User Guide

Table of Contents

Taskpool Components

- Overview
- Engine Components
  - Camunda Engine Taskpool Support SpringBoot Starter
  - Camunda Engine Eventing Plugin
  - Camunda Engine Interaction Client
  - Taskpool Collector
  - Datapool Collector
- Core Components
  - Taskpool Core
  - Datapool Core
- View Components
  - Simple View
  - Mongo View
  - Taskpool Cockpit

Taskpool Examples

- Examples

Taskpool Usage Scenarios

- Scenarios

Taskpool Components

Engine Components

Engine Components are designed to be a part of process application deployment:

- Camunda Engine Taskpool Support SpringBoot Starter
- Camunda Engine Eventing Plugin
- Camunda Engine Interaction Client
- Taskpool Collector
- Datapool Collector

Core Components

Core Components are responsible for the processing of engine commands and form an event stream consumed by the view components. Depending on scenario, they can be deployed either within the process application, task list application or even completely separately.

- Taskpool Core
- Datapool Core

View Components

View Components are responsible for creation of a unified read-only projection of tasks and business data items. They are typically deployed as a part of the task list application.

- Simple View
- Mongo View
- Taskpool Cockpit

Camunda Engine Taskpool Support SpringBoot Starter

Camunda Engine Taskpool Support SpringBoot Starter

Purpose

The Camunda Engine Taskpool Support SpringBoot Starter is a convenience module providing a single module dependency to be included in the process application. It includes all process application modules and provides meaningful defaults for their options.

Configuration

In order to enable the starter, please put the following annotation on any `@Configuration` annotated class of your SpringBoot application.

```
@SpringBootApplication
@EnableProcessApplication
@EnableTaskpoolEngineSupport (1)
public class MyApplication {

  public static void main(String... args) {
    SpringApplication.run(MyApplication.class, args);
  }
}
```

1. Annotation to enable the engine support.

The `@EnableTaskpoolEngineSupport` annotation has the same effect as the following block of annotations:

```
@EnableCamundaSpringEventing
@EnableCamundaEngineClient
@EnableTaskCollector
@EnableDataEntryCollector
public class MyApplication {
  //...
}
```

Camunda Engine Eventing Plugin

Camunda Engine Eventing Plugin

Note Starting from Camunda BPM SpringBoot version 3.3.0, the functionality of the eventing plugin has been contributed to the Camunda BPM SpringBoot and the component is not required anymore.

Purpose

The purpose of this component is to emit all changes happening in Camunda process engine as Spring events. In doing so, it registers execution listeners, task listeners and history event listener by providing a special Camunda Engine Core plugin and executes the delegation to the 'Taskpool Collector' and 'Datapool Collector'.

Configuration options

The eventing plugin is controlled by the properties prefixed by `camunda.taskpool.engine-eventing`. To enable eventing (`false` by default). by setting `camunda.taskpool.engine-eventing.enabled` property. The three boolean properties `camunda.taskpool.engine-eventing.task-eventing`, `camunda.taskpool.engine-eventing.execution-eventing` and `camunda.taskpool.engine-eventing.historic-eventing` (all enabled by default) control the three types of listeners enabled by the plugin.

Tip Taskpool requires at least Task Listener and History eventing to operate properly.

In addition to the eventing, the process application MUST run with enabled history.

Tip We recommend to run in *Full History Mode* in order to get all events from Camunda BPM engine.

Camunda Engine Interaction Client

Camunda Engine Interaction Client

Purpose

Configuration options

Taskpool Collector

Taskpool Collector

Purpose

Taskpool Collector is a component usually deployed as a part of the process application (aside with Camunda BPM Engine) that is responsible for collecting Spring events fired by the Camunda Engine Eventing Plugin and creating the corresponding commands for the taskpool. In doing so, it collects and enriches data and transmits it to Taskpool Core.

In the following description, we use the term *event* and *command*. Event denotes a data entity received from Camunda BPM Engine (from delegate event listener or from history event listener) which is passed over to the Task Collector using internal Spring eventing mechanism. The Task Collector converts the series of such events into an Taskpool Engine Command - an entity carrying an intent of change inside of the taskpool core.

Features

- Collection of task events and history events
- Creation of corresponding task engine commands
- Enrichment of task engine commands with process variables
- Attachment of correlation information to task engine commands
- Transmission of task engine commands
- Provision of properties for process application

Architecture

```
+---------+      +-------------+     +---------+     +--------------+     +-----------+
|         |      |             |     |         |     |    Task      |     |           |
| Camunda +-=--->+    Event    +-=---|  Task   +-=-->+   Command    +---->+  Command  |
|  Event  |      |  Collector  |     | Command |     |   Enricher   |     |   Sender  |
|     {d} |      |             |     |     {d} |     |   Service    |     |           |
+---------+      +-------------+     +---------+     +--+-------+---+     +-----------+
                                                        :       ^
                                                        |       :
                                                        V       |
                                                     +--+-------+--+
                                                     |             |
                                                     |   Enricher  |
                                                     |             |
                                                     +-------------+
```

The Taskpool Collector consists of several components:

- Event collector receives Spring Events from `camunda-eventing-engine-plugin` and forms commands
- Enricher performs the command enrichment with payload and data correlation

- Command sender is responsible for accumulating commands and sending them to Command Gateway

Usage and configuration

In order to enable collector component, include the Maven dependency to your process application:

```
<dependency>
  <groupId>io.holunda.taskpool<groupId>
  <artifactId>camunda-bpm-taskpool-collector</artifactId>
  <version>${camunda-taskpool.version}</version>
<dependency>
```

Then activate the taskpool collector by providing the annotation on any Spring Configuration:

```
@Configuration
@EnableDataEntryCollector
class MyDataCollectorConfiguration {

}
```

Event collection

Taskpool collector registers Spring Event Listener to the following events, fired by Camunda Eventing Engine Plugin:

- `DelegateTask` events:
  - create
  - assign
  - delete
  - complete
- `HistoryEvent` events:
  - HistoricTaskInstanceEvent
  - HistoricIdentityLinkLogEvent

Task commands enrichment

Alongside with data attributes received from the Camunda BPM engine, the task engine commands can be enriched with additional business data. There are three enrichment modes available controlled by the `camunda.taskpool.collector.enricher.type` property:

- `no`: No enrichment takes place
- `process-variables`: Enrichment with process variables
- `custom`: User provides own implementation

Process variable enrichment

In particular cases, the task related data is not sufficient for the information required in task list or other user-related components. The information may be available as process variables and need to be attached to the task in the taskpool. This is where *Process Variable Task Enricher* can be used. For this purpose, set the property `camunda.taskpool.collector.enricher.type` to `process-variables` and the enricher will put all process variables into the task payload (defaults to a empty `EXCLUDE` filter).

You can control what variables will be put into task command payload by providing the Process Variables Filter. The `ProcessVariablesFilter` is a Spring bean holding a list individual `ProcessVariableFilter` - one per process definition key.

A `ProcessVariableFilter` can be of the following type:

- `INCLUDE`: task-level include filter, denoting a list of variables to be added for task.
- `EXCLUDE`: task-level exclude filter, denoting a list of variables to be ignored. All other variables are included.
- `PROCESS_INCLUDE`: process-level include filter, denoting a list of variables to be added for all tasks.
- `PROCESS_EXCLUDE`: process-level exclude filter, denoting a list of variables to be ignored for all tasks.

Here is an example, how the process variable filter can configure the enrichment:

```
@Configuration
public class MyTaskCollectorConfiguration {

  @Bean
  public ProcessVariablesFilter myProcessVariablesFilter() {

    return new ProcessVariablesFilter(
      // define a applyFilter for every process
      new ProcessVariableFilter[]{
        // for every process definition
        new ProcessVariableFilter(
          ProcessApproveRequest.KEY,
          // filter type
          FilterType.INCLUDE,
          ImmutableMap.<String, List<String>>builder()
            // define a applyFilter for every task
            .put(ProcessApproveRequest.Elements.APPROVE_REQUEST, Lists.newArrayList(
              ProcessApproveRequest.Variables.REQUEST_ID,
              ProcessApproveRequest.Variables.ORIGINATOR)
            )
            // and again
            .put(ProcessApproveRequest.Elements.AMEND_REQUEST, Lists.newArrayList(
              ProcessApproveRequest.Variables.REQUEST_ID,
              ProcessApproveRequest.Variables.COMMENT,
              ProcessApproveRequest.Variables.APPLICANT)
            ).build(),
          Collections.emptyList()
        )
      }, Collections.emptyMap()
    );
  }

}
```

Tip If you want to implement a custom enrichment, please provide your own implementation of the interface `VariablesEnricher` (register a Spring Component of the type) and set the property `camunda.taskpool.collector.enricher.type` to `custom`.

Data Correlation

Apart from task payload attached by the enricher, the so-called *Correlation* with data entries can be configured. The idea is to attach one or several references (that is `entryType` and `entryId`) to business data entry(ies) to a task. In a view projection this correlations can be resolved and the information from business data events can be shown together with task information.

The correlation to data events can be configured by providing a `ProcessVariablesCorrelator`. Here is an example how this can be done:

```
@Bean
open fun processVariablesCorrelator() = ProcessVariablesCorrelator(

  ProcessVariableCorrelation(ProcessApproveRequest.KEY, (1)
    mapOf(
      ProcessApproveRequest.Elements.APPROVE_REQUEST to mapOf( (2)
        ProcessApproveRequest.Variables.REQUEST_ID to BusinessDataEntry.REQUEST
      )
    ),
    mapOf(ProcessApproveRequest.Variables.REQUEST_ID to BusinessDataEntry.REQUEST) (3)
  )
)
```

1. define correlation for every process
2. define a correlation for every task needed
3. define a correlation globally (for the whole process)

The process variable correlator holds a list of process variable correlations - one for every process definition key. Every `ProcessVariableCorrelation` configures global (that is for every task) or task correlation (for particular task definition key) by providing a correlation map. A correlation map is keyed by the Camunda Process Variable Name and holds business data Entry Type as value.

Here is an example. Imagine the process instance is storing the id of an approval request in a process variable called `varRequestId`. The system responsible for storing approval requests fires data entry events supplying the data and using the entry type `approvalRequest` and the id of the request as `entryId`. In order to create a correlation in task `task_approve_request` of the `process_approval_process` we would provide the following configuration of the correlator:

```
@Bean
open fun processVariablesCorrelator() = ProcessVariablesCorrelator(

  ProcessVariableCorrelation("process_approval_process",
    mapOf(
      "task_approve_request" to mapOf(
        "varRequestId" to "approvalRequest"
      )
    )
  )
)
```

If the process instance now contains the approval request id `"4711"` in the process variable `varRequestId` and the process reaches the task `task_approve_request`, the task will get the following correlation created (here written in JSON):

```
"correlations": [
  { "entryType": "approvalRequest", "entryId": "4711" }
]
```

Command transmission

In order to control sending of commands to command gateway, the command sender activation property `camunda.taskpool.collector.sender.enabled` (default is `true`) is available. If disabled, the command sender will log any command instead of sending it to the command gateway.

In addition you can control by the property `camunda.taskpool.collector.sender.type` if you want to use the default command sender or provide your own implementation. The default provided command sender (type: `tx`) is collects all task commands during one transaction, group them by task id and accumulates by creating one command reflecting the intent of the task operation. It uses Axon Command Bus (encapsulated by the `AxonCommandListGateway`.

Tip If you want to implement a custom command sending, please provide your own implementation of the interface `CommandSender` (register a Spring Component of the type) and set the property `camunda.taskpool.collector.sender.type` to `custom`.

The Spring event listeners receiving events from the Camunda Engine plugin are called before the engine commits the transaction. Since all processing inside collector and enricher is performed synchronous, the sender must waits until transaction to be successfully committed before sending any commands to the Command Gateway. Otherwise, on any error the transaction would be rolled back and the command would create an inconsistency between the taskpool and the engine.

Depending on your deployment scenario, you may want to control the exact point in time when the commands are send to Command Bus. The property `camunda.taskpool.collector.sender.send-within-transaction` is designed to influence this. If set to `true`, the commands are sent *before* the process engine transaction is committed, otherwise commands are sent *after* the process engine transaction is committed.

Warning Never send commands over remote messaging before the transaction is committed, since you may produce unexpected results if Camunda fails to commit the transaction.

Datapool Collector

Datapool Collector

Purpose

Datapool collector is a component usually deployed as a part of the process application (but not necessary) that is responsible for collecting the Business Data Events fired by the application in order to allow for creation of a business data projection. In doing so, it collects and transmits it to Datapool Core.

Features

- Provides an API to submit arbitrary changes of business entities
- Provides an API to track changes (aka. Audit Log)
- Authorization on business entries
- Transmission of business entries commands

Usage and configuration

```
    <dependency>
      <groupId>io.holunda.taskpool</groupId>
      <artifactId>camunda-bpm-datapool-collector</artifactId>
      <version>${camunda-taskpool.version}</version>
    </dependency>
```

Then activate the datapool collector by providing the annotation on any Spring Configuration:

```
@Configuration
@EnableDataEntryCollector
class MyDataEntryCollectorConfiguration {

}
```

Taskpool Core

Taskpool Core

Purpose

Configuration options

Description

Datapool Core

Datapool Core

Purpose

Configuration options

In-Memory View

In-Memory View

Purpose

Configuration options

Mongo View

Mongo View

Purpose

The Mongo View is component responsible for creating read-projections of tasks and business data entries. It implements the Taskpool and Datapool View API and persists the projection as document collections in a Mongo database.

Features

- stores JSON document representation of enriched tasks, process definitions and business data entries
- provides single query API
- provides subscription query API (reactive)
- switchable subscription query API (AxonServer or MongoDB ChangeStream)

Configuration options

Depending on your setup, you might want to use Axon Query Bus for subscription queries or not. MongoDB provides a change stream if run in a replication set. Using the property `camunda.taskpool.view.mongo.change-tracking-mode` you can control, whether you use subscription query based on Axon Query Bus (value `EVENT_HANDLER`, default) or based on Mongo Change Stream (value `CHANGE_STREAM`). If you are not interested in publication of any subscription queries you might choose to disable it by setting the option to value `NONE`.

Collections

The Mongo View uses several collections to store the results. These are:

- data-entries: collection for business data entries
- processes: collection for process definitions
- tasks: collection for user tasks
- tracking-tokens: collection for Axon Tracking Tokens

Data Entries Collection

The data entries collection stores the business data entries in a uniform Datapool format. Here is an example:

```
{
    "_id" : "io.holunda.camunda.taskpool.example.ApprovalRequest#2db47ced-83d4-4c74-a644
    "entryType" : "io.holunda.camunda.taskpool.example.ApprovalRequest",
    "payload" : {
        "amount" : "900.00",
        "subject" : "Advanced training",
        "currency" : "EUR",
        "id" : "2db47ced-83d4-4c74-a644-44dd738935f8",
        "applicant" : "hulk"
    },
    "correlations" : {},
    "type" : "Approval Request",
    "name" : "AR 2db47ced-83d4-4c74-a644-44dd738935f8",
    "applicationName" : "example-process-approval",
    "description" : "Advanced training",
    "state" : "Submitted",
    "statusType" : "IN_PROGRESS",
    "authorizedUsers" : [
        "gonzo",
        "hulk"
    ],
    "authorizedGroups" : [],
    "protocol" : [
        {
            "time" : ISODate("2019-08-21T09:12:54.779Z"),
            "statusType" : "PRELIMINARY",
            "state" : "Draft",
            "username" : "gonzo",
            "logMessage" : "Draft created.",
            "logDetails" : "Request draft on behalf of hulk created."
        },
        {
            "time" : ISODate("2019-08-21T09:12:55.060Z"),
            "statusType" : "IN_PROGRESS",
            "state" : "Submitted",
            "username" : "gonzo",
            "logMessage" : "New approval request submitted."
        }
    ]
}
```

Tasks Collections

Tasks are stored in the following format (an example):

```
{
    "_id" : "dc1abe54-c3f3-11e9-86e8-4ab58cfe8f17",
    "sourceReference" : {
        "_id" : "dc173bca-c3f3-11e9-86e8-4ab58cfe8f17",
        "executionId" : "dc1a9742-c3f3-11e9-86e8-4ab58cfe8f17",
        "definitionId" : "process_approve_request:1:91f2ff26-a64b-11e9-b117-3e6d125b91e2
        "definitionKey" : "process_approve_request",
        "name" : "Request Approval",
        "applicationName" : "example-process-approval",
        "_class" : "process"
    },
    "taskDefinitionKey" : "user_approve_request",
    "payload" : {
        "request" : "2db47ced-83d4-4c74-a644-44dd738935f8",
        "originator" : "gonzo"
    },
    "correlations" : {
        "io:holunda:camunda:taskpool:example:ApprovalRequest" : "2db47ced-83d4-4c74-a644
        "io:holunda:camunda:taskpool:example:User" : "gonzo"
    },
    "dataEntriesRefs" : [
        "io.holunda.camunda.taskpool.example.ApprovalRequest#2db47ced-83d4-4c74-a644-44d
        "io.holunda.camunda.taskpool.example.User#gonzo"
    ],
    "businessKey" : "2db47ced-83d4-4c74-a644-44dd738935f8",
    "name" : "Approve Request",
    "description" : "Please approve request 2db47ced-83d4-4c74-a644-44dd738935f8 from go
    "formKey" : "approve-request",
    "priority" : 23,
    "createTime" : ISODate("2019-08-21T09:12:54.872Z"),
    "candidateUsers" : [
        "fozzy",
        "gonzo"
    ],
    "candidateGroups" : [],
    "dueDate" : ISODate("2019-06-26T07:55:00.000Z"),
    "followUpDate" : ISODate("2023-06-26T07:55:00.000Z"),
    "deleted" : false
}
```

Process Collection

Process definition collection allows for storage of startable process definitions, deployed in a Camunda Engine. This information is in particular interesting, if you are building a process-starter component and want to react dynamically on processes deployed in your landscape.

```
{
    "_id" : "process_approve_request:1:91f2ff26-a64b-11e9-b117-3e6d125b91e2",
    "processDefinitionKey" : "process_approve_request",
    "processDefinitionVersion" : 1,
    "applicationName" : "example-process-approval",
    "processName" : "Request Approval",
    "processDescription" : "This is a wonderful process.",
    "formKey" : "start-approval",
    "startableFromTasklist" : true,
    "candidateStarterUsers" : [],
    "candidateStarterGroups" : [
        "muppetshow",
        "avengers"
    ]
}
```

Tracking Token Collection

The Axon Tracking Token reflects the index of the event processed by the Mongo View and is stored in the following format:

```
{
    "_id" : ObjectId("5d2b45d6a9ca33042abea23b"),
    "processorName" : "io.holunda.camunda.taskpool.view.mongo.service",
    "segment" : 0,
    "owner" : "18524@blackstar",
    "timestamp" : NumberLong(1566379093564),
    "token" : { "$binary" : "PG9yZy5heG9uZnJhbWV3b3JrLmV2ZW50aGFuZGxpbmcuR2xvYmFsU2VxdWVu
    "tokenType" : "org.axonframework.eventhandling.GlobalSequenceTrackingToken"
}
```

Taskpool Cockpit View

Taskpool Cockpit View

Purpose

Configuration options

Taskpool Examples

Taskpool Examples

Taskpool Usage Scenarios

Usage Scenarios

Depending on your requirements and infrastructure available several deployment scenarios of the components is possible.

One of the challenging issues for distribution and connecting microservices is a setup of messaging technology supporting required message exchange patterns (MEPs) for a CQRS system. Because of different semantics of commands, events and queries and additional requirements of event-sourced persistence a special implementation of command bus, event bus and event store are required.

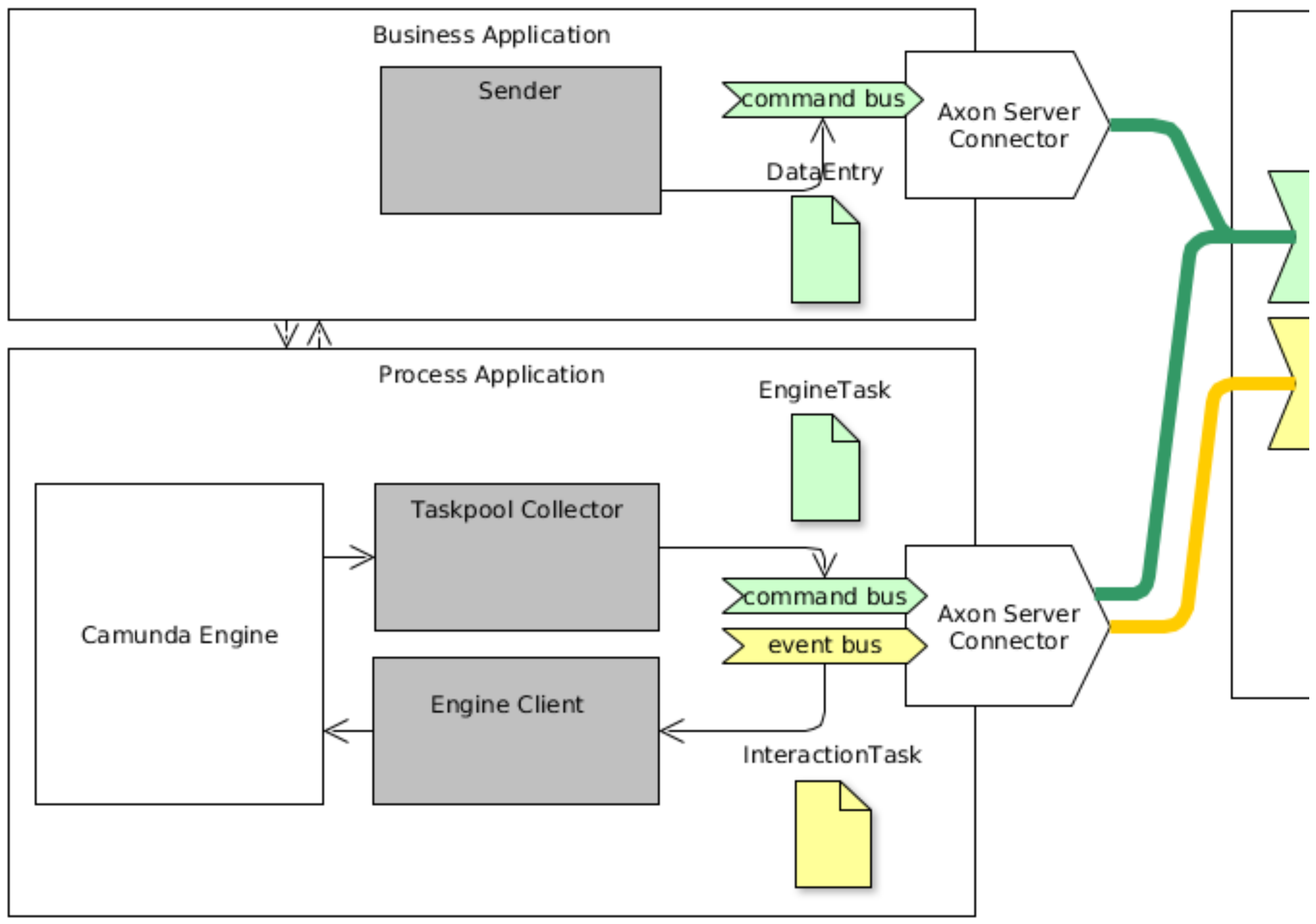
Axon Server Scenario

Axon Server provides such implementation leading to a distributed command and event-bus and a central event store. It is easy to use, easy to configure and easy to run. If you need a HA setup, you will need the enterprise license of Axon Server. Essentially, if don’t have another HA ready-to use messaging, this scenario might be your way to go.

This scenario supports:

- central task pool / data pool
- view must not have a persistent storage (can be replayed)
- no direct communication between task list and engine is required (routed via command bus)

The following diagram depicts the distribution of the components and the messaging.



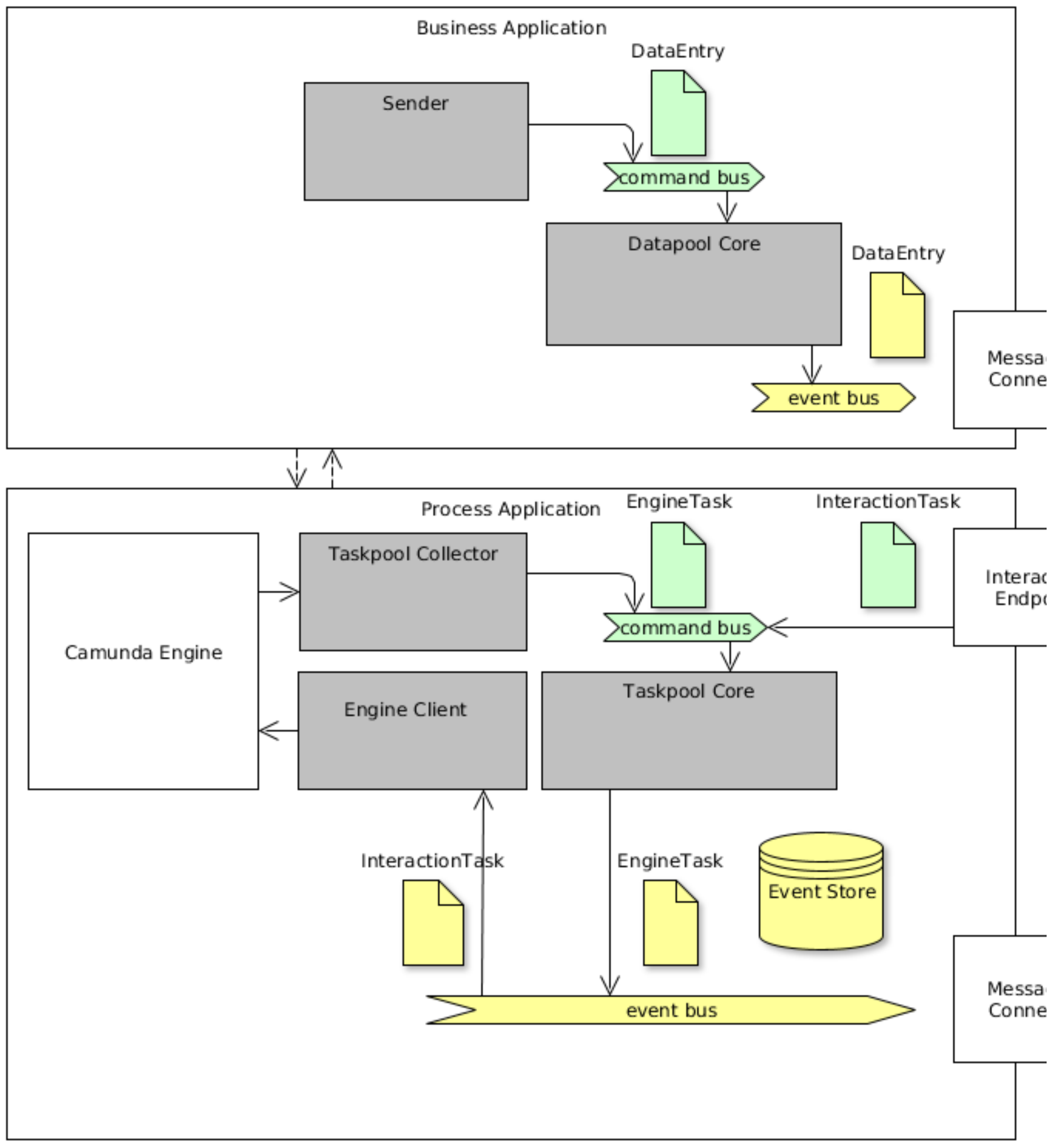
Scenario without Axon Server

If you already have another messaging at place, like Kafka or RabbitMQ, you might skip the usage of Axon Server. In doing so, you will be responsible for distribution of events and will need to surrender some features.

This scenario supports:

- distributed task pool / data pool
- view must be persistent
- direct communication between task list / engines required (addressing, routing)
- concurrent access to engines might become a problem (no unit of work guarantees)

The following diagram depicts the distribution of the components and the messaging.


Business Application
DataEntry
Sender
command bus
Datapool Core
DataEntry
event bus
Messa
Conne
Process Application
EngineTask
InteractionTask
Taskpool Collector
command bus
Interac
Endpo
Camunda Engine
Engine Client
Taskpool Core
InteractionTask
EngineTask
Event Store
event bus
Messa
Conne