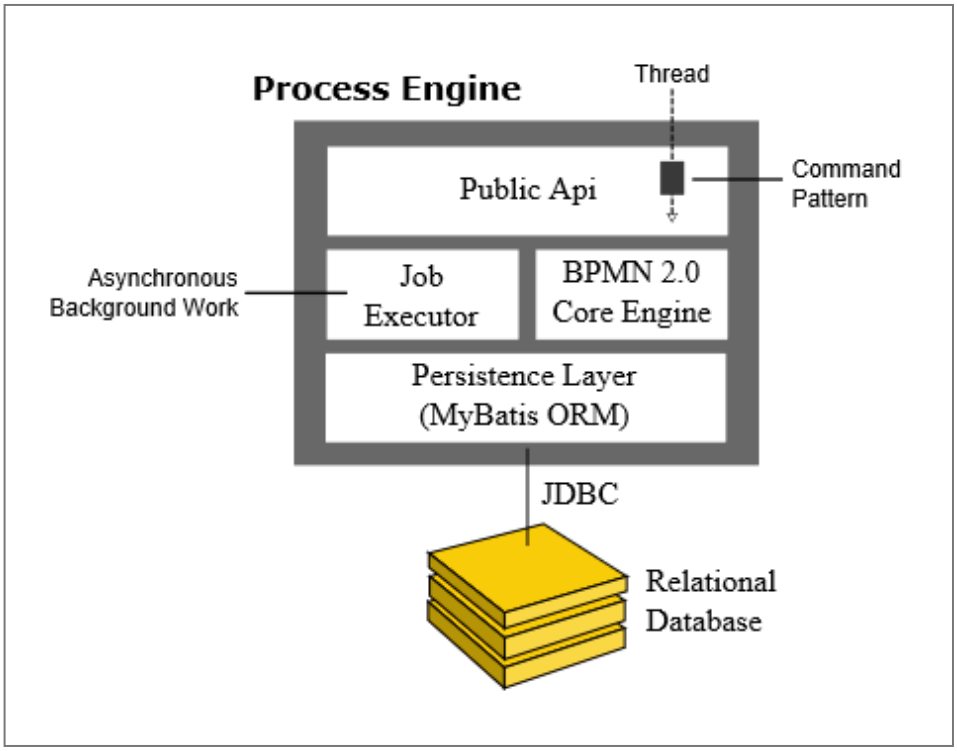
Architecture Overview

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

Camunda can be used both as a standalone process engine server or embedded inside custom Java applications. The embeddability requirement is at the heart of many architectural decisions within Camunda. For instance, we work hard to make the process engine component a lightweight component with as little dependencies on third-party libraries as possible. Furthermore, the embeddability motivates programming model choices such as the capabilities of the process engine to participate in Spring Managed or JTA transactions and the threading model.

Process Engine Architecture



- 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.
- **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 (providing the implementation for BPMN 2.0 constructs such as Gateways or Service Tasks).
- 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. We use the MyBatis mapping engine for object relational mapping.

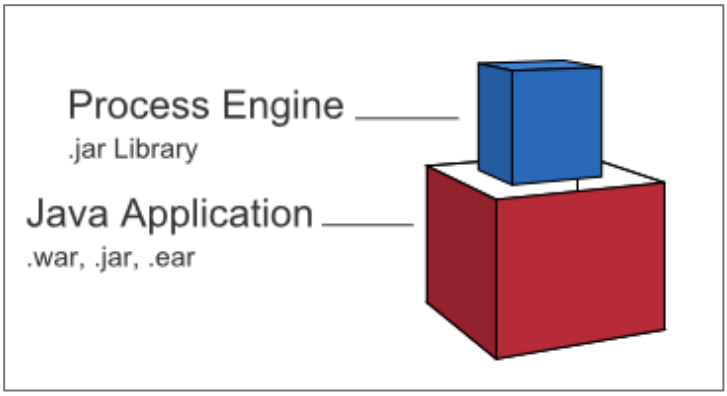
Required Third-Party Libraries

See the section on third-party libraries.

Camunda Architecture

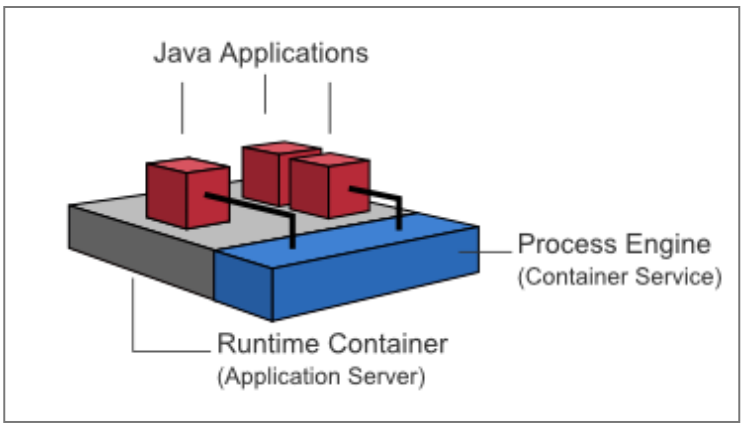
Camunda is a flexible framework which can be deployed in different scenarios. This section provides an overview of the most common deployment scenarios.

Embedded Process Engine



In this case, 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.

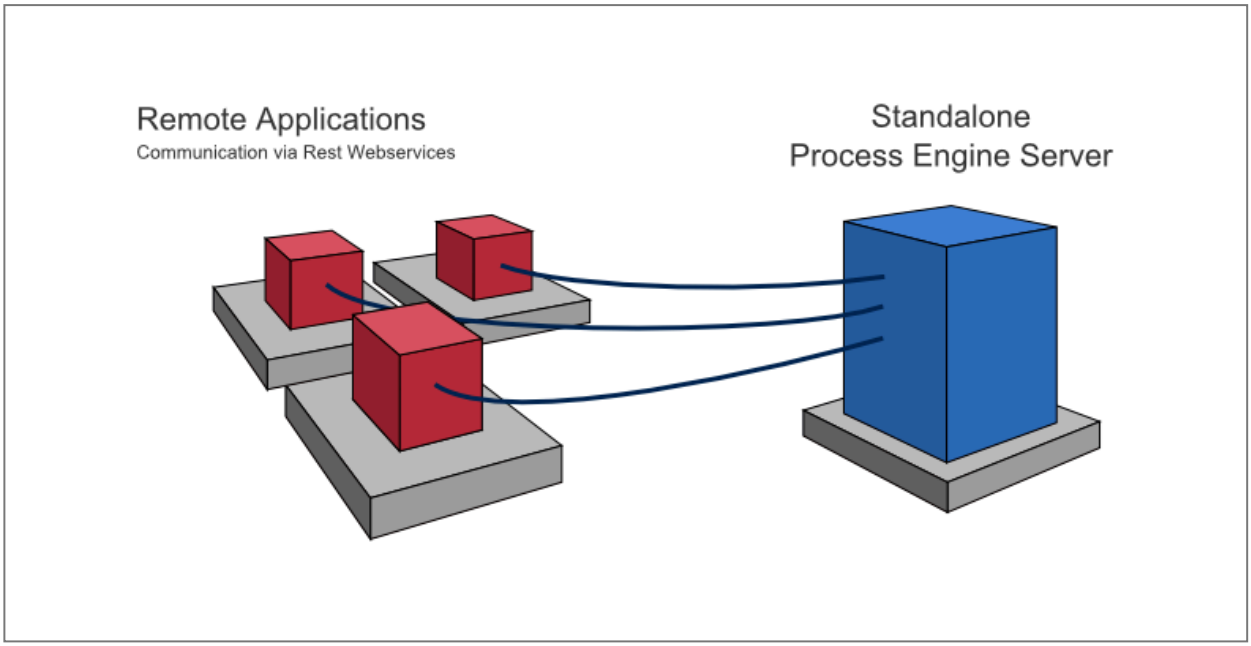
Shared, Container-Managed Process Engine



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 concept can be compared to a JMS

Message Queue which is provided by the runtime and can be used by all applications. There is a one-to-one mapping between process deployments and applications: the process engine keeps track of the process definitions deployed by an application and delegates execution to the application in question.

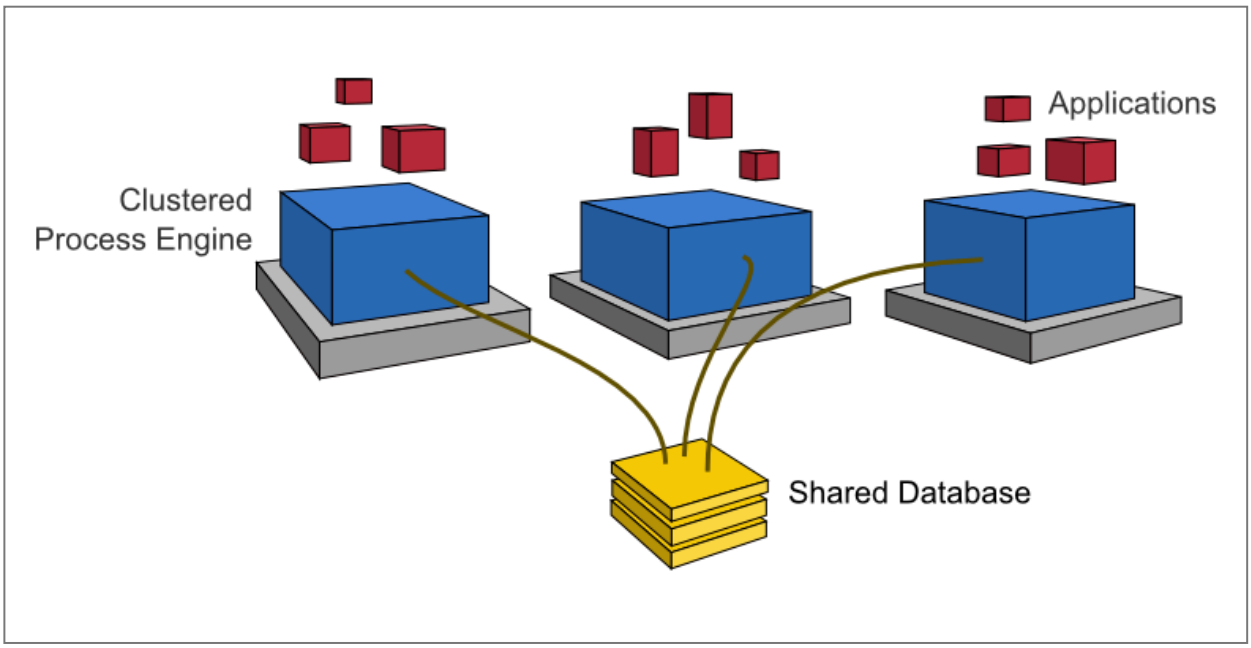
Standalone (Remote) Process Engine Server



In this case, 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 Webservices or JMS are possible but need to be implemented by users.

Clustering Model

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. This model is very simple and easy to understand and imposes limited restrictions when it comes to deploying a cluster installation. 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).

Session State in a Clustered Environment

Camunda 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, and session replication would need to be provided by the host application server.

In a clustered setup, if users are going to login to the web applications, an extra step will need to be taken to ensure that users aren't asked to login multiple times. Two options exist:

1. "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.
2. 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.

If neither of the above approaches are implemented in a cluster setup, connections to multiple nodes - intentionally or via a load-balancing solution - will result in multiple login requests.

The Job Executor in a Clustered Environment

The process engine job executor is also clustered and runs on each node. This way, there is no single point of failure as far as the process engine is concerned. The job executor can run in both homogeneous and heterogeneous clusters.

> Time zones
>
> The are some limitations on time zone usage in a cluster.

Multi-Tenancy Models

To serve multiple, independent parties with one Camunda installation, the process engine supports multi-tenancy. The following multi tenancy models are supported:

- Table-level data separation by using different database schemas or databases
- Row-level data separation by using a tenant marker

Users should choose the model which fits their data separation needs. Camunda's APIs provide access to processes and related data specific to each tenant. More details can be found in the multi-tenancy section.

Web Application Architecture

The Camunda web applications are based on a RESTful architecture.

Frameworks used:

- JAX-RS based Rest API
- AngularJS
- RequireJS
- jQuery
- Twitter Bootstrap

Additional custom frameworks developed by Camunda hackers:

- camunda-bpmn.js: Camunda BPMN 2.0 JavaScript libraries
- ngDefine: integration of AngularJS into RequireJS powered applications
- angular-data-depend: toolkit for implementing complex, data heavy AngularJS applications