

# Application Delivery Fundamentals



camunda BPM platform

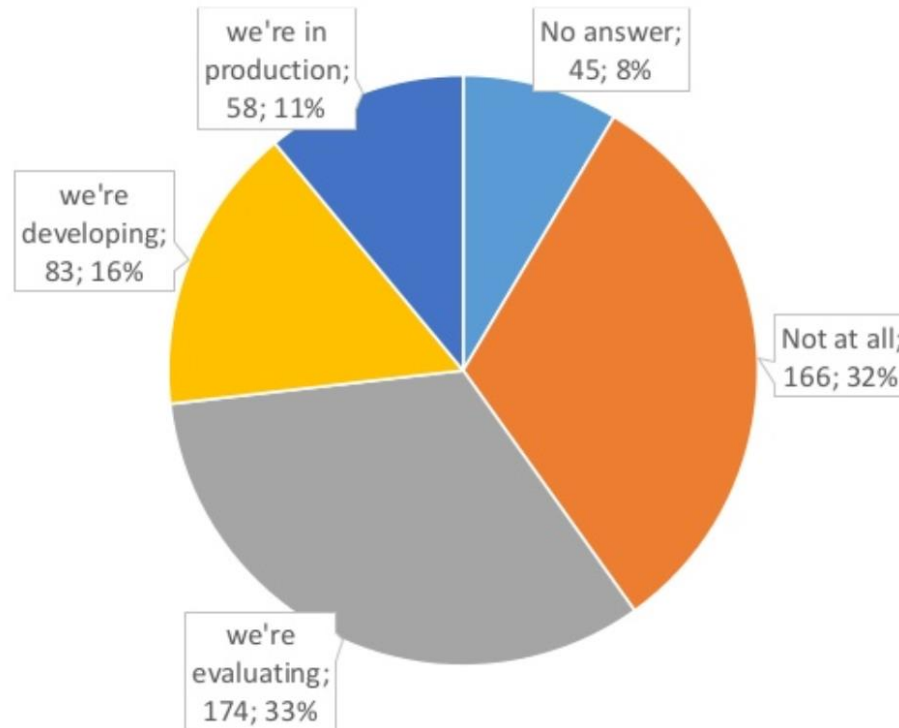
High performance. Delivered.

Parameswari Ettiappan

consulting | technology | outsourcing

- BPMN
- DMN Training
- BPM & Process Automation
- Process Engine
- Process Applications
- Camunda BPM and Microservices
- Tools & Infrastructure
- Camunda BPM Docker / Open shift Deployment

# Do you already use Camunda?



# Why do you use Camunda?



## Reasons:

- Executes BPMN 2.0
- Best option for Java
- is Open Source

## Advantages:

- Standardized technologies and Skills
- More flexible
- Easier, quicker development
- cost-effective

Also trusted by:

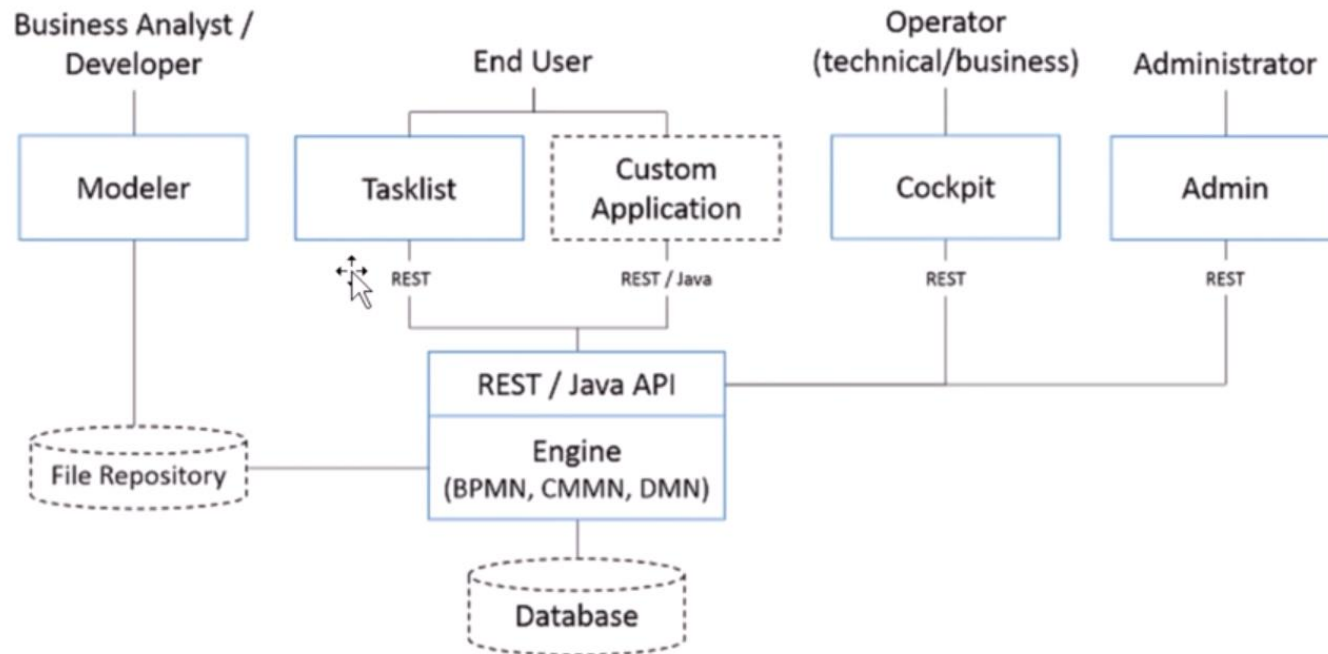


**Lufthansa Technik**

**Sparda-Bank**

 **zalando**

# Camunda BPM Components



Model

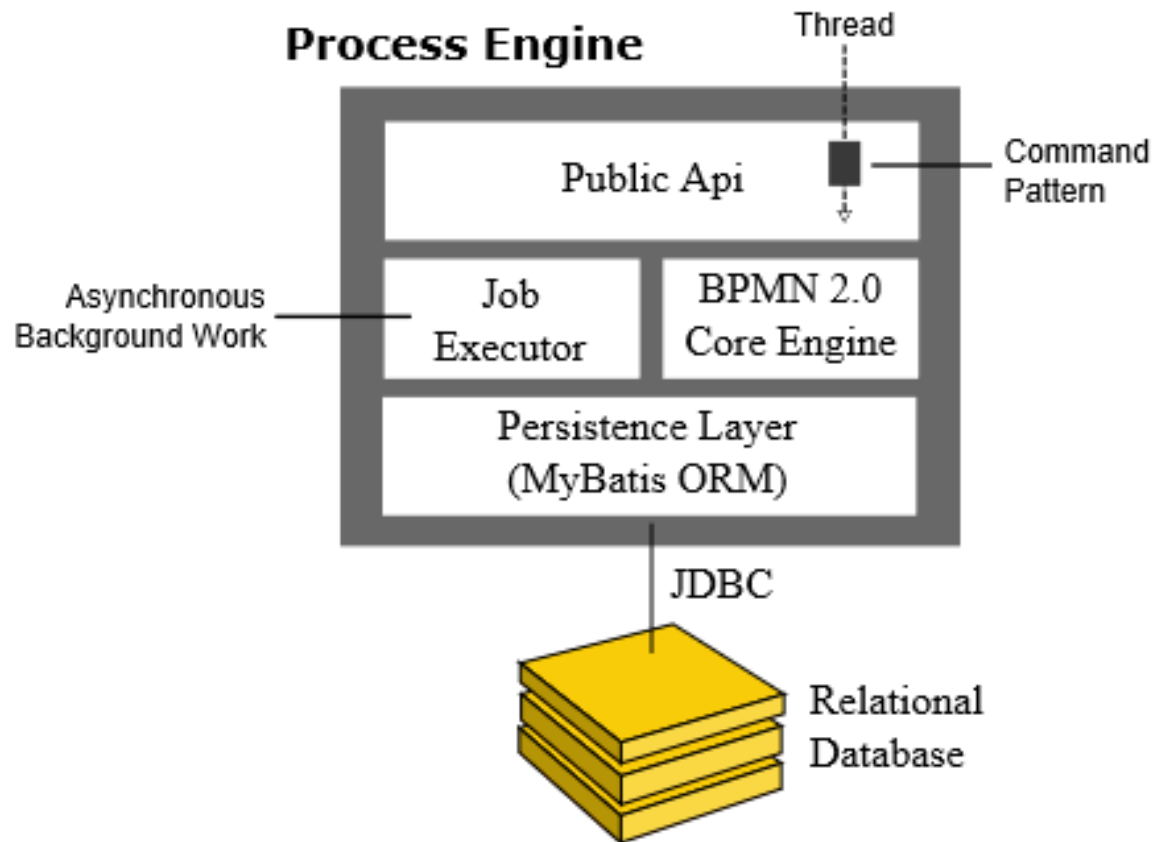
Execute

- Camunda BPM is a Java-based framework.
- The main components are written in Java.
- It has focus on providing Java developers with the tools they need for designing, implementing and running business processes and workflows on the JVM.
- It provides process engine technology available to non-Java developers.
- This is why Camunda BPM also provides a REST API which is used to build applications connecting to a remote process engine.



- Camunda BPM can be used both as a standalone process engine server or embedded inside custom Java applications.

# Process Engine Architecture





- BPMN 2.0 Core Engine: This is the core of the process engine.
- It features a lightweight execution engine for graph structures (PVM - Process Virtual Machine).
- A BPMN 2.0 parser which transforms BPMN 2.0 XML files into Java Objects and a set of BPMN Behavior implementations.

# Process Engine Public API

- Process Engine Public API: Service-oriented API allowing Java applications to interact with the process engine.
- The different responsibilities of the process engine (i.e., Process Repository, Runtime Process Interaction, Task Management, ...) are separated into individual services.
- The public API features a command-style access pattern:
- Threads entering the process engine are routed through a Command Interceptor which is used for setting up Thread Context such as Transactions.

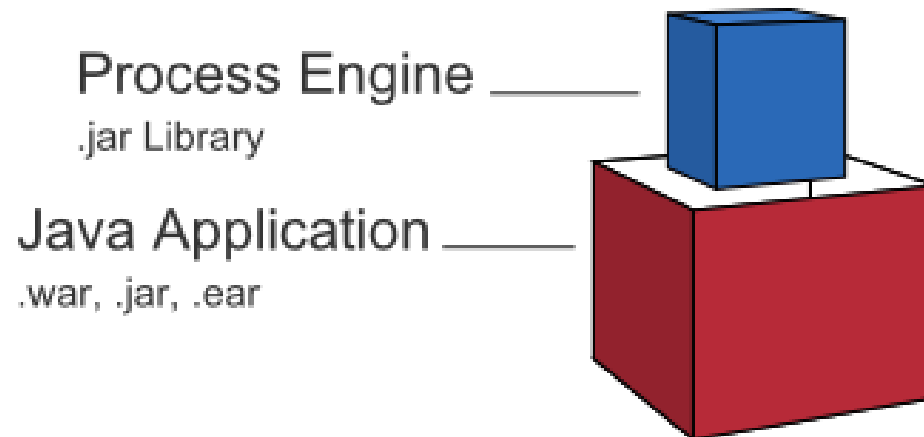
- Job Executor: The Job Executor is responsible for processing asynchronous background work such as Timers or asynchronous continuations in a process.
- The Persistence Layer: The process engine features a persistence layer responsible for persisting process instance state to a relational database.
- It uses MyBatis mapping engine for object relational mapping.



- Camunda BPM platform is a flexible framework which can be deployed in different scenarios.
  - Embedded Process Engine
  - Shared, Container-Managed Process Engine
  - Standalone (Remote) Process Engine Server
  - Clustering Model

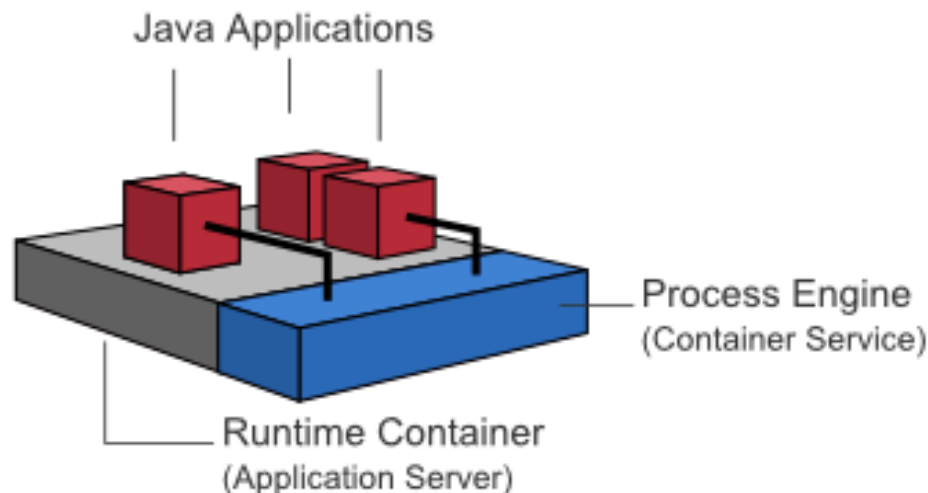
# Embedded Process Engine

- The process engine is added as an application library to a custom application.
- This way, the process engine can easily be started and stopped with the application lifecycle.
- It is possible to run multiple embedded process engines on top of a shared database.



# Shared, Container-Managed Process Engine

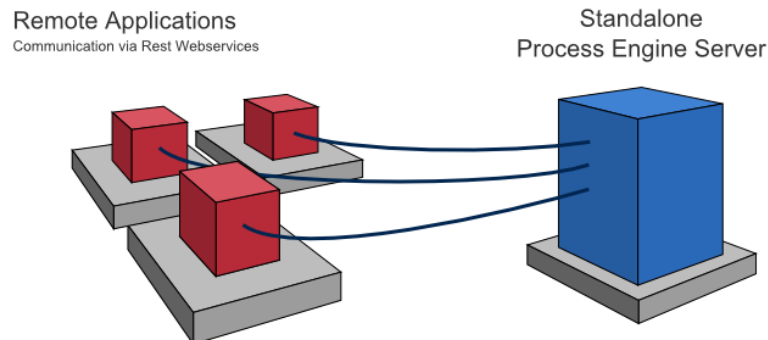
- In this case, the process engine is started inside the runtime container (Servlet Container, Application Server, ...).
- The process engine is provided as a container service and can be shared by all applications deployed inside the container.
- The process engine keeps track of the process definitions deployed by an application and delegates execution to the application.



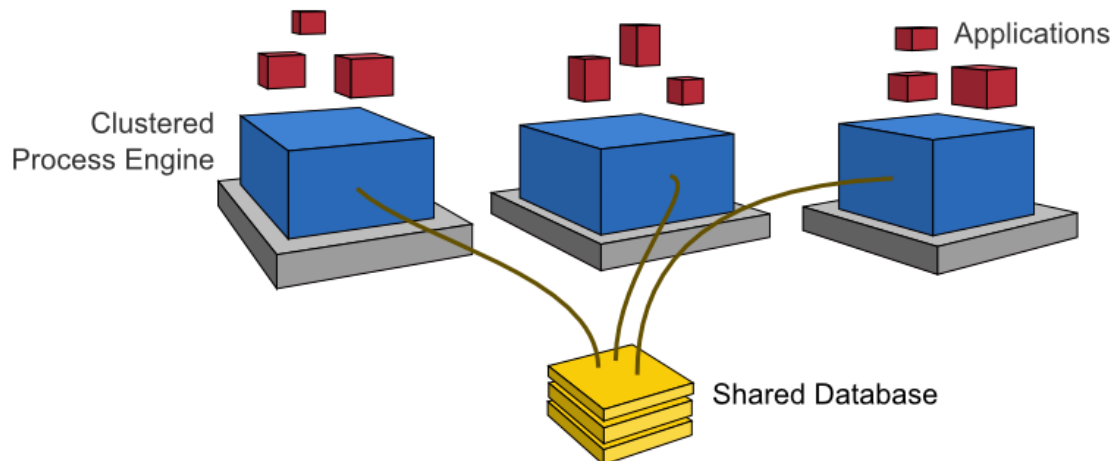
# Standalone (Remote) Process Engine Server



- The process engine is provided as a network service.
- Different applications running on the network can interact with the process engine through a remote communication channel.
- The easiest way to make the process engine accessible remotely is to use the built-in REST API.
- Different communication channels such as SOAP Web services or JMS are possible but need to be implemented by users.



- In order to provide scale-up or fail-over capabilities, the process engine can be distributed to different nodes in a cluster.
- Each process engine instance must then connect to a shared database.





- The individual process engine instances do not maintain session state across transactions.
- Whenever the process engine runs a transaction, the complete state is flushed out to the shared database.
- This makes it possible to route subsequent requests which do work in the same process instance to different cluster nodes.
- As far as the process engine is concerned, there is no difference between setups for scale-up and setups for fail-over (as the process engine keeps no session state between transactions).



- Camunda BPM doesn't provide load-balancing capabilities or session replication capabilities out of the box.
- The load-balancing function would need to be provided by a third-party system.
- Session replication would need to be provided by the host application server.




- “Sticky sessions” could be configured and enabled within your load balancing solution.
- This would ensure that all requests from a given user are directed to the same instance over a configurable period of time.
- Session sharing can be enabled in your application server such that the application server instances share session state.
- This would allow users to connect to multiple instances in the cluster without being asked to login multiple times.

# Selecting the Process Engine Mode



## Selecting the **Process Engine Mode**

	Container-Managed Engine	Embedded Engine	Remote Engine
	Run the engine as a service preconfigured in your Java EE container.	Use the process engine as a simple library right within your own application, typically started via Spring Boot  .	Run the engine as an isolated BPM server only communicating with it via Web Services.
<b>Engine Bootstrap / Lifecycle Management</b>	Out-of-the-box	Out-of-the-box for Spring Boot, otherwise do-it-yourself (see options below)	
<b>Camunda Webapps work in all use-cases</b>	✓	See limitations below	
<b>Camunda REST API works in all use-cases</b>	✓	See options below	
<b>Multiple <b>Process Applications</b> can share a central engine</b>	✓	Doable with a shared database, but requires custom development and	

# Selecting the Process Engine Mode





Multiple Engines can share resources (e.g. share the <a href="#">Job Executor</a> )	✓		The Remote Engine is bootstrapped either as embedded or container-managed engine
One application WAR/EAR can include the process engine		✓	
Supports untouched ("vanilla") containers		✓	
Runs in every Java environment	On <a href="#">Supported Containers</a>	✓	
Development Effort	Minimal	Depends	
Responsibility for Engine Installation and Configuration	Operations or Application Developer	Application Developer	
Application point of view on process engine	Library	Library	Remote Server
Possible communication types with services	Java InVM, Remote	Java InVM, Remote	Remote

# Selecting the Process Engine Mode




Camunda Best Practices

Changelog 

Use when	You use a supported application server and prefer to separate engine installation from application development	You use Spring Boot 	Your architecture or applications are not Java based.
		You want a single deployment including the engine.	You cannot touch your core application.
		You cannot make changes to your container.	For security, your BPM platform is separated from applications.
		You use a container we do not support.	

# Understanding **Embedded Engine** Specifics

## Using Spring Boot


The Camunda Spring Boot Starter is a clean way of controlling the embedded engine easily, so you don't have to think about the specifics mentioned below in this section. This makes Spring Boot the  **GREENFIELD** choice for Camunda projects.

## Bootstrapping the Engine and Managing its Lifecycle

When running the engine in embedded mode you have to control the **lifecycle** of the engine yourself, basically **starting up** and **shutting down** the engine - and providing access to the API whenever a client needs it. You have several options to do that.


	Spring Boot	Spring Application Context	processes.xml	Programmatic
	Configure, start and stop the engine via Spring Boot Starter	Configure, start and stop the engine via Spring Beans defined in your Application Context.	Configure, start and stop the engine via Camunda's processes.xml descriptor and a ProcessApplication class.	Configure, start and stop the engine yourself programmatically by using Java code.
Use when	You target Spring Boot as runtime environment.	You already use Spring.	You do not want to introduce a Spring dependency just for Camunda.	You need full control over the engine or want to do advanced customizations.
Unlimited Configuration Options	✓	✓		✓
Development Effort	Low	Medium	Low	High

## Choosing a Programming Framework

	Spring	Spring Boot 	CDI	OSGI / Blueprint
	The Java EE alternative to write JVM based systems.	Spring applications with convention over configuration.	Contexts and Dependency Injection (CDI) for Java EE.	A Dynamic Module System for Java.
Use to Control Embedded Engine Lifecycle	✓	✓		
Use Named Beans in Expression Language	✓	✓	✓	✓
Required Skill Level	Spring basics to get started	Spring Boot basics to get started	CDI basics to get started	Big experience with OSGI, introduces complexity many customers struggle with
Use when	Use the framework <b>you are already experienced with!</b>			
Supported in Enterprise Edition	✓	✓	✓	<a href="#">Community Extension</a>



## Choosing a Container / Application Server

		Java SE containers		Wildfly	Other Java EE containers	Unsupported Containers	OSGI Containers
		Tomcat (see supported Java SE environments)	Spring Boot Starter 	JBoss AS, Wildfly, Glassfish, IBM WAS, Oracle WLS (see supported Java EE environments)			Apache Karaf
(Out-of-the-box) JTA Transaction Integration			(✓)	✓		Maybe	
Java EE Features (JPA, JSF, CDI, ...)			(✓)	✓		Maybe	
Engine Mode	Container-Managed Engine	✓		✓			Community Extension
	Embedded Engine	✓	✓	✓		✓	✓
Supported with Enterprise Edition		✓	✓	✓		Engine only	Engine only
Use when		Use the container <b>you already operate or are experienced with!</b>					
		Tomcat	Spring Boot	Wildfly, Glassfish, IBM WAS, Oracle WLS			Apache Karaf

# Considering the User Interface

## Understanding the Use Cases


In the User Interface you might have different components, for which you might even decide differently.

	Organisation Internal			Public
	Tasklist Application	Custom Business Application	Monitoring and/or Reporting Application	Internet Website
	A (central) dedicated tasklist, showing tasks from potentially different contexts.	Your business application includes use cases steering the process engine.	An application showing state of processes, based on instances or as statistics.	Your internet website e.g. kicks off process instances when a user submits an order.
Target Group	All employees	Specific employees	Operators (Business/Technical)	Customers
Published in	Intranet			Internet
Required Usability	Generic tasklist, customizable for roles	Tailored to a specific business use case	Different for business or technical operations	Easy-to-use modern web application
Required Technology		Use technology of business application		Use modern HTML5 stack
Possibility to Reuse	✓		✓	

## Choosing a **User Interface Technology**

	HTML5	Java UI Frameworks	Non Java Technology	Portal
	e.g. JavaScript, AngularJS	e.g. JSF, Vaadin, Wicket	e.g. PHP	e.g. Portlets, Liferay
<b>Learning Curve for Java Developers</b>	high	low	medium	depends on portal
<b>Typical Use Cases</b>	Single Page Applications	Java-based Applications and Form-based Applications	Standalone Web Applications	Company wide portals
<b>Use when</b>	Use the framework <b>you are experienced with or want to build up experience with!</b> (Don't forget about the learning curve!)			

## Using a **Build Tool**

	<b>Maven</b>  <b>GREENFIELD</b>	Gradle	Ant	Other
	Manages a project's build, reporting and documentation from a project object model (POM) file.	Automates projects written in Java and other JVM languages, including Groovy, Scala and Clojure.	Java build tool driving tasks and processes described as targets dependent upon each other.	
<b>Used in Camunda examples</b>	✓			
<b>Use when</b>	Always, if there is no reason driving you in a different direction.	If you already use Gradle and have experience with it.	Try to avoid	Try to avoid



[Learn More](#)
[Learn More](#)
[Learn More](#)

## Choosing a Database

Camunda requires a **relational database** for persistence.



Even if the persistence provider can be plugged and e.g. exchanged by some **NoSQL** persistence this is neither recommended nor supported. Therefore, if you have use cases for this discuss them with Camunda beforehand!

	<b>PostgreSQL</b>  <b>GREENFIELD</b> (Prod)	<b>Oracle</b>	<b>H2</b>  <b>GREENFIELD</b> (Test)	<b>Other Database</b>
	PostgreSQL is an open source object-relational database system.	Oracle Database is a commercial object-relational database system.	H2 is a Java SQL database with in-memory mode and a small footprint.	
<b>Best Performance Observations</b>	✓	✓		
<b>In-Memory Mode</b>			✓	
<b>No installation required</b>			✓	
<b>Recommended for Production Use</b>	✓	✓		✓ (if <a href="#">supported</a> )

- The Camunda BPM web applications are based on a RESTful architecture.
- Frameworks used:
- JAX-RS based Rest API
- AngularJS
- RequireJS
- jQuery
- Twitter Bootstrap
- camunda-bpmn.js
- ngDefine
- angular-data-depend

- Application-Embedded Process Engine
  - All Java application servers
  - Camunda Spring Boot Starter: Embedded Tomcat
- Container-Managed Process Engine and Camunda Cockpit, Tasklist, Admin
  - Apache Tomcat 7.0 / 8.0 / 9.0
  - JBoss EAP 6.4 / 7.0 / 7.1 / 7.2
  - Wildfly Application Server 10.1 / 11.0 / 12.0 / 13.0 / 14.0 / 15.0 / 16.0 / 17.0 / 18.0
  - IBM WebSphere Application Server 8.5 / 9.0 (Enterprise Edition only)
  - Oracle WebLogic Server 12c (12R1,12R2) (Enterprise Edition only) and Deployment scenarios).

- Databases
- Supported Database Products
  - MySQL 5.6 / 5.7
  - MariaDB 10.0 / 10.2 / 10.3
  - Oracle 11g / 12c / 18c
  - IBM DB2 10.5 / 11.1 (excluding IBM z/OS for all versions)
  - PostgreSQL 9.4 / 9.6 / 10.4 / 10.7 / 11.1 / 11.2
  - Amazon Aurora PostgreSQL compatible with PostgreSQL 9.6 / 10.4 / 10.7
  - Microsoft SQL Server 2012/2014/2016/2017
  - H2 1.4



# Container/Application Server for Runtime Components

---



- Web Browser
  - Google Chrome latest
  - Mozilla Firefox latest
  - Internet Explorer 11
  - Microsoft Edge
- Java
  - Java 8 / 9 / 10 / 11 / 12 / 13 (if supported by your application server/container)

- Java Runtime
  - Oracle JDK 8 / 9 / 10 / 11 / 12 / 13
  - IBM JDK 8 (with J9 JVM)
  - OpenJDK 8 / 9 / 10 / 11 / 12 / 13, including builds of the following products:
    - Oracle OpenJDK
    - AdoptOpenJDK (with HotSpot JVM)
    - Amazon Corretto
    - Azul Zulu

# Container/Application Server for Runtime Components



- Camunda Modeler
- Supported on the following platforms:
  - Windows 7 / 10
  - Mac OS X 10.11
  - Ubuntu LTS (latest)
- Reported to work on
  - Ubuntu 12.04 and newer
  - Fedora 21
  - Debian 8

- List of Community Extensions
- The following is a list of current (unsupported) community extensions:
  - Apache Camel Integration
  - Camunda Docker Images
  - Custom Batch
  - DMN Scala Extension
  - Elastic Search Extension
  - Email Connectors
  - FEEL Scala Extension
  - Grails Plugin

# Container/Application Server for Runtime Components



- GraphQL API
- Keycloak Identity Provider
- Migration API
- Mockito Testing Library
- Needle Testing Library
- OSGi Integration
- PHP SDK
- Process Test Coverage
- Reactor Event Bus
- REST Client Spring Boot
- Scenario Testing Library
- Single Sign On for JBoss
- Tasklist Translations
- Wildfly Swarm

# Container/Application Server for Runtime Components



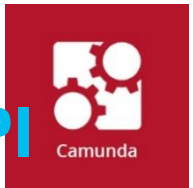
- Definition of Public API
- The Camunda BPM public API is limited to the following items:
- Java API:
- All non-implementation Java packages (package name does not contain impl) of the following modules.
- camunda-engine
- camunda-engine-spring
- camunda-engine-cdi
- camunda-engine-dmn
- camunda-bpmn-model

# Container/Application Server for Runtime Components

---



- camunda-cmmn-model
- camunda-dmn-model
- camunda-spin-core
- camunda-connect-core
- camunda-commons-typed-valuesSwarm



# Bootstrap a Process Engine Using the Java API

- `ProcessEngineConfiguration.createStandaloneProcessEngineConfiguration();`
- `ProcessEngineConfiguration.createStandaloneInMemProcessEngineConfiguration();`

```
ProcessEngine processEngine =  
ProcessEngineConfiguration.createStandaloneInMemProcess  
sEngineConfiguration()  
.setDatabaseSchemaUpdate(ProcessEngineConfiguration.DB  
B_SCHEMA_UPDATE_FALSE)  
.setJdbcUrl("jdbc:h2:mem:my-own-  
db;DB_CLOSE_DELAY=1000")  
.setJobExecutorActivate("true")  
.buildProcessEngine();
```



# Configure Process Engine Using camunda cfg XML



```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="processEngineConfiguration"
class="org.camunda.bpm.engine.impl.cfg.StandaloneProcessEngineConfiguration">

    <property name="jdbcUrl" value="jdbc:h2:mem:camunda;DB_CLOSE_DELAY=1000" />

    <property name="jdbcDriver" value="org.h2.Driver" />

    <property name="jdbcUsername" value="sa" />

    <property name="jdbcPassword" value="" />

    <property name="databaseSchemaUpdate" value="true" />

    <property name="jobExecutorActivate" value="false" />

    <property name="mailServerHost" value="mail.my-corp.com" />

    <property name="mailServerPort" value="5025" />

  </bean></beans>
```



# Configure Process Engine in the bpm-platform.xml

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

<bpm-platform xmlns="http://www.camunda.org/schema/1.0/BpmPlatform"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://www.camunda.org/schema/1.0/BpmPlatform
http://www.camunda.org/schema/1.0/BpmPlatform">

  <job-executor>

    <job-acquisition name="default" />

  </job-executor>

  <process-engine name="default">

    <job-acquisition>default</job-acquisition>

    <configuration>org.camunda.bpm.engine.impl.cfg.StandaloneProcessEngineConfiguration</configuration>

    <datasource>java:jdbc/ProcessEngine</datasource>

    <properties>

      <property name="history">full</property>

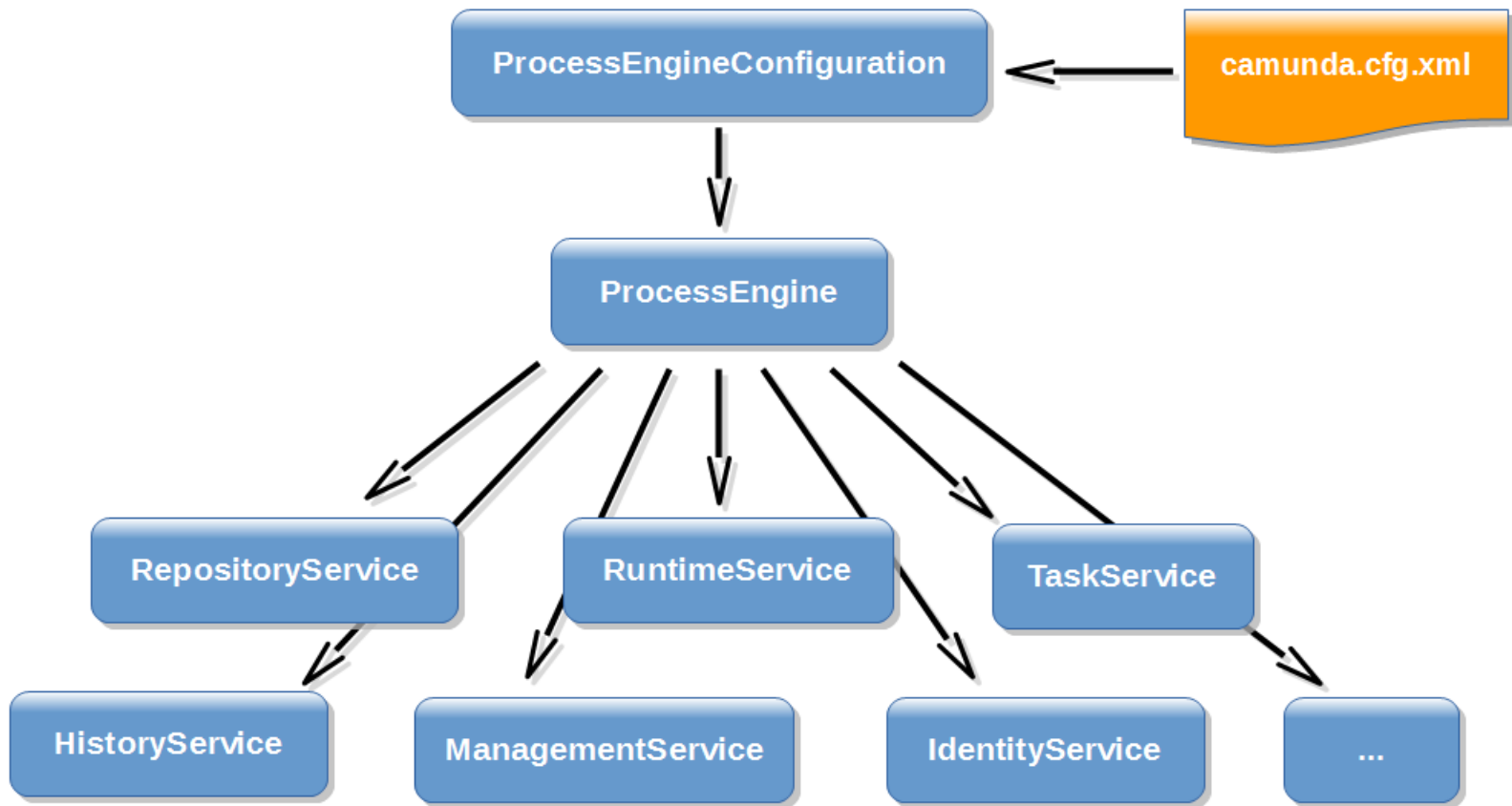
      <property name="databaseSchemaUpdate">true</property>

      <property name="authorizationEnabled">true</property>

    </properties>

  </process-engine>

</bpm-platform>
```





- The RepositoryService is probably the first service needed when working with the Camunda engine.
- This service offers operations for managing and manipulating deployments and process definitions.
- RuntimeService is dynamic. It deals with starting new process instances of process definitions.
- A process instance is one execution of such a process definition.
- For each process definition there are typically many instances running at the same time.
- The RuntimeService is also the service which is used to retrieve and store process variables.
- This is data specific to the given process instance and can be used by various constructs in the process (e.g., an exclusive gateway often uses process variables to determine which path is chosen to continue the process).
- The RuntimeService also allows to query on process instances and executions. Executions are a representation of the 'token' concept of BPMN 2.0.



- Tasks that need to be performed by actual human users of the system are core to the process engine.
- Everything around tasks is grouped in the Task Service, such as
  - Querying tasks assigned to users or groups.
  - Creating new standalone tasks. These are tasks that are not related to a process instances.
  - Manipulating to which user a task is assigned or which users are in some way involved with the task.
  - Claiming and completing a task. Claiming means that someone decided to be the assignee for the task, meaning that this user will complete the task.



- The Identity Service is pretty simple.
- It allows the management (creation, update, deletion, querying, ...) of groups and users.
- It is important to understand that the core engine actually doesn't do any checking on users at runtime.
- For example, a task could be assigned to any user, but the engine does not verify if that user is known to the system.
- This is because the engine can also be used in conjunction with services such as LDAP, Active Directory, etc.



- The Form Service is an optional service.
- This means that the Camunda engine can be used perfectly without it, without sacrificing any functionality.
- This service introduces the concept of a start form and a task form.
- A start form is a form that is shown to the user before the process instance is started, while a task form is the form that is displayed when a user wants to complete a form.



- The HistoryService exposes all historical data gathered by the engine.
- When executing processes, a lot of data can be kept by the engine (this is configurable) such as process instance start times, who did which tasks, how long it took to complete the tasks, which path was followed in each process instance, etc.
- This service exposes mainly query capabilities to access this data.





- The Management Service is typically not needed when coding custom applications.
- It allows to retrieve information about the database tables and table metadata.
- Furthermore, it exposes query capabilities and management operations for jobs.
- Jobs are used in the engine for various things such as timers, asynchronous continuations, delayed suspension/activation, etc.



- The Filter Service allows to create and manage filters.
  - Filters are stored queries like task queries.
  - For example filters are used by Tasklist to filter user tasks.
- 
- The External TaskService provides access to external task instances.
  - External tasks represent work items that are processed externally and independently of the process engine.



- The Case Service is like the RuntimeService but for case instances.
- It deals with starting new case instances of case definitions and managing the lifecycle of case executions.
- The service is also used to retrieve and update process variables of case instances.
- The Decision Service allows to evaluate decisions that are deployed to the engine.
- It is an alternative to evaluate a decision within a business rule task that is independent from a process definition.

```
ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
RepositoryService repositoryService = processEngine.getRepositoryService();
RuntimeService runtimeService = processEngine.getRuntimeService();
TaskService taskService = processEngine.getTaskService();
IdentityService identityService = processEngine.getIdentityService();
FormService formService = processEngine.getFormService();
HistoryService historyService = processEngine.getHistoryService();
ManagementService managementService =
processEngine.getManagementService();
FilterService filterService = processEngine.getFilterService();
ExternalTaskService externalTaskService =
processEngine.getExternalTaskService();
CaseService caseService = processEngine.getCaseService();
DecisionService decisionService = processEngine.getDecisionService();
```

# Process Variables

---

- Variables can be used to add data to process runtime state or, more particular, variable scopes.
- Various API methods that change the state of these entities allow updating of the attached variables.
- In general, a variable consists of a name and a value.
- The name is used for identification across process constructs.
- For example, if one activity sets a variable named var, a follow-up activity can access it by using this name.
- The value of a variable is a Java object.

# Variable Scopes and Variable Visibility

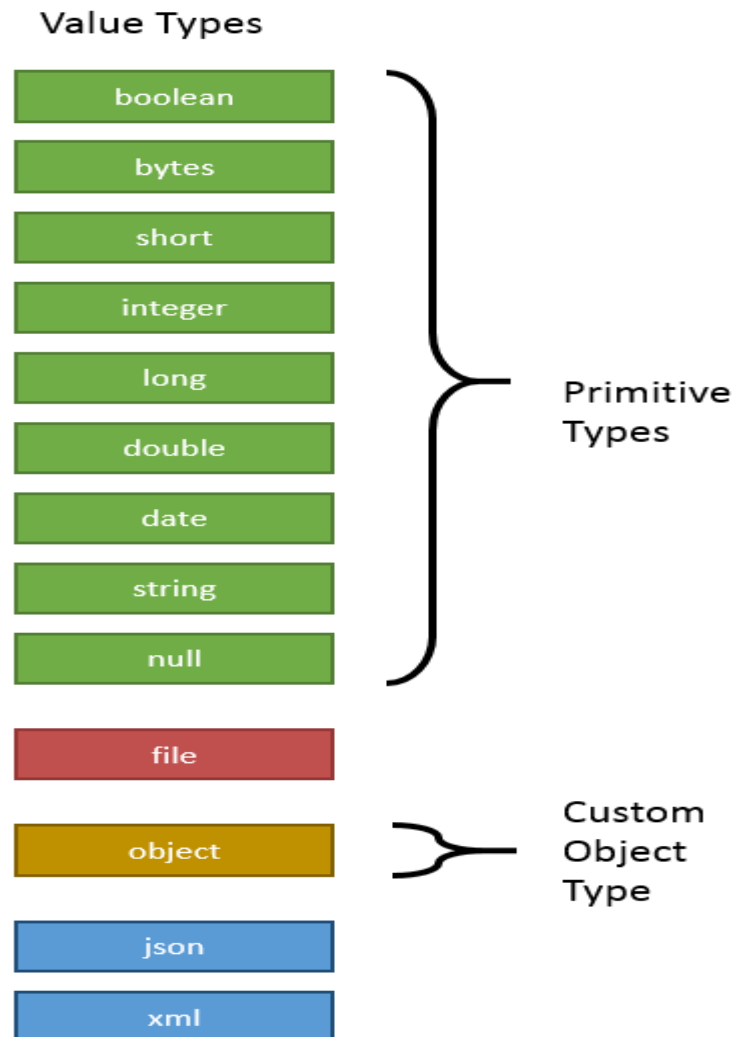
---

- In general, variables are accessible in the following cases:
- Instantiating processes
- Delivering messages
- Task lifecycle transitions, such as completion or resolution
- Setting/getting variables from outside
- Setting/getting variables in a Delegate
- Expressions in the process model
- Scripts in the process model
- (Historic) Variable queries

# Variable Scopes and Variable Visibility

- `// Java Object API: Get Variable`
- `Integer val1 = (Integer) execution.getVariable("val1");`
- `// Typed Value API: Get Variable`
- `IntegerValue typedVal2 = execution.getVariableTyped("val2");`
- `Integer val2 = typedVal2.getValue();`
- `Integer diff = val1 - val2;`
- `// Java Object API: Set Variable`
- `execution.setVariable("diff", diff);`
- `// Typed Value API: Set Variable`
- `IntegerValue typedDiff = Variables.integerValue(diff);`
- `execution.setVariable("diff", typedDiff);`

# Supported Variable Values





# Delegation Code

---

- Delegation Code allows you to execute external Java code, evaluate expressions or scripts when certain events occur during process execution.
- There are different types of Delegation Code:
- Java Delegates can be attached to a BPMN ServiceTask.
- Execution Listeners can be attached to any event within the normal token flow, e.g. starting a process instance or entering an activity.
- Task Listeners can be attached to events within the user task lifecycle, e.g. creation or completion of a user task.

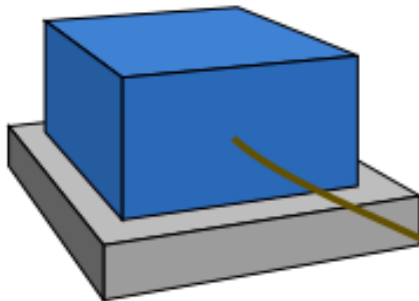
- public class ToUppercase implements JavaDelegate {
  - public void execute(DelegateExecution execution)  
throws Exception {
    - String var = (String) execution.getVariable("input");
    - var = var.toUpperCase();
    - execution.setVariable("input", var);
    - }
- }

- Multi-Tenancy regards the case in which a single Camunda installation should serve more than one tenant.
- For each tenant, certain guarantees of isolation should be made.
- For example, one tenant's process instances should not interfere with those of another tenant.

- Multi-Tenancy can be achieved in two different ways.
- One way is to use one process engine per tenant.
- The other way is to use just one process engine and associate the data with tenant identifiers.
- The two ways differ from each other in the level of data isolation, the effort of maintenance and the scalability.
- A combination of both ways is also possible.

# Single Process Engine With Tenant-Identifiers

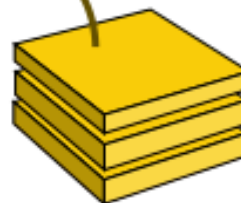
Process Engine  
with multiple Tenants



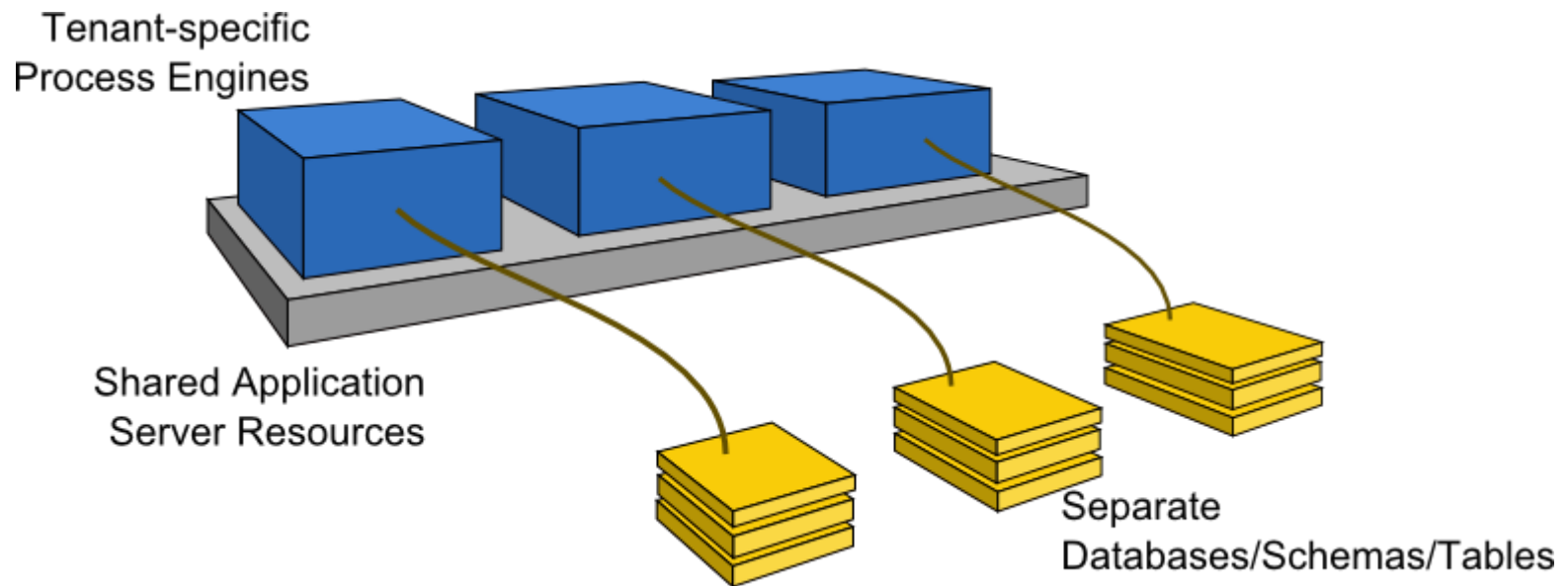
Database Rows  
with Tenant-Identifiers

ACT_RE_PROCDEF			
ID_	KEY_	...	TENANT_ID_
invoice:1:1f..	invoice	...	tenant1
invoice:1:8d..	invoice	...	tenant2

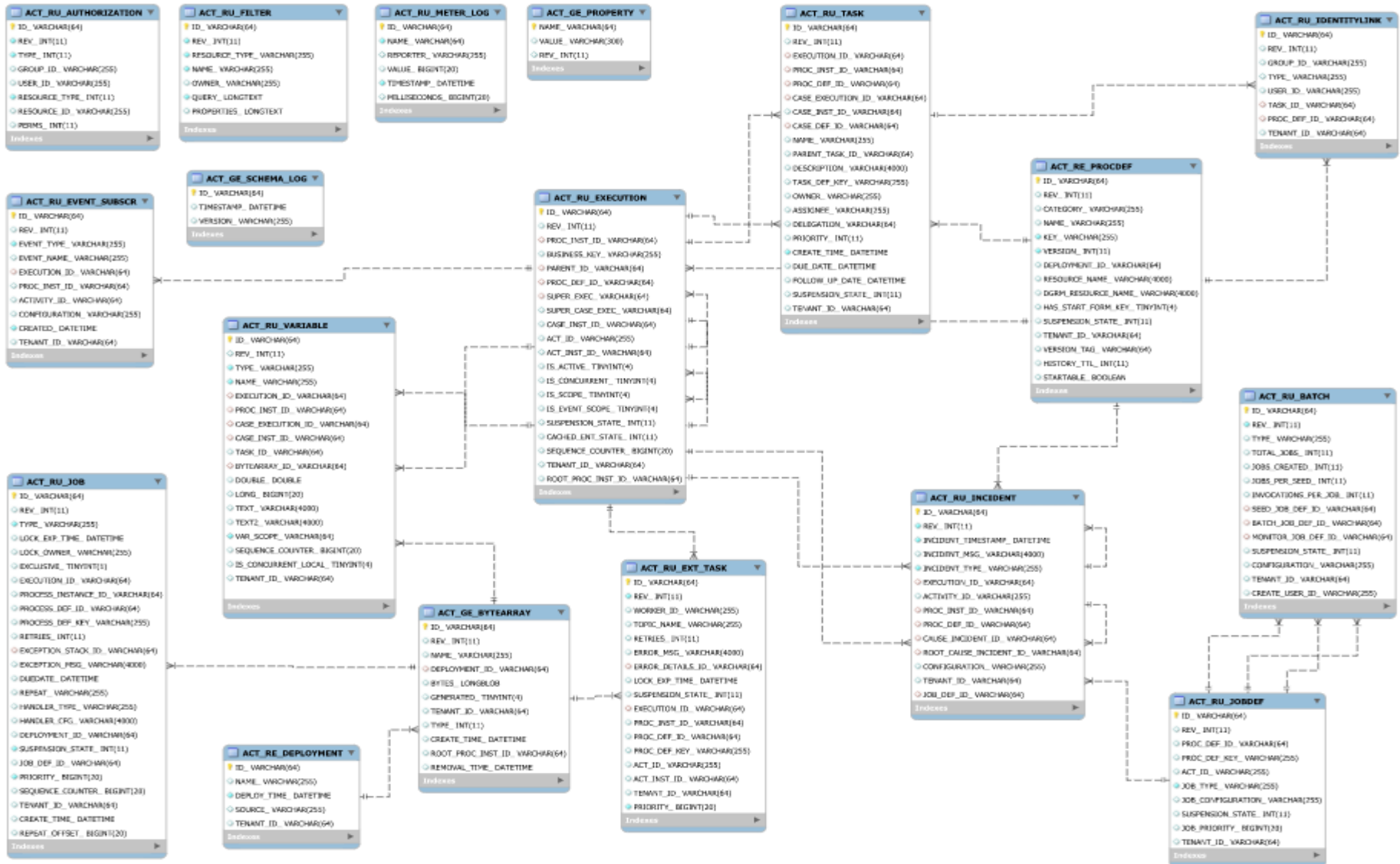
One Database  
for all Tenants



# One Process Engine Per Tenant



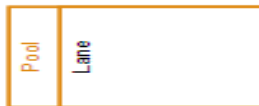
# Configuration for Microsoft SQL



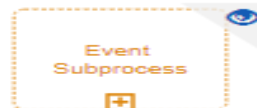
- Business Process Model and Notation (BPMN) is a standard for Workflow and Process Automation. Camunda supports the 2.0 version of BPMN.



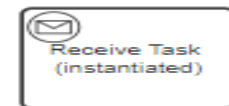
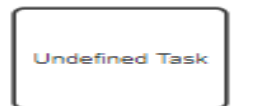
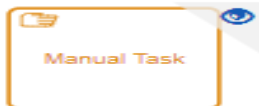
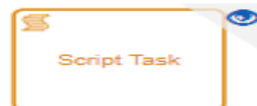
## Participants



## Subprocesses



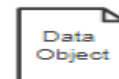
## Tasks



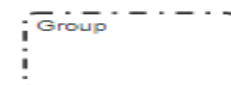
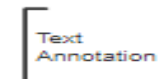
## Gateways



## Data



## Artifacts



- Start events define where a Process or Sub Process starts.
- The process engine supports different types of start events:
- Blank
- Timer
- Message
- Signal
- Conditional

## None Events (Blank)

- None events are unspecified events, also called 'blank' events.
- For instance, a 'none' start event technically means that the trigger for starting the process instance is unspecified.
- This means that the engine cannot anticipate when the process instance must be started.
- The none start event is used when the process instance is started through the API by calling one of the `startProcessInstanceBy...` methods.
- `ProcessInstance processInstance = runtimeService.startProcessInstanceByKey('invoice');`

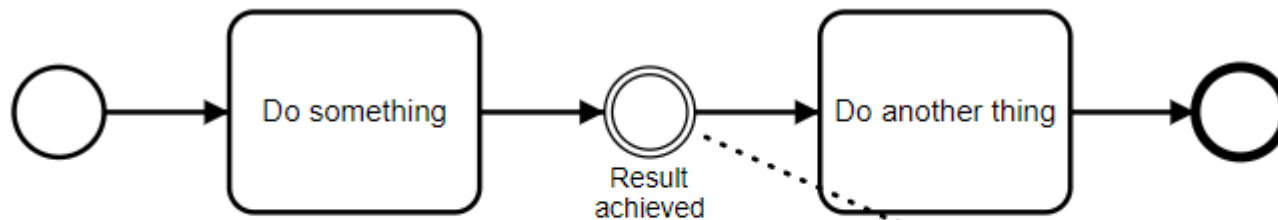
# None Events (Blank)



- A 'none' end event means that the result thrown when the event is reached is unspecified.
- As such, the engine will not do anything besides ending the current path of execution.
- The XML representation of a none end event is the normal end event declaration, without any sub-element (other end event types all have a sub-element declaring the type).
- `<endEvent id="end" name="my end event" />`

## Intermediate None Event (throwing)

- The following process diagram shows a simple example of an intermediate none event, which is often used to indicate some state achieved in the process.

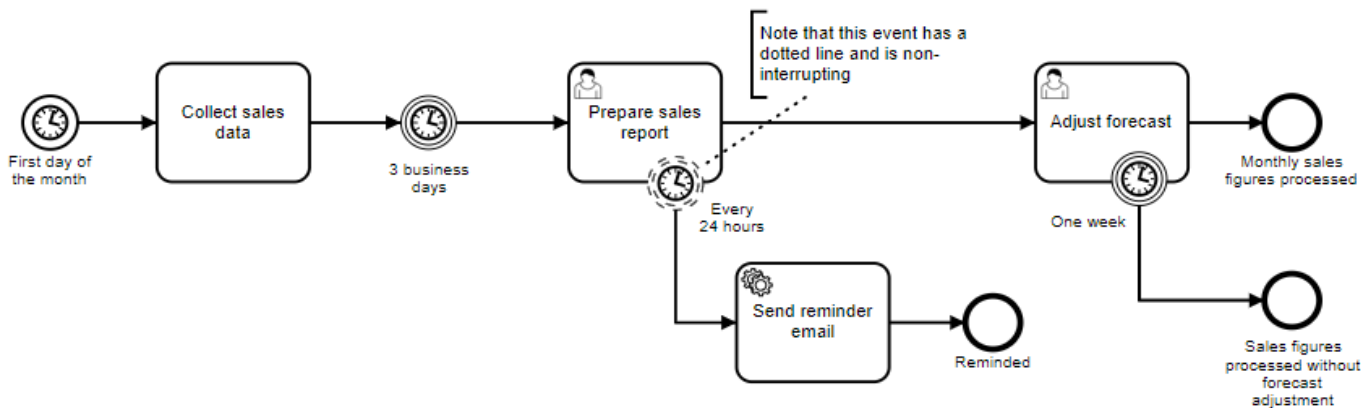


```

<intermediateThrowEvent id="noneEvent">
  <extensionElements>
    <camunda:executionListener
      class="org.camunda.bpm.engine.test.bpm
n.event.IntermediateNoneEventTest$MyEx
ecutionListener" event="start" />
  </extensionElements>
</intermediateThrowEvent>
  
```

Visualize a milestone, often a hook to send events to for KPI monitoring.

- Timer events are events which are triggered by a defined timer.
- They can be used as start event, intermediate event or boundary event. Boundary events can be interrupting or not.



## Defining a Timer

---

- `<timerEventDefinition>`
- `<timeDate>2011-03-11T12:13:14Z</timeDate>`
- `</timerEventDefinition>`



- To specify how long the timer should run before it is fired, a `timeDuration` can be specified as a sub-element of `timerEventDefinition`.
- It is possible to define the duration in two different ISO 8601 Durations formats:
  - `PnYnMnDTnHnMnS`
  - `PnW`
- `<timerEventDefinition>`
- `<timeDuration>P10D</timeDuration>`
- `</timerEventDefinition>`

# Events



Type	Start			Intermediate				End
	Normal	Event Subprocess	Event Subprocess non-interrupt	catch	boundary	boundary non-interrupt	throw	
None								
Message								
Timer								
Conditional								
Link								
Signal								
Error								
Escalation								

# Events



Termination								
Compensation								
Cancel								
Multiple								
Multiple Parallel								

- Service Task
  - Invoke or execute business logic.
- Send Task
  - Send a message.
- User Task
  - A task performed by a human participant.
- Business Rule Task
  - Execute an automated business decision.
- Script Task
  - Execute a Script.
- Receive Task
  - Wait for a message to arrive.
- Manual Task
  - A task which is performed externally.
- Task Markers
  - Markers control operational aspects like repetition.

- A Service Task is used to invoke services.
- In Camunda this is done by calling Java code or providing a work item for an external worker to complete asynchronously.



- Calling Java Code
- There are four ways of declaring how to invoke Java logic:
  - Specifying a class that implements a Java Delegate or Activity Behavior
  - Evaluating an expression that resolves to a delegation object
  - Invoking a method expression
  - Evaluating a value expression

```
<serviceTask id="javaService"  
    name="My Java Service Task"  
    camunda:class="org.camunda.bpm.MyJavaDelegate"  
/>
```

```
<serviceTask id="beanService"  
    name="My Bean Service Task"  
    camunda:delegateExpression="${myDelegateBean}" />
```

```
<serviceTask id="expressionService"  
    name="My Expression Service Task"  
    camunda:expression="${myBean.doWork()}" />
```



- `<serviceTask id="aMethodExpressionServiceTask"  
camunda:expression="#{myService.doSomething()}"  
camunda:resultVariable="myVar" />`



## External Tasks

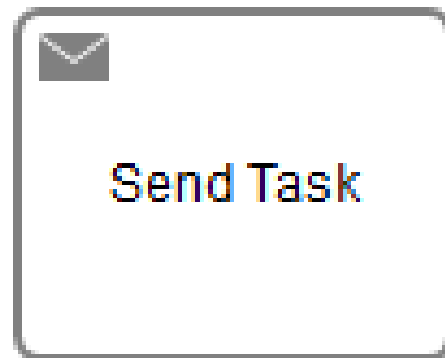
- When a Service Task is declared external, the process engine offers a work item to workers that independently poll the engine for work to do.
- This decouples the implementation of tasks from the process engine and allows to cross system and technology boundaries.
- Calling Java code, where the process engine synchronously invokes Java logic, External Task contrast to this.

# External Tasks

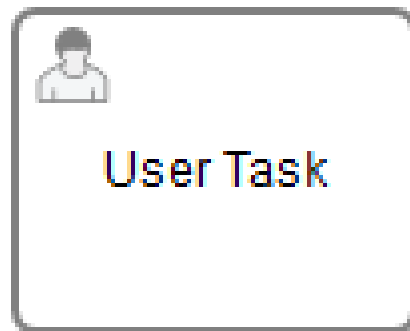


```
<serviceTask id="anExternalServiceTask"  
    camunda:type="external"  
    camunda:topic="ShipmentProcessing" />
```

- A Send Task is used to send a message. In Camunda this is done by calling Java code.
- The Send Task has the same behavior as a Service Task.
- `<sendTask id="sendTask"  
camunda:class="org.camunda.bpm.MySendTaskDelegate" />`

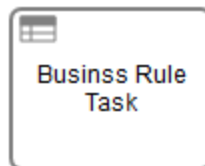


- A User Task is used to model work that needs to be done by a human actor.
- When the process execution arrives at such a User Task, a new task is created in the task list of the user(s) or group(s) assigned to that task.
- `<userTask id="theTask" name="Important task" />`



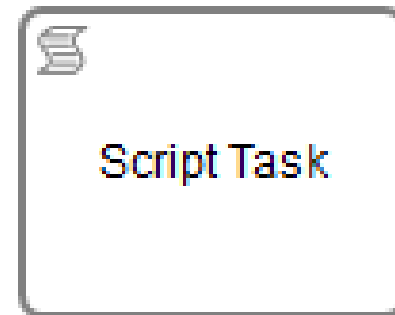
# Business Rule Task

- A Business Rule Task is used to synchronously execute one or more rules.
- `<businessRuleTask id="businessRuleTask"`
- `camunda:decisionRef="myDecision"`
- `camunda:decisionRefBinding="version"`
- `camunda:decisionRefVersion="12" />`



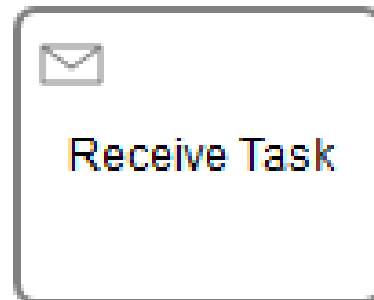
# Script Task

- A Script Task is an automated activity. When a process execution arrives at the Script Task, the corresponding script is executed.
- `<scriptTask id="theScriptTask" name="Execute script" scriptFormat="groovy">`
- `<script>`
- `sum = 0`
- `for ( i in inputArray ) {`
- `sum += i`
- `}`
- `</script>`
- `</scriptTask>`



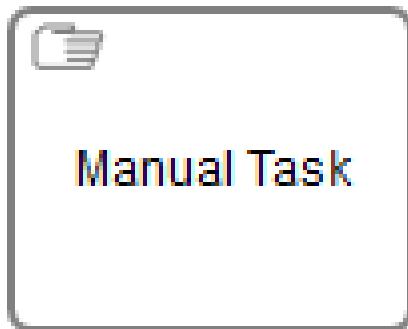
# Receive Task

- A Receive Task is a simple task that waits for the arrival of a certain message.
- When the process execution arrives at a Receive Task, the process state is committed to the persistence storage.
- This means that the process will stay in this wait state until a specific message is received by the engine, which triggers continuation of the process beyond the Receive Task.



# Manual Task

- A Manual Task defines a task that is external to the BPM engine.
- It is used to model work that is done by somebody who the engine does not need to know of and is there no known system or UI interface.
- For the engine, a manual task is handled as a pass-through activity, automatically continuing the process at the moment the process execution arrives at it.

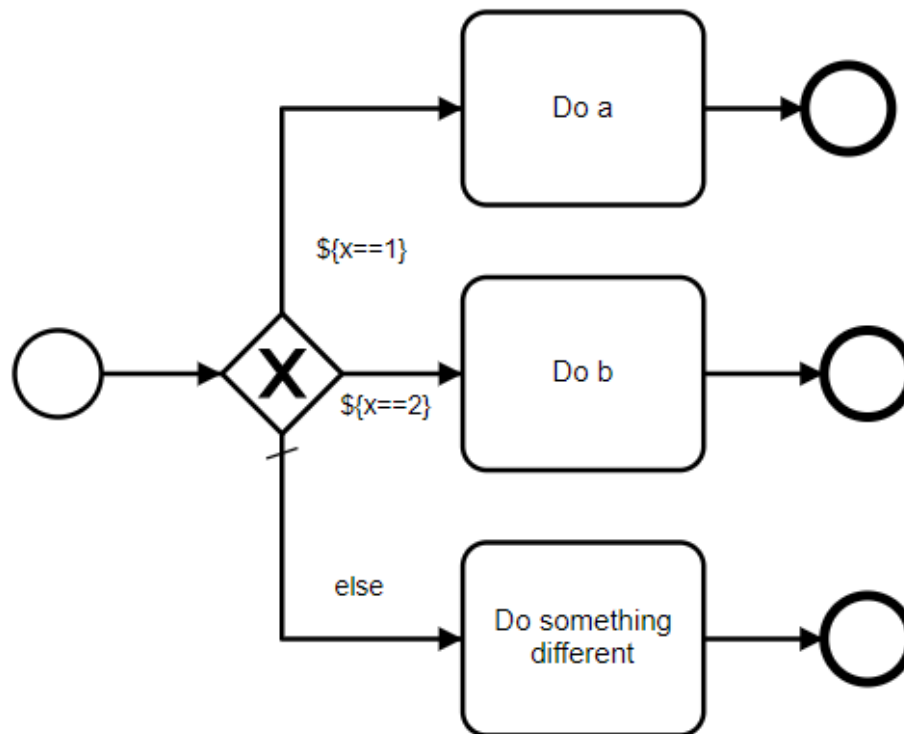


```
<manualTask id="myManualTask" name="Manual Task" />
```

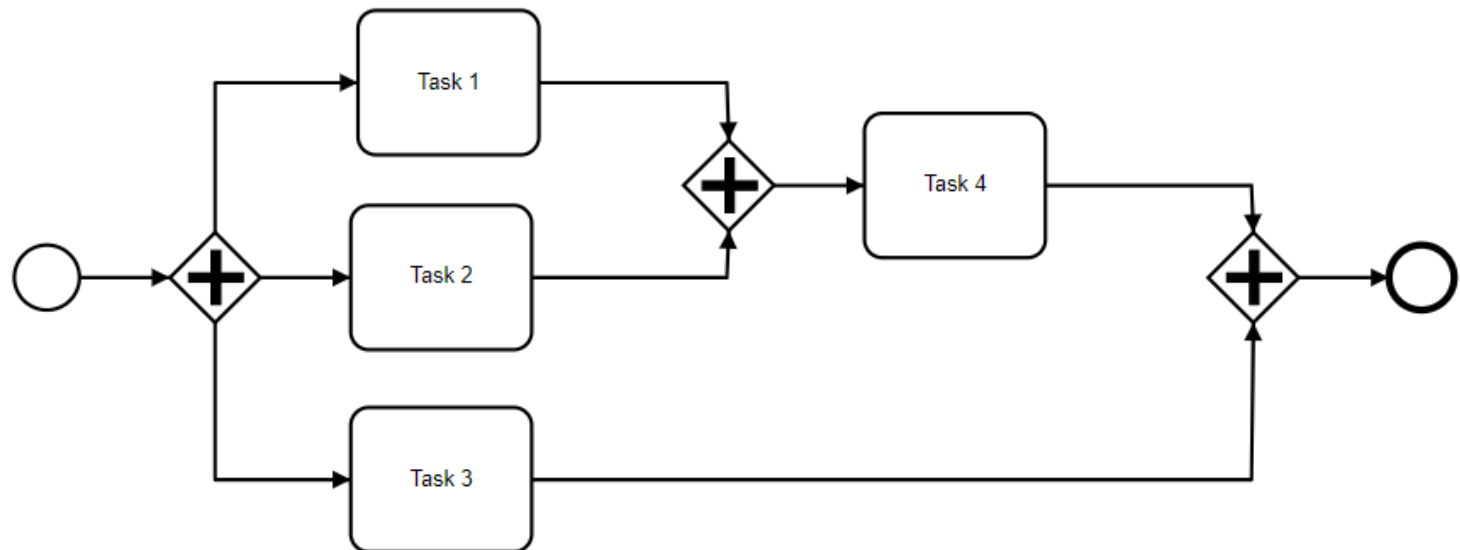


- Data-based Exclusive Gateway (XOR)
  - Model decisions based on data.
- Conditional and Default Sequence Flows
  - Concurrency and decisions without Gateways.
- Parallel Gateway
  - Model fork / join concurrency.
- Inclusive Gateway
  - Model conditional fork / join concurrency.
- Event-based Gateway
  - Model decisions based on events.

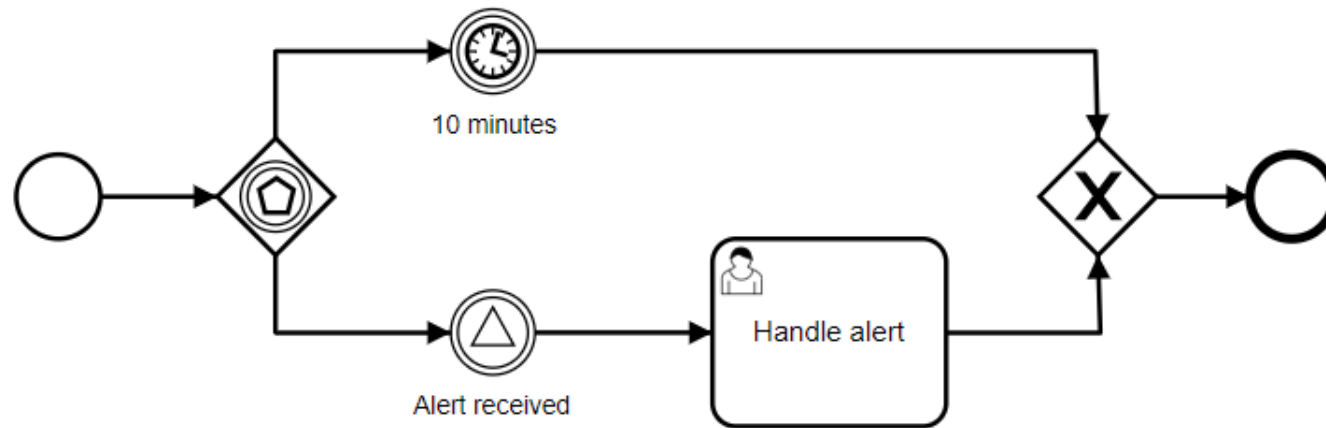
# Conditional and Default Sequence Flows



# Parallel Gateway



# Event-based Gateway



# Spring Boot Query History

- <http://localhost:7070/rest/history/process-instance>
- <http://localhost:7070/rest/history/process-instance?finishedAfter=2013-01-01T00:00:00.000+0200&finishedBefore=2013-04-01T23:59:59.000+0200&executedActivityAfter=2013-03-23T13:42:44.000+0200>

- Decision Model and Notation (DMN) is a standard for Business Decision Management.
- Currently the Camunda DMN engine partially supports DMN 1.1, including Decision Tables, Decision Literal Expressions, Decision Requirements Graphs and the Friendly Enough Expression Language (FEEL).

# DMN Decision Table

Decision Name & Id

Hit Policy

Input Expression

Input Type Definition

Output Name

Output Type Definition

Hide details

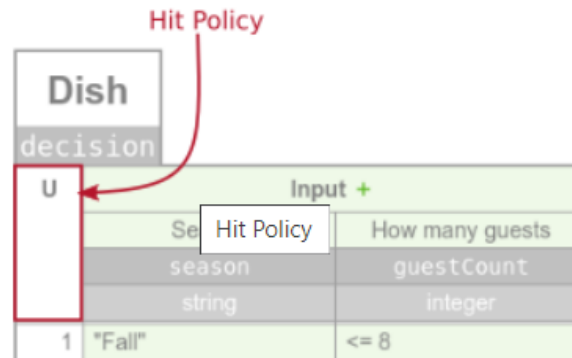
U	Input +		Output +	Annotation
	Season	How many guests	Dish	
	season	guestCount	desiredDish	
	string	integer	string	
1	"Fall"	<= 8	"Spareribs"	-
2	"Winter"	<= 8	"Roastbeef"	-
3	"Spring"	<= 4	"Dry Aged Gourmet Steak"	-
4	"Spring"	[5..8]	"Steak"	Save money
5	"Fall", "Winter", "Spring"	> 8	"Stew"	Less effort
6	"Summer"	-	"Light Salad ad nice Steak"	Hey, why not?
+	-	-	-	-

Input Entry (Condition)

Rule

Output Entry (Conclusion)

# DMN Hit Policy



A decision table has a hit policy that specifies what the results of the evaluation of a decision table consist of.

The hit policy is set in the `hitPolicy` attribute on the `decisionTable` XML element. If no hit policy is set, then the default hit policy `UNIQUE` is used.

```
<definitions xmlns="http://www.omg.org/spec/DMN/20151101/dmn.xsd" id="definitions" name="defin:
  <decision id="dish" name="Dish">
    <decisionTable id="decisionTable" hitPolicy="RULE ORDER">
      <!-- .. -->
    </decisionTable>
  </decision>
</definitions>
```



Visual representation	XML representation
U	UNIQUE
A	ANY
F	FIRST
R	RULE ORDER
C	COLLECT

## Unique Hit Policy

Only a single rule can be satisfied. The decision table result contains the output entries of the satisfied rule.

If more than one rule is satisfied, the Unique hit policy is violated.

See the following decision table.

Dish		Hit Policy: UNIQUE		Output +	
U	input	Season	Dish	desiredDish	string
1	"Fall"		"Spareribs"		
2	"Winter"		"Roastbeef"		
3	"Spring"		"Steak"		
4	"Summer"		"Light Salad and a nice Steak"		
+	-	-	-	-	-


Depending on the current season the dish should be chosen. Only one dish can be chosen, since only one season can exist at the same time.

## Any Hit Policy

Multiple rules can be satisfied. However, all satisfied rules must generate the same output. The decision table result contains only the output of one of the satisfied rules.

If multiple rules are satisfied which generate different outputs, the hit policy is violated.

See the following example:

Leave apply			
le			
Hit Policy: ANY 		Output +	
A	input		result
	Vacation Days	State	applicationResult
	vacationDays	state	string
	integer	string	
1	0	-	refused
2	> 0	probation period	refused
3	> 0	no probation period	applied
+	-	-	-

This is a decision table for the leave application. If the applier has no vacation days left or is currently in the probation period, the application will be refused. Otherwise the application is applied.

## First Hit Policy

Multiple rules can be satisfied. The decision table result contains only the output of the first satisfied rule.

Advertisement		
ad		
F	Hit Policy: FIRST	Output +
	Age	Advertised Objects
	age	advertisedObjects
	integer	string
1	> 18	Cars
2	> 12	Videogames
3	-	Toys
+	-	-

Hit Policy First

See the above decision table for advertisement. Regarding the current age of the user, which advertisement should be shown is decided. For example, the user is 19 years old. All the rules will match, but since the hit policy is set to first only, the advertisement for Cars is used.

## [Rule Order Hit Policy](#)

Multiple rules can be satisfied. The decision table result contains the output of all satisfied rules in the order of the rules in the decision table.

Advertisement			Hit Policy Rule Order
R	Hit Policy: <b>RULE ORDER</b> ✓		Output +
	Age		Advertised Objects
	age		advertisedObjects
	integer		string
1	> 18		Cars
2	> 12		Videogames
3	-		Toys
+	-		-

Again, see the advertisement example with the rule order policy. Say we have a user at the age of 19 again. All rules are satisfied so all outputs are given, ordered by the rule ordering. It can perhaps be used to indicate the priority of the displayed advertisements.

## Collect Hit Policy

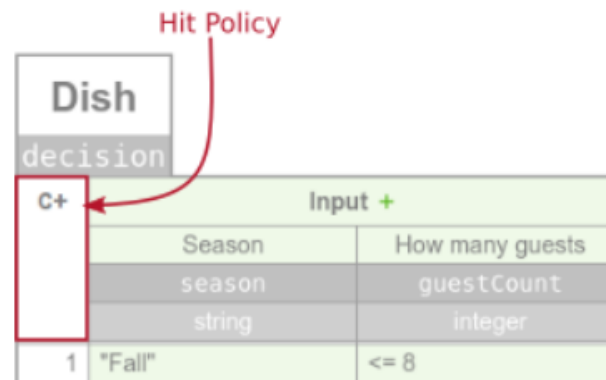
Multiple rules can be satisfied. The decision table result contains the output of all satisfied rules in an arbitrary order as a list.

Advertisement		
ad		
C	Hit Policy:	COLLECT ✓
	Collect Operator:	LIST ▼
	Output +	
	Advertised Objects	
	age	advertisedObjects
	integer	string
1	> 18	Cars
2	> 12	Videogames
3	-	Toys
+	-	-

Hit Policy Collect

With this hit policy, the output list has no ordering. So the advertisement will be arbitrary if, for example, the age is 19.

## Aggregators for Collect Hit Policy



In the visual representation of the decision table the aggregator is specified by a marker after the hit policy. The following aggregators are supported by the Camunda DMN engine:

Visual representation	XML representation	Result of the aggregation
+	SUM	the sum of all output values
<	MIN	the smallest value of all output values
>	MAX	the largest value of all output values
#	COUNT	the number of output values

## [SUM aggregator](#)

The SUM aggregator sums up all outputs from the satisfied rules.

Salary			
dec			
C+	Hit Policy:	COLLECT ✓	Output +
	Collect Operator:	SUM ▼	Bonus
	year	bonus	
	string	integer	
1	> 1	100	
2	> 2	200	
3	> 3	300	
4	> 5		
		Hit Policy Collect SUM	
+		-	-

The showed decision table can be used to sum up the salary bonus for an employee. For example, the employee has been working in the company for 3.5 years. So the first, second and third rule will match and the result of the decision table is 600, since the output is summed up.



## MIN aggregator

The MIN aggregator can be used to return the smallest output value of all satisfied rules. See the following example of a car insurance. After years without a car crash the insurance fee will be reduced.

Car insurance		
C<	Hit Policy: COLLECT ✓	Output +
	Collect Operator: MIN ▼	Insurance fee
	year	fee
	integer	double
1	-	205.43
2	> 2	150.21
3	> 3	98.83
4	> 4	64.32
+	-	-

For example, if the input for the decision table is 3.5 years, the result will be 98.83, since the first three rules match but the third rule has the minimal output.

## MAX aggregator



The MAX aggregator can be used to return the largest output value of all satisfied rules.

Pocket Money		
C>	Hit Policy:	COLLECT ✓
	Collect Operator:	MAX ▼
	age	amount
	integer	integer
1	> 5	2
2	> 8	5
3	> 14	20
4	> 16	50
+	-	-

This decision table represents the decision for the amount of pocket money for a child. Depending of the age, the amount grows. For example, an input of 9 will satisfy the first and second rules. The output of the second rule is larger then the output of the first rule, so the output will be 5. A child at the age of 9 will get 5 as pocket money.

## [🔗](#) COUNT aggregator

The COUNT aggregator can be used to return the count of satisfied rules.

Salary		
decision table		
C#	Hit Policy:	COLLECT 
	Collect Operator:	COUNT 
		Output +
		Bonus
	year	bonus
	string	integer
1	> 1	100
2	> 2	200
3	> 3	300
4	> 5	500
+	-	-

For example, see the salary bonus decision table again, this time with the COUNT aggregator. With an input of 4, the first three rules will be satisfied. Therefore, the result from the decision table will be 3, which means that after 4 years the result of the decision table is 3 salary bonuses.

# DMN Decision Literal Expression

Decision Name & Id

Literal Expression

Exit Advanced Mode Show DRD

**Season**  
season

```
calendar.getSeason(date)
```

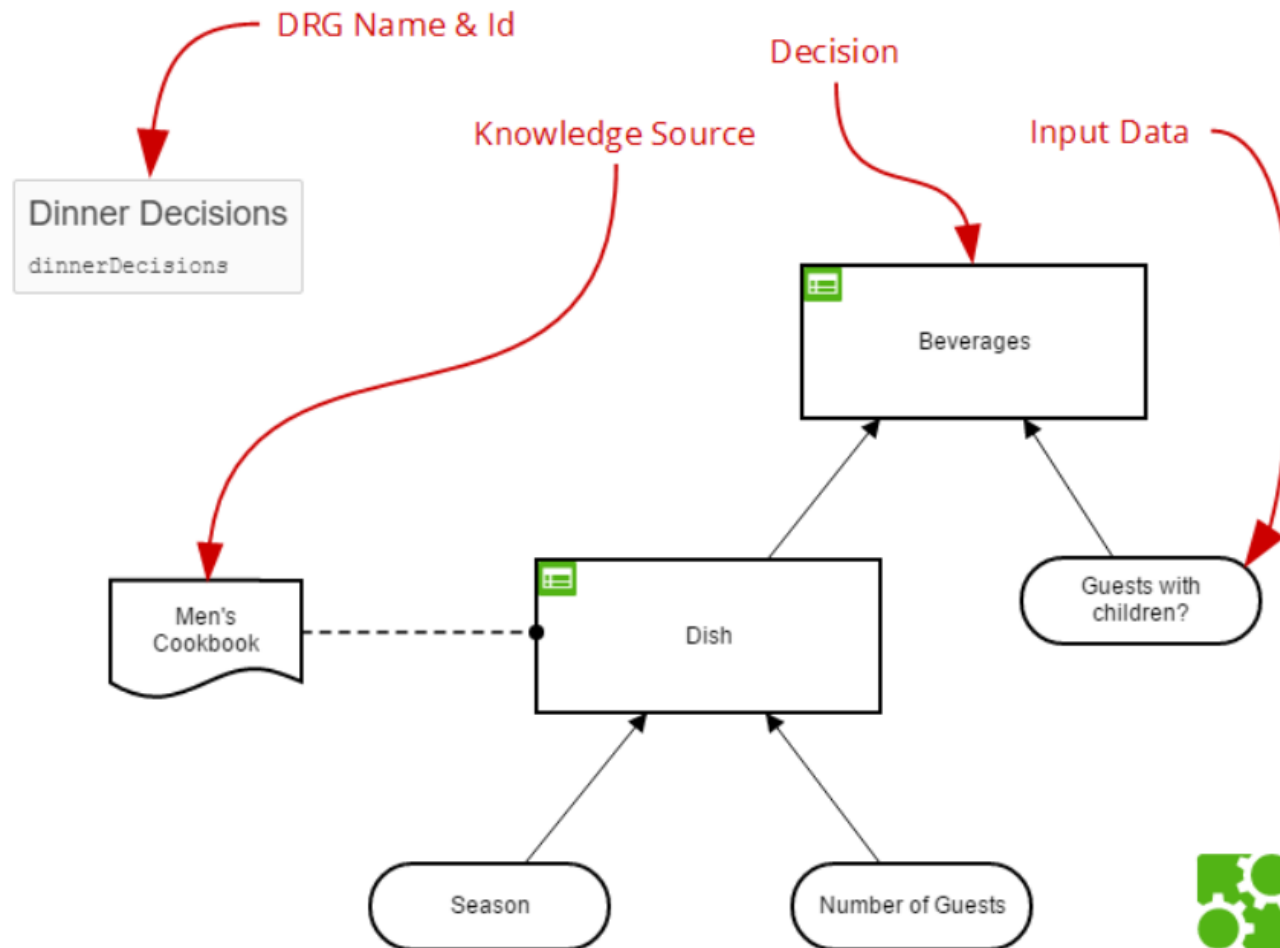
Decision Literal Expression

Variable Name:

Variable Type:

Expression Language:

# Decision Requirements Graph



# Friendly Enough Expression Language (FEEL)



- Decision Model and Notation (DMN) defines a Friendly Enough Expression Language (FEEL).
- It can be used to evaluate expressions in a decision table.
- The Camunda DMN engine only supports FEEL for input entries of a decision table.
- This corresponds to FEEL simple unary tests.

## FEEL Data Types

The Camunda DMN engine supports the following FEEL data types.

### String

Input +
Typed Input
input
string
"Fall"

FEEL supports Strings. They must be encapsulated in double quotes. They support only the equal [comparison](#) operator.

### Numeric Types

Input +
Typed Input
input

# Friendly Enough Expression Language (FEEL)



## Boolean

Input +
Typed Input
input
boolean
true

FEEL supports the boolean value `true` and `false`. The boolean type only supports the [equal comparison](#) operator.

## Date

Input +
Typed Input
input
date
date and time("2015-11-30T12:00:00")

FEEL supports date types. In the Camunda DMN engine the following date types are available:

- date and time

To create a date and time value, the function `date` and `time` has to be used with a single String parameter. The parameter specifies the date and time in the format `yyyy-MM-dd'T'HH:mm:ss`.

Date types support all [comparison](#) operators and [ranges](#).



## Comparison

FEEL simple unary tests support the following comparison operators. Please note that the equals operator is empty and *not* =. Also, a non equal operator such as != does *not* exist. To express this, [negation](#) has to be used.

Name	Operator	Example	Description
Equal		"Steak"	Test that the input value is equal to the given value.
Less	<	< 10	Test that the input value is less than the given value.
Less or Equal	<=	<= 10	Test that the input value is less than or equal to the given value.
Greater	>	> 10	Test that the input value is greater than the given value.
Greater or Equal	>=	>= 10	Test that the input value is greater than or equal to the given value.

# FEEL Language Elements

## Range

Some [FEEL data types](#), such as numeric types and date types, can be tested against a range of values. These ranges consist of a start value and an end value. The range specifies if the start and end value is included in the range.

Start	End	Example	Description
include	include	[1..10]	Test that the input value is greater than or equal to the start value and less than or equal to the end value.
exclude	include	]1..10] or (1..10]	Test that the input value is greater than the start value and less than or equal to the end value.
include	exclude	[1..10[ or [1..10)	Test that the input value is greater than or equal to the start value and less than the end value.
exclude	exclude	]1..10[ or (1..10)	Test that the input value is greater than the start value and less than the end value.

## Disjunction

A FEEL simple unary test can be specified as conjunction of expressions. These expressions have to either have [comparisons](#) or [ranges](#). The test is true if at least one of conjunct expressions is true.

Examples:

- `3,5,7`: Test if the input is either 3, 5 or 7
- `<2,>10`: Test if the input is either less than 2 or greater than 10
- `10,[20..30]`: Test if the input is either 10 or between 20 and 30
- `"Spareribs","Steak","Stew"`: Test if the input is either the String Spareribs, Steak or Stew
- `date and time("2015-11-30T12:00:00"),date and time("2015-12-01T12:00:00")`: Test if the input is either the date November 30th, 2015 at 12:00:00 o'clock or December 1st, 2015 at 12:00:00 o'clock
- `>customer.age,>21`: Test if the input is either greater than the `age` property of the variable `customer` or greater than 21

## Negation

A FEEL simple unary test can be negated with the `not` function. This means if the containing expression returns `true`, the test will return `false`. Please *note* that only one negation as first operator is allowed but it can contain a [disjunction](#).

Examples:

- `not("Steak")`: Test if the input is not the String Steak
- `not(>10)`: Test if the input is not greater than 10, which means it is less than or equal to 10
- `not(3,5,7)`: Test if the input is neither 3, 5 nor 7
- `not([20..30])`: Test if the input is not between 20 and 30

## Qualified Names

FEEL simple unary tests can access variables and object properties by qualified names.

Examples:

- `x`: Test if the input is equal to the variable `x`
- `>= x`: Test if the input is greater than or equal to the variable `x`
- `< customer.age`: Test if the input is less than the `age` property of the variable `customer`

## Date Functions

FEEL simple unary tests provide functions to create [date types](#). The Camunda DMN engine supports the following date functions:

- `date and time("...")`: Creates a date and time value from a String with the format `yyyy-MM-dd'T'HH:mm:ss`

Examples:

- `date and time("2015-11-30T12:00:00")`: Test if the input is the date November 30th, 2015 at 12:00:00 o'clock
- `[date and time("2015-11-30T12:00:00")..date and time("2015-12-01T12:00:00")]`: Test if the input is between the date November 30th, 2015 at 12:00:00 o'clock and December 1st, 2015 at 12:00:00 o'clock



# Camunda BPM Platform Docker Images

---

- `docker pull camunda/camunda-bpm-platform:latest`
- `docker run -d --name camunda -p 8080:8080 camunda/camunda-bpm-platform:latest`
- # open browser with url:  
<http://192.168.99.101:8080/camunda-welcome/index.html>

# Questions

