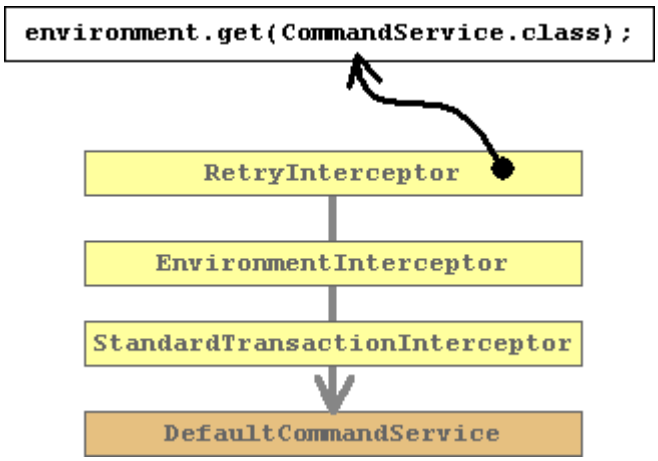


Figure: The CommandService interceptors

The retry interceptor is the first in the chain and that one that will be exposed as the `CommandService.class` from the environment. So the retry interceptor will be given to the respective services `process-service`, `execution-service` and `management-service`.

The `retry-interceptor` will catch hibernate StaleObjectExceptions (indicating optimistic locking failures) and retry to execute the command.

The `environment-interceptor` will put an environment block around the execution of the command.

The `standard-transaction-interceptor` will initialize a `StandardTransaction`. The hibernate session/transaction will be enlisted as a resource with this standard transaction.

Different configurations of this interceptor stack will also enable to

- delegate execution to a local ejb command service so that an container managed transaction is started.
- delegate to a remote ejb command service so that the command actually gets executed on a different JVM.
- package the command as an asynchronous message so that the command gets executed asynchronously in a different transaction.

Threads

The client API always performs it's work in the thread of the client. This includes methods like starting a new execution for a process definition and providing an external trigger.

The motivation for this is that the process engine itself should not enforce multithreaded computation in case it's not needed. That will only make the usage of the process engine harder to integrate and test.

The control is returned to the client when process execution reaches

- a wait state
- an asynchronous continuation

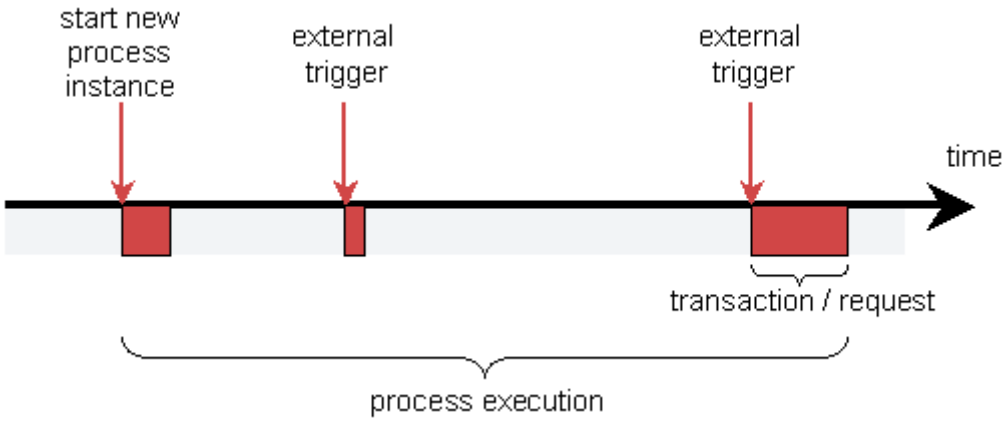
In case a client wants all the process engine work done in a separate thread, then the service API should be used and an asynchronous command service could be installed.

Transactions

All work for executing a process is always done by some Java thread in which an execution interpretes the process definition diagram. In case of persistent execution mode, all updates to the state of the execution should be stored in the database. Those updates are grouped in transactions.

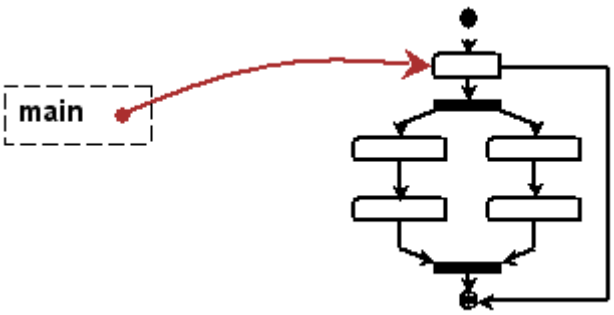
The natural demarcation of the transaction is the wait state in a process. Consider a process like a theoretical state machine. A state machine transitions its state instantanious. This corresponds perfect to our notion of a database transaction. Process executions move from one state to another in a transaction.

Typically the automatic work to be done in each state change is much less then 1 second. This includes the process updates and the automated business logic associated to the process execution. Then inbetween the transactions (the wait states) there is a long time in which the execution state is just persisted in the DB.

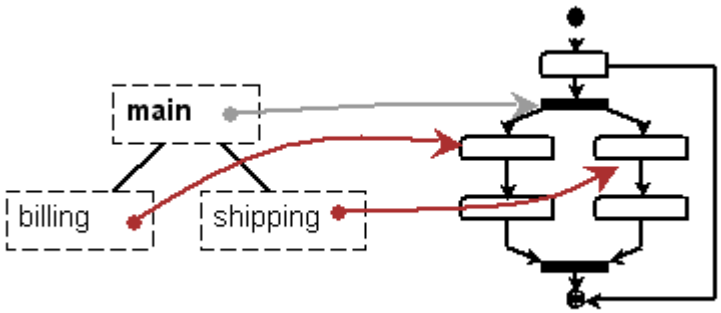


Often for developers the difference between the notion of a thread and a path of execution is confusing, especially when they look at the implementation of the jPDL fork, which is done in a single thread.

But the difference between



and



is just 3 SQL statements (2 inserts and an update). In practice, there is usually not much automated work being done on those transitions. The only typical heavy weight operation is generation of PDF documents. As long as that is not positioned on those transitions, it is most performant when those 3 updates are done in a single transaction, rather then forcing multiple transactions for this work.

And once you place all that work in one transaction, there is no real need to start multiple threads and then synchronize the transactional resources being used in those concurrent threads.

Potentially all transitions leaving a fork could be done in an asynchronous continuation. Logically that makes sense. But I doubt if that would be a good default as in most cases, this will only increase the number of transactions and bring down the throughput capabilities.