User Guide

Table of Contents

Taskpool Components

Engine Components

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

- Camunda Engine Taskpool Support SpringBoot Starter
- 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.

- In-Memory View
- Mongo View

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 Interaction Client

Camunda Engine Interaction Client

Purpose

This component performs changes delivered by Camunda Interaction Events on Camunda BPM engine. The following Camunda Interaction Events are supported:

- Claim User Task
- Unclaim User Task
- Defer User Task
- Undefer User Task
- Complete User Task

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 an 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 of individual `VariableFilter` - at most one per process definition key and optionally one without process definition key (a global filter).

A `VariableFilter` can be of the following type:

- `TaskVariableFilter`:
  - `INCLUDE`: task-level include filter, denoting a list of variables to be added for the task.
  - `EXCLUDE`: task-level exclude filter, denoting a list of variables to be ignored. All other variables are included.
- `ProcessVariableFilter` with process definition key:
  - `INCLUDE`: process-level include filter, denoting a list of variables to be added for all tasks of the process.
  - `EXCLUDE`: process-level exclude filter, denoting a list of variables to be ignored for all tasks of the process.
- `ProcessVariableFilter` *without* process definition key:
  - `INCLUDE`: global include filter, denoting a list of variables to be added for all tasks of all processes for which no dedicated `ProcessVariableFilter` is defined.
  - `EXCLUDE`: global exclude filter, denoting a list of variables to be ignored for all tasks of all processes for which no dedicated `ProcessVariableFilter` is defined.

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 variable filter for every process
      new VariableFilter[]{
        // define for every process definition
        // either a TaskVariableFilter or ProcessVariableFilter
        new TaskVariableFilter(
          ProcessApproveRequest.KEY,
          // filter type
          FilterType.INCLUDE,
          ImmutableMap.<String, List<String>>builder()
            // define a variable filter for every task of the process
            .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()
        ),
        // optionally add a global filter for all processes
        // for that no individual filter was created
        new ProcessVariableFilter(FilterType.INCLUDE,
          Lists.newArrayList(CommonProcessVariables.CUSTOMER_ID))
      }
    );
  }

}
```

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.

Handling command transmission

The commands sent via gateway (e.g. `AxonCommandListGateway`) are received by Command Handlers. The latter may accept or reject commands, depending on the state of the aggregate and other components. The `AxonCommandListGateway` is informed about the command outcome. By default, it will log the outcome to console (success is logged in `DEBUG` log level, errors are using `ERROR` log level).

In some situations it is required to take care of command outcome. A prominent example is to include a metric for command dispatching errors into monitoring. For doing so, it is possible to provide own handlers for success and error command outcome.

For the Task Command Sender (as a part of `Taskpool Collector`) please provide a Spring Bean implementing the `io.holunda.camunda.taskpool.sender.gateway.TaskCommandSuccessHandler` and `io.holunda.camunda.taskpool.sender.gateway.TaskCommandErrorHandler` accordingly.

```
@Bean
@Primary
fun taskCommandErrorHandler(): TaskCommandErrorHandler = object : LoggingTaskCommandErr
  override fun apply(commandMessage: Any, commandResultMessage: CommandResultMessage<ou
    logger.info { "<--------- CUSTOM ERROR HANDLER REPORT --------->" }
    super.apply(commandMessage, commandResultMessage)
    logger.info { "<------------------- END ----------------------->" }
  }
}
```

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 {

}
```

Command transmission

In order to control sending of commands to command gateway, the command sender activation property `camunda.taskpool.dataentry.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.dataentry.sender.type` if you want to use the default command sender or provide your own implementation. The default provided command sender (type: `simple`) just sends the commands synchronously using Axon Command Bus.

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

Handling command transmission

The commands sent by the `Datapool Collector` are received by Command Handlers. The latter may accept or reject commands, depending on the state of the aggregate and other components. The `SimpleDataEntryCommandSender` is informed about the command outcome. By default, it will log the outcome to console (success is logged in `DEBUG` log level, errors are using `ERROR` log level).

In some situations it is required to take care of command outcome. A prominent example is to include a metric for command dispatching errors into monitoring. For doing so, it is possible to provide own handlers for success and error command outcome.

For Data Entry Command Sender (as a part of `Datapool Collector`) please provide a Spring Bean implementing the `io.holunda.camunda.datapool.sender.DataEntryCommandSuccessHandler` and `io.holunda.camunda.datapool.sender.DataEntryCommandErrorHandler` accordingly.

```
@Bean
@Primary
fun dataEntryCommandSuccessHandler() = object: DataEntryCommandResultHandler {
  override fun apply(commandMessage: Any, commandResultMessage: CommandResultMessage<ou
    // do something here
    logger.info { "Success" }
  }
}

@Bean
@Primary
fun dataEntryCommandErrorHandler() = object: DataEntryCommandErrorHandler {
  override fun apply(commandMessage: Any, commandResultMessage: CommandResultMessage<ou
    // do something here
    logger.error { "Error" }
  }
}
```

Taskpool Core

Taskpool Core

Purpose

The component is responsible for maintaining and storing the consistent state of the taskpool core concepts:

- Task (represents a user task instance)
- Process Definition (represents a process definition)

The component receives all commands and emits events, if changes are performed on underlying entities. The event stream is used to store all changes (purely event-sourced) and should be used by all other parties interested in changes.

Datapool Core

Datapool Core

Purpose

The component is responsible for maintaining and storing the consistent state of the datapool core concept of Business Data Entry.

The component receives all commands and emits events, if changes are performed on underlying entities. The event stream is used to store all changes (purely event-sourced) and should be used by all other parties interested in changes.

In-Memory View

In-Memory View

The In-Memory 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 in memory. The projection is transient and relies on event replay on every application start. It is good for demonstration purposes if the number of events is manageable small, but will fail to delivery high performance results on a large number of items.

Features

- uses concurrent hash maps to store the read model
- provides single query API
- provides subscription query API (reactive)
- relies on event replay and transient token store

Configuration options

In order to activate the in-memory implementation, please include the following dependency on your classpath:

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

Then, add the following annotation to any class marked as Spring Configuration loaded during initialization:

```
@Configuration
@EnableTaskPoolSimpleView
public class MyViewConfiguration {

}
```

The view implementation provides runtime details using standard logging facility. If you want to increase the logging level, please setup it e.g. in your `application.yaml`:

```
logging.level.io.holunda.camunda.taskpool.view.simple: DEBUG
```

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

In order to activate the Mongo implementation, please include the following dependency on your classpath:

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

The implementation relies on Spring Data Mongo and needs to activate those. Please add the following annotation to any class marked as Spring Configuration loaded during initialization:

```
@Configuration
@EnableTaskPoolMongoView
@Import({
    org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration.class,
    org.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration.class,
    org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration.class,
    org.springframework.boot.autoconfigure.data.mongo.MongoReactiveDataAutoConfiguration
  })
public class MyViewConfiguration {

}
```

In addition, configure a Mongo connection to database called `tasks-payload` using `application.properties` or `application.yaml`:

```
spring:
  data:
    mongodb:
      database: tasks-payload
      host: localhost
      port: 27017
```

The view implementation provides runtime details using standard logging facility. If you want to increase the logging level, please setup it e.g. in your `application.yaml`:

```
logging.level.io.holunda.camunda.taskpool.view.mongo: DEBUG
```

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:

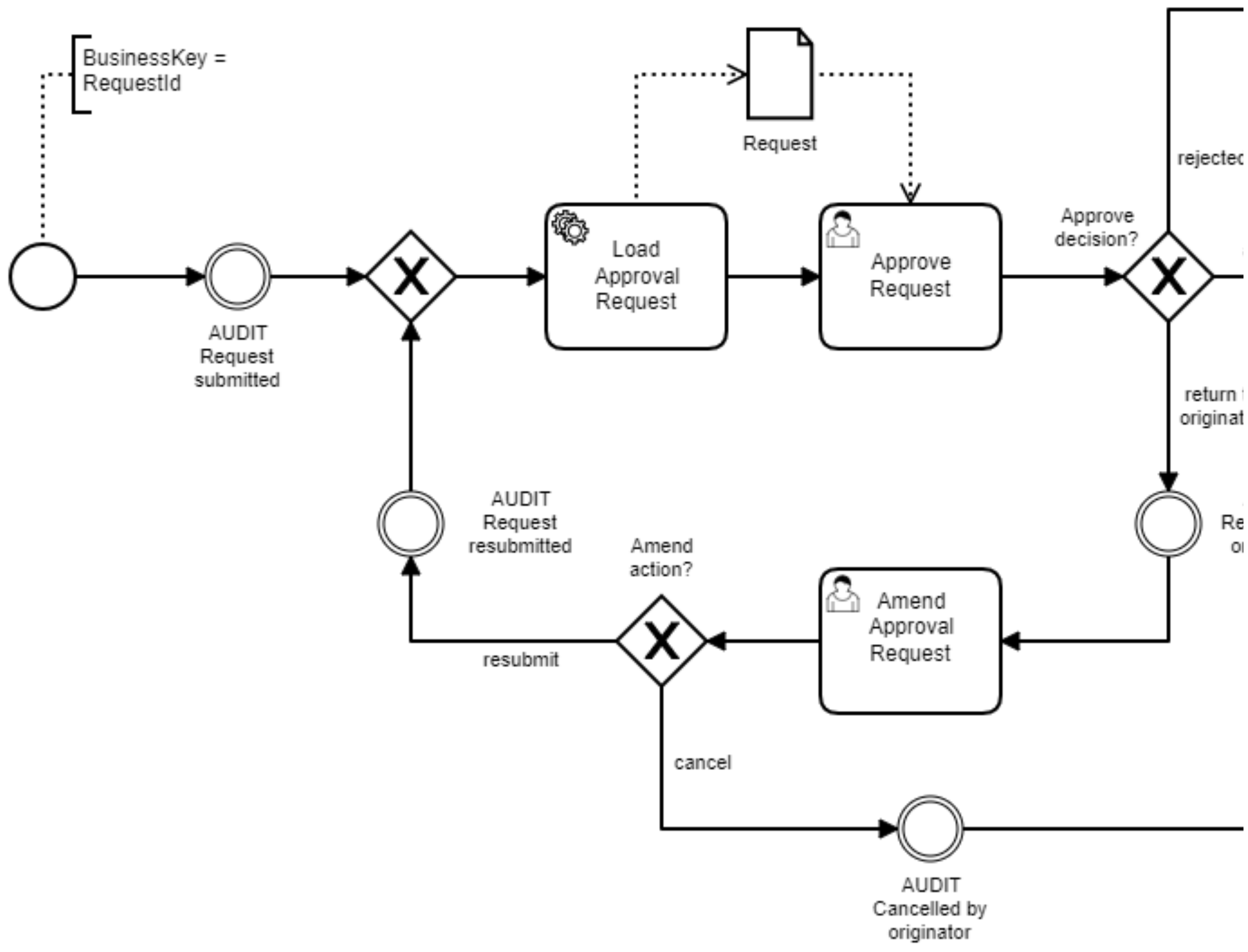
```
{
    "_id" : ObjectId("5d2b45d6a9ca33042abea23b"),
    "processorName" : "io.holunda.camunda.taskpool.view.mongo.service",
    "segment" : 0,
    "owner" : "18524@blackstar",
    "timestamp" : NumberLong(1566379093564),
    "token" : { "$binary" : "PG9yZy5heG9uZnJhbWV3b3JrLmV2ZW50aGFuZGxpbmcuR2xvYmFsU2VxdWVu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

Working Example

Working Example

Along with library modules several example modules and applications are provided, demonstrating the main features of the solution. This includes a series of example applications for usage in different Usage Scenarios.

Business context



Imagine a system that responsible for management of all requests in the company. Using this system, you can submit requests which then get eventually approved or rejected. Sometimes, the approver doesn’t approve or reject, but returns the request back to the originator (that is the person, who submitted the request). Then, the originator can amend the request and resubmit it or cancel the request.

The request is initially crated in `DRAFT` mode. It gets to state `IN PROGRESS` as soon as the process is started and will eventually get to `ACCEPTED` or `REJECTED` as a final state.

For sample purposes two groups of users are created: The Muppet Show (Kermit, Piggy, Gonzo and Fozzy) and The Avengers (Ironman, Hulk). Gonzo and Fozzy are responsible for approvals.

Story board

The following storyboard can be used to understand the mechanics behind the provided implementation:

- To start the `Approval process` for a given request open your browser and navigate to the `Tasklist` : http://localhost:8081/tasklist/. Please note that the selected user is `Ironman`. Open the menu (Start new…) in the top-left corner and select 'Approval Request'. You should see the start form for the example approval process.

- Select from one of predefined templates and click *Start*. The start form will disappear and redirect back to the `Tasklist` where the new approval request process created one task with the name *Approve Request*. If the selected user is still `Ironman` you won’t see any task, but it will be visible if you switch to `Gonzo`.

- Examine the task details by clicking *Data* tab in *Details* column. You can see the data of the request correlated to the current process instance.

- Switch to `Archive` and you should see the request business object. Examine the approval request by clicking *Data*, *Audit* and *Description* tabs in *Details* column.

- Let’s assume the request amount is too high and we want to inform `Ironman` about this. We are not rejecting the request completely, but returning it to back to `Ironman`. To do so click on the task name which opens the user task form *Approve Request* and complete the task, by selecting the *Return request to originator* option and clicking the *Complete* button. The process will create a new user task `Amend Request` for the originator, who is `Ironman` which is visible in the `Tasklist`.

- `Ironman` should now change the amount of the request in the and re-submit the request. For doing so, click on task named *Amend Approval Request*, change the amount to a new value and complete the task by selecting the *Re-submit request* option and clicking on *Complete* button.

- Again, let’s examine the data correlated with the task. Note that the amount is changed, despite the fact that the request amount is not a data item available to the process instance but is still available in the `Tasklist`. You can examine the details of the `Approval Request` in the `Archive` again.

- Switch back to the `Tasklist` and approve the request by selecting the appropriate option.

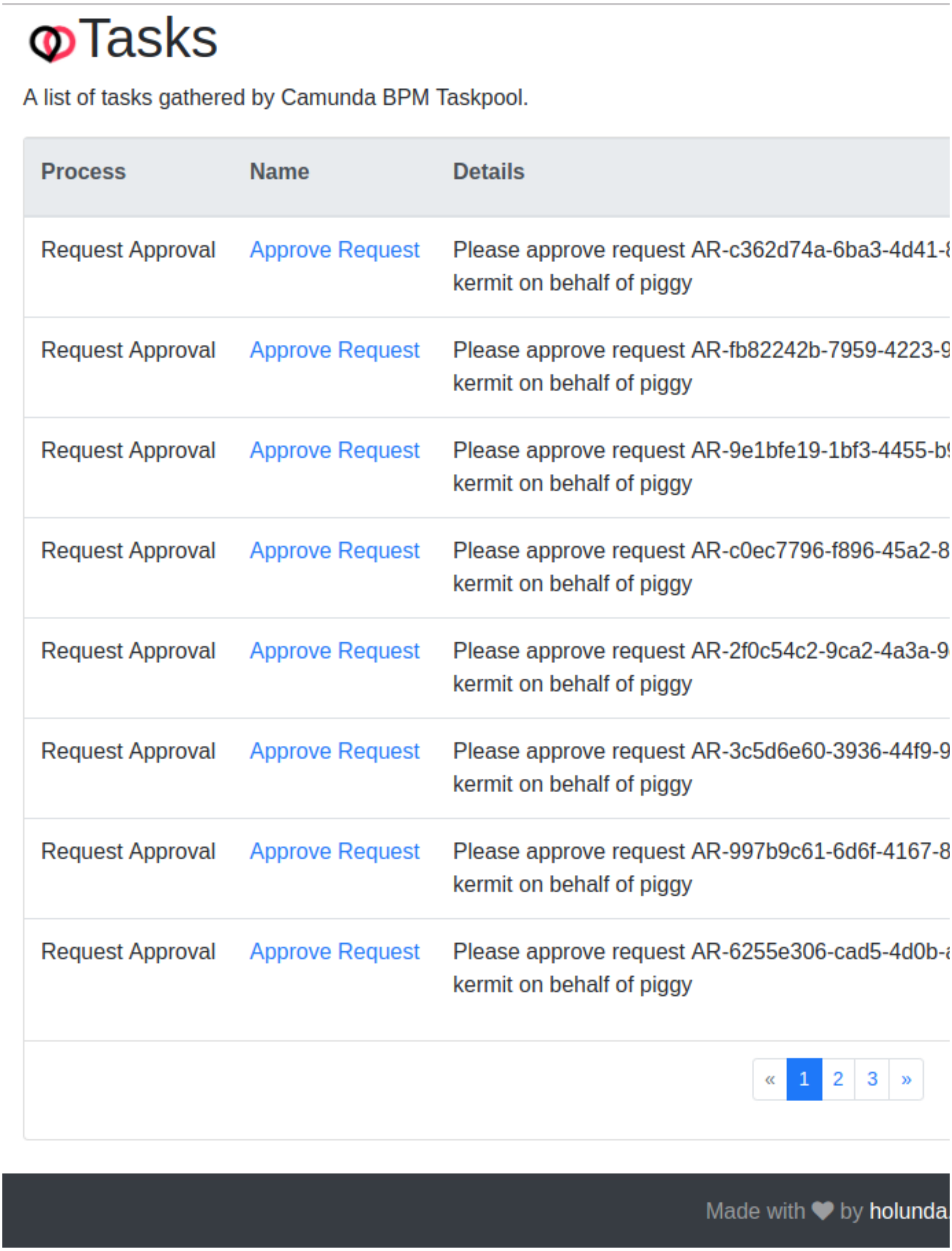
Running Examples

To run the example please consult the Usage Scenarios section.

Tip Since the process application includes Camunda BPM engine, you can use the standard Camunda webapps by navigating to http://localhost:8080/. The default user and password are `admin / admin `.

Tasklist

The currently implemented `Tasklist` is a simple application (implemented as a single-page-application based on Angular) that shows the list of taks available in the task pool. In doing so it provides the ability to filter, sort and page tasks with correlated business events. Here is how it looks like now:



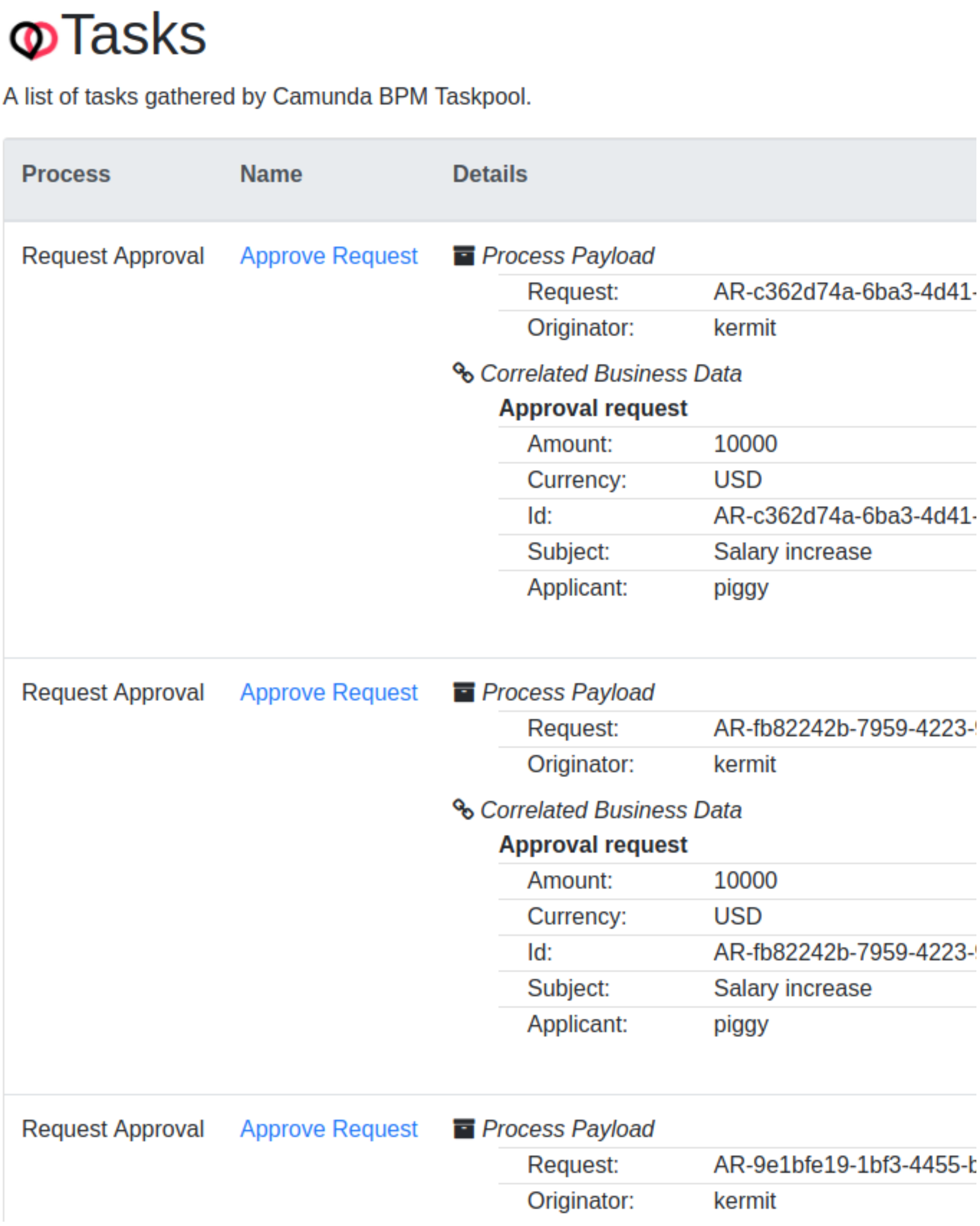
Tasks

A list of tasks gathered by Camunda BPM Taskpool.

Process	Name	Details
Request Approval	Approve Request	Please approve request AR-c362d74a-6ba3-4d41-8 kermit on behalf of piggy
Request Approval	Approve Request	Please approve request AR-fb82242b-7959-4223-9 kermit on behalf of piggy
Request Approval	Approve Request	Please approve request AR-9e1bfe19-1bf3-4455-b kermit on behalf of piggy
Request Approval	Approve Request	Please approve request AR-c0ec7796-f896-45a2-8 kermit on behalf of piggy
Request Approval	Approve Request	Please approve request AR-2f0c54c2-9ca2-4a3a-9 kermit on behalf of piggy
Request Approval	Approve Request	Please approve request AR-3c5d6e60-3936-44f9-9 kermit on behalf of piggy
Request Approval	Approve Request	Please approve request AR-997b9c61-6d6f-4167-8 kermit on behalf of piggy
Request Approval	Approve Request	Please approve request AR-6255e306-cad5-4d0b-a kermit on behalf of piggy

« 1 2 3 »

Made with ❤ by holunda



Tasks

A list of tasks gathered by Camunda BPM Taskpool.

Process	Name	Details
Request Approval	Approve Request	*Process Payload* Request: AR-c362d74a-6ba3-4d41- Originator: kermit *Correlated Business Data* **Approval request** Amount: 10000 Currency: USD Id: AR-c362d74a-6ba3-4d41- Subject: Salary increase Applicant: piggy
Request Approval	Approve Request	*Process Payload* Request: AR-fb82242b-7959-4223- Originator: kermit *Correlated Business Data* **Approval request** Amount: 10000 Currency: USD Id: AR-fb82242b-7959-4223- Subject: Salary increase Applicant: piggy
Request Approval	Approve Request	*Process Payload* Request: AR-9e1bfe19-1bf3-4455- Originator: kermit

Features

- Lists tasks in the system for selected user
- Allows for switching users
- Tasks include information about the process, name, description, create time, due date, priority and assignment.

- Tasks include process data (from process instance)
- Tasks include correlated business data
- The list of tasks is sortable
- The list of tasks is paged (7 items per page)
- Claiming / Unclaiming
- Jump to form
- Allows to start processes

Ongoing / TODO

- Filtering

Archive list

Features

- Lists business object in the system for selected user
- Allows for switching users
- Business objects include information about the type, status (with sub status), name, details
- Business objects include details about contained data
- Business objects include audit log with all state changes
- The list is paged (7 items per page)

Ongoing / TODO

- Business object view

Usage Scenarios

Usage Scenarios

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

The simplest setup is to run all components on a single node. A more advanced scenario is to distribute components

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. In particular, two scenarios can be distinguished: using Axon Server or using a different distribution technology.

The provided Example is implemented several times demonstrating the following usage scenarios:

- Single Node Scenario
- Distributed Scenario using Axon Server
- Distributed Scenario without Axon Server

Scenario for running on a single node

Scenario for running on a single node

Distributed Scenario using Axon Server

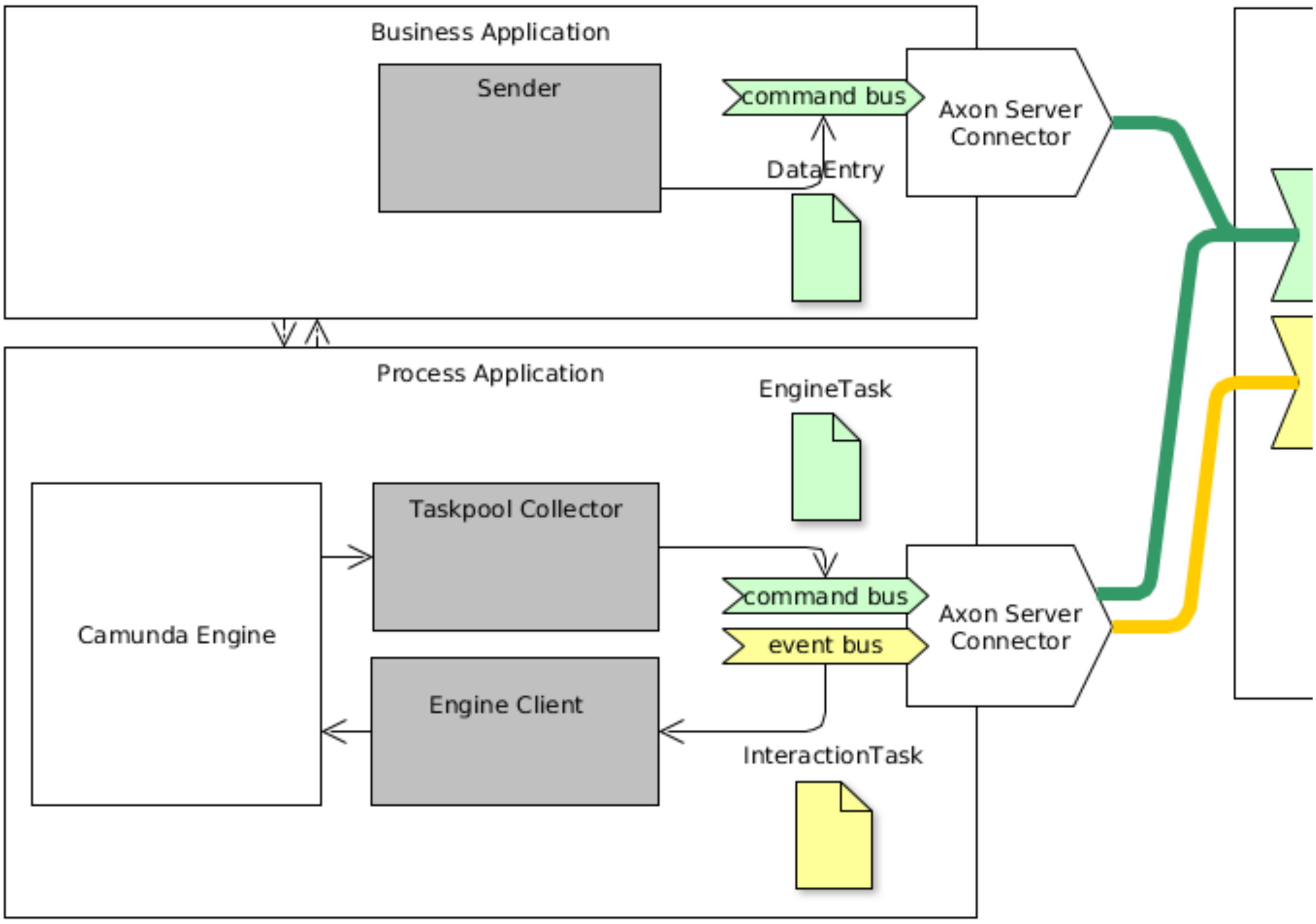
Distributed Scenario using Axon Server

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.



Running Example

The following application is an example demonstrating the usage of the Camunda BPM Taskpool. The application is built as a SpringBoot process application and shows a simple approval process.

System Requirements

- JDK 8
- Docker
- Docker Compose

Preparations

Before you begin, please build the entire project with `mvn clean install` from the command line in the project root directory.

You will need some backing services (Axon Server, PostgreSQL, MongoDB) and you can easily start them locally by using the provided `docker-compose.yml` file.

Before you start change the directory to `examples/scenarios/distributed-axon-server`, and execute once:

```
cd examples/scenarios/distributed-axon-server
.docker/setup.sh
```

Now, start required containers. The easiest way to do so is to run:

```
docker-compose up -d
```

To verify it is running, open your browser http://localhost:8024/. You should see the Axon Server administration console.

Start

The demo application consists of several Maven modules. In order to start the example, you will need to start only two of them in the following order:

1. taskpool-application
2. process-application (example application)

The modules can be started by running from command line in the top-level directory using Maven or start the packaged application using:

```
java -jar taskpool-application/target/*.jar
java -jar process-application/target/*.jar
```

Goto: http://localhost:8081/tasklist/

Scenario without Axon Server

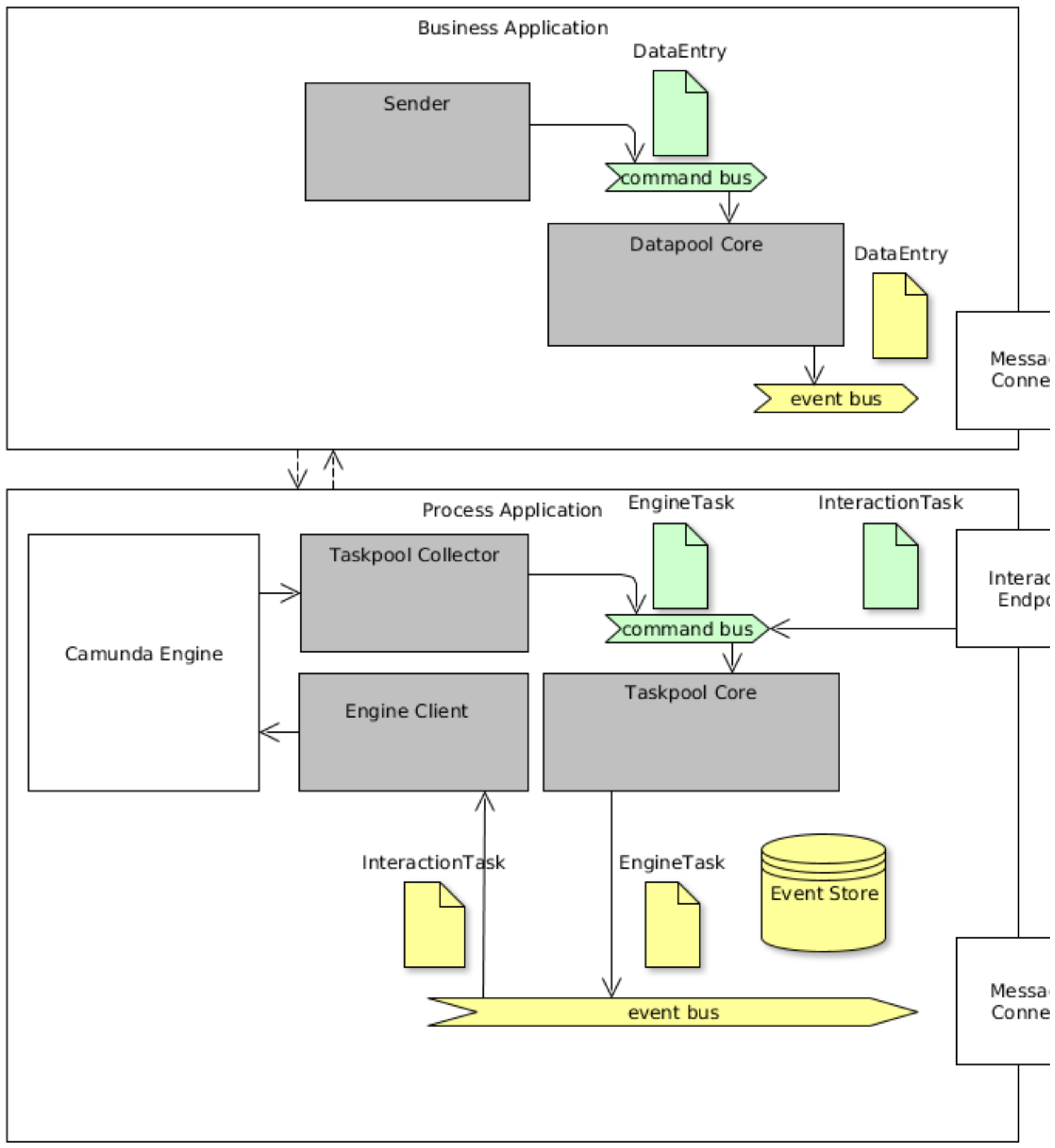
Scenario without Axon Server

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:

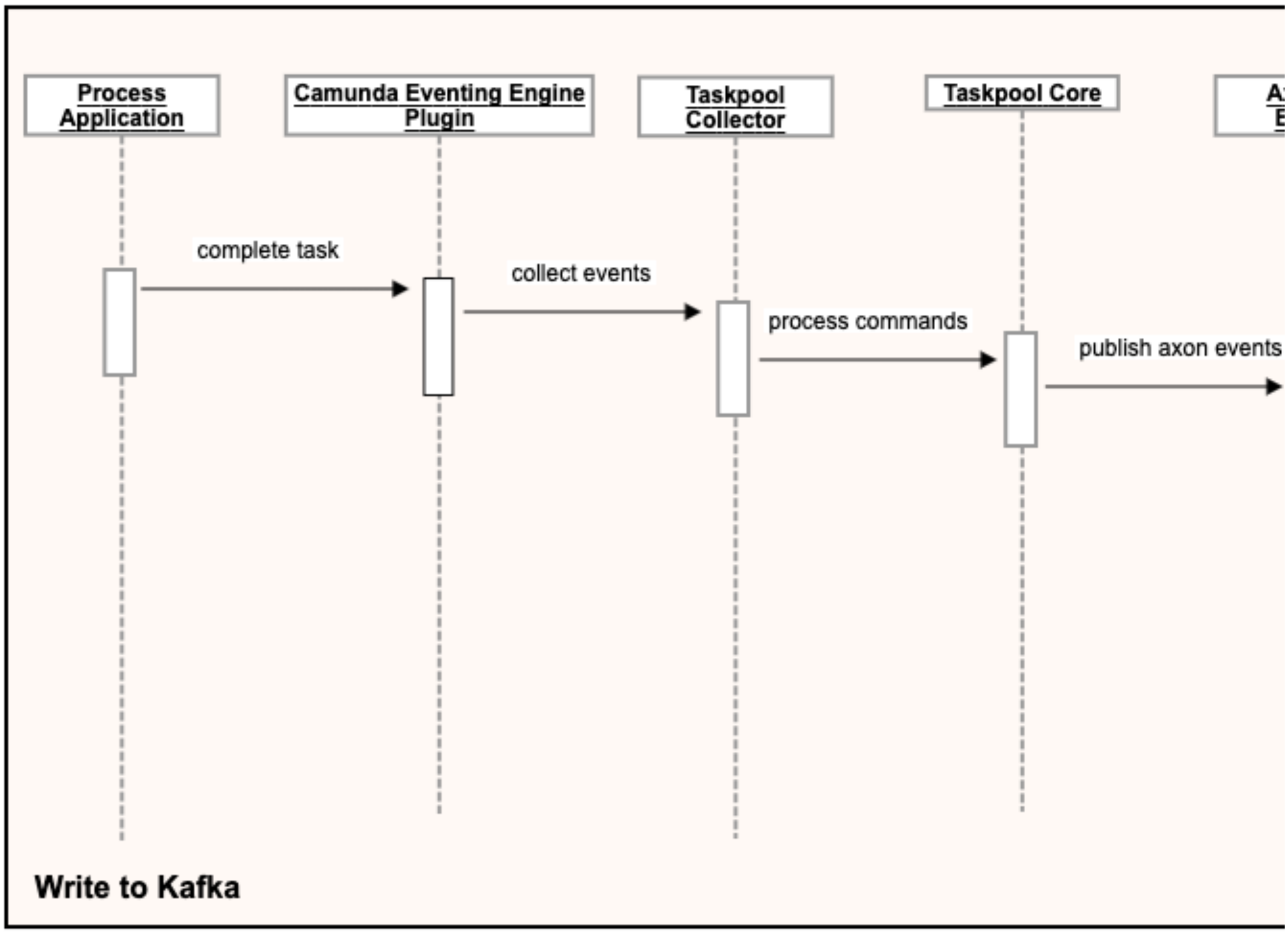
- distributed task pool / data pool
- view must be persistent
- direct communication between task list / engines required (addressing, routing)
- concurrent access to engines might become a problem (no unit of work guarantees)

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



The following diagram depicts the task run from Process Application to the end user, consuming it via Tasklist API.

Process Application to Tasklist API Messaging (Top Level View)



- The `CamundaEventingEnginePlugin` provided with the Taskpool tracks events in the Camunda engine (e.g. the creation, deletion or modification of a User Task) and makes them available as Spring events.

- The `Taskpool Collector` component listens to those events. It collects all relevant events that happen in a single transaction and registers a transaction synchronization to process them beforeCommit. Just before the transaction is committed, the collected events are accumulated and sent as Axon Commands through the `CommandGateway`.

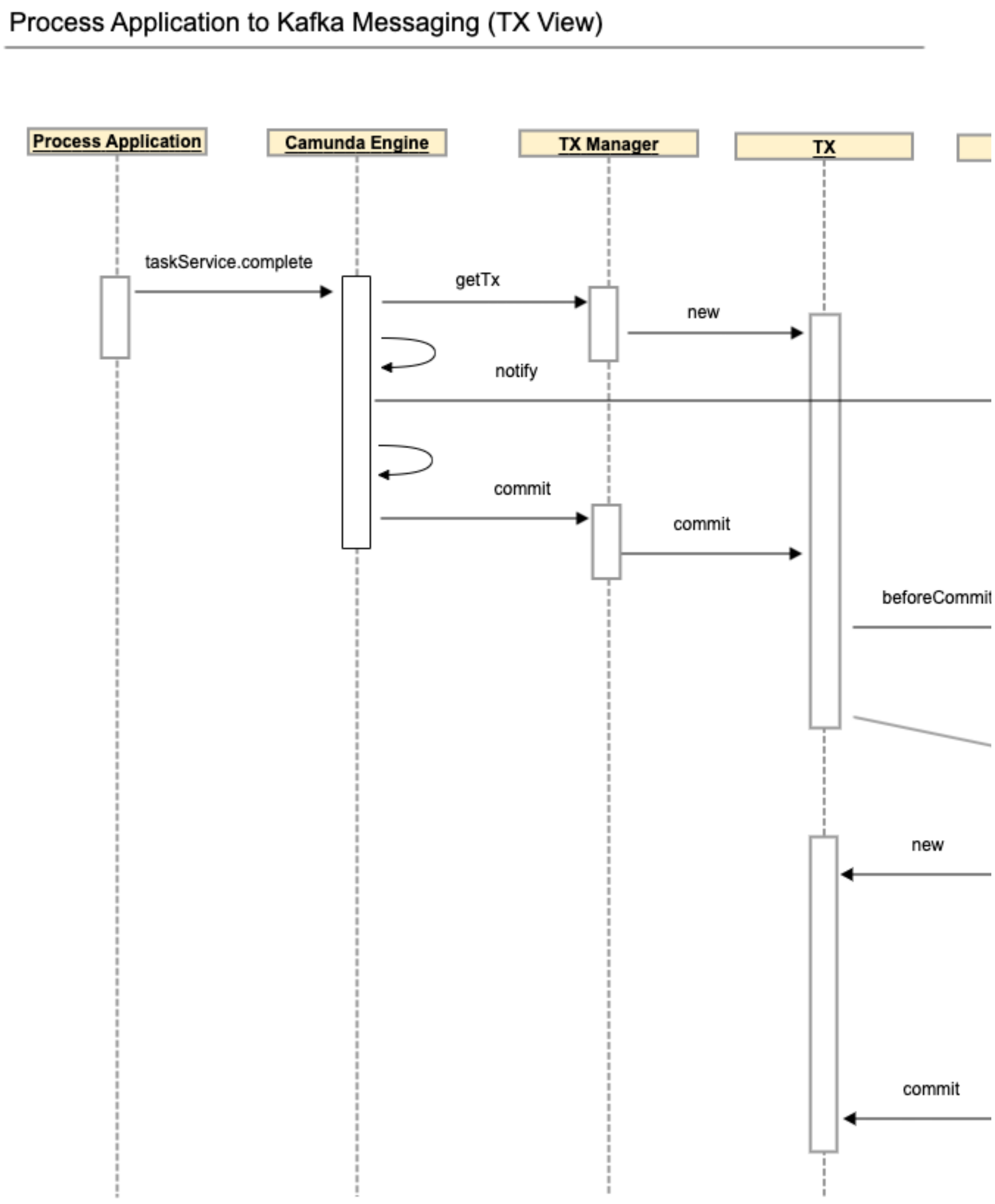
- The `Taskpool Core` processes those commands and issues Axon Events through the EventGateway which are stored in Axon's database tables within the same transaction.

- The transaction commit finishes. If anything goes wrong before this point, the transaction rolls back and it is as though nothing ever happened.

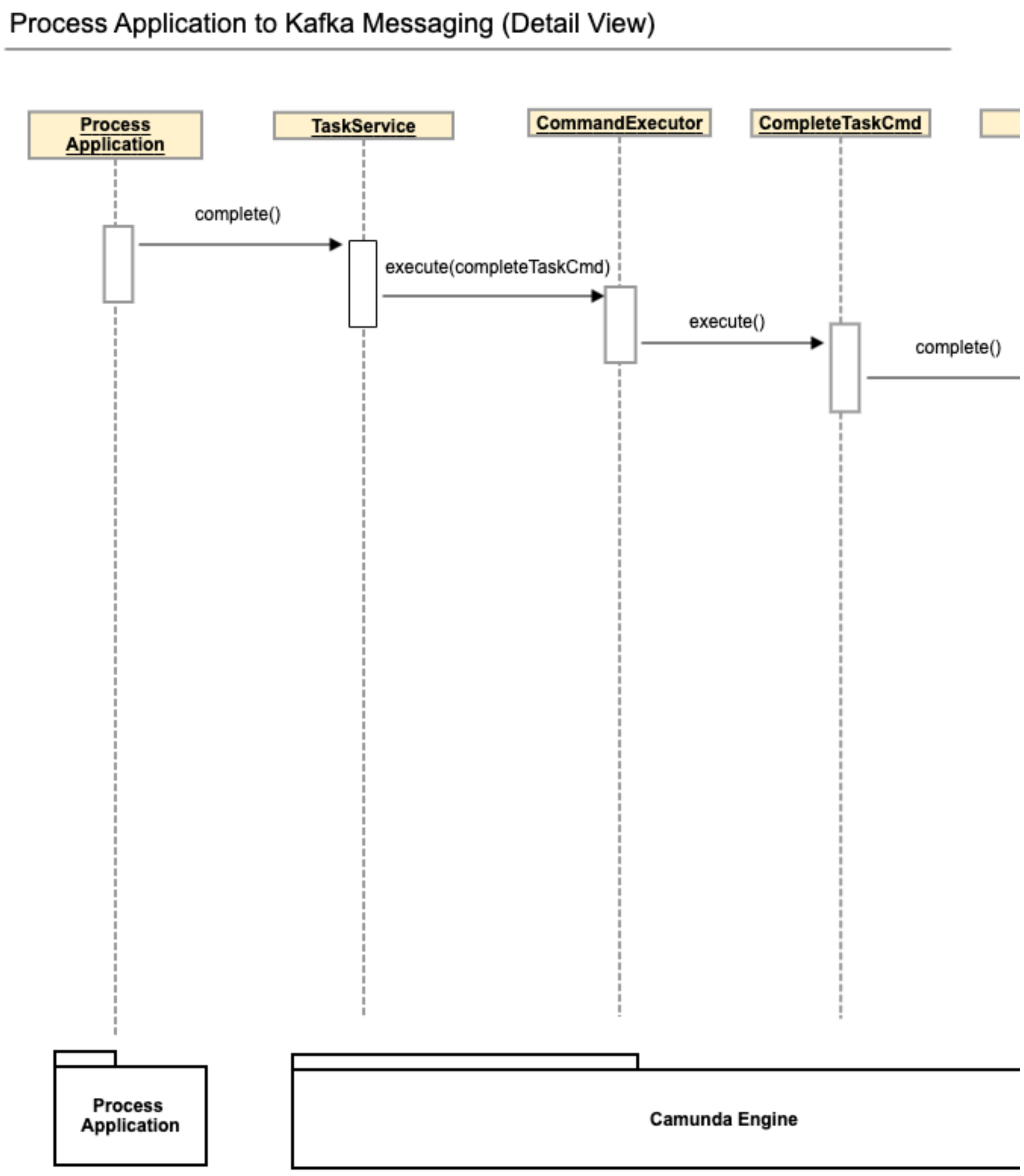
- In the `Axon Kafka Extension`, a `TrackingEventProcessor` polls for events and sees them as soon as the transaction that created them is committed. It sends each event to Kafka and waits for an acknowledgment from Kafka. If sending fails or times out, the event processor goes into error mode and retries until it succeeds. This can lead to events being published to Kafka more than once but guarantees at-least-once delivery.

- Within the Tasklist API, the `Axon Kafka Extension` polls the events from Kafka and another TrackingEventProcessor forwards them to the `TaskPoolMongoService` where they are processed to update the Mongo DB accordingly.

- When a user queries the Tasklist API for tasks, two things happen: Firstly, the Mongo DB is queried for the current state of tasks for this user and these tasks are returned. Secondly, the Tasklist API subscribes to any changes to the Mongo DB. These changes are filtered for relevance to the user and relevant changes are returned after the current state as an infinite stream until the request is cancelled or interrupted for some reason.

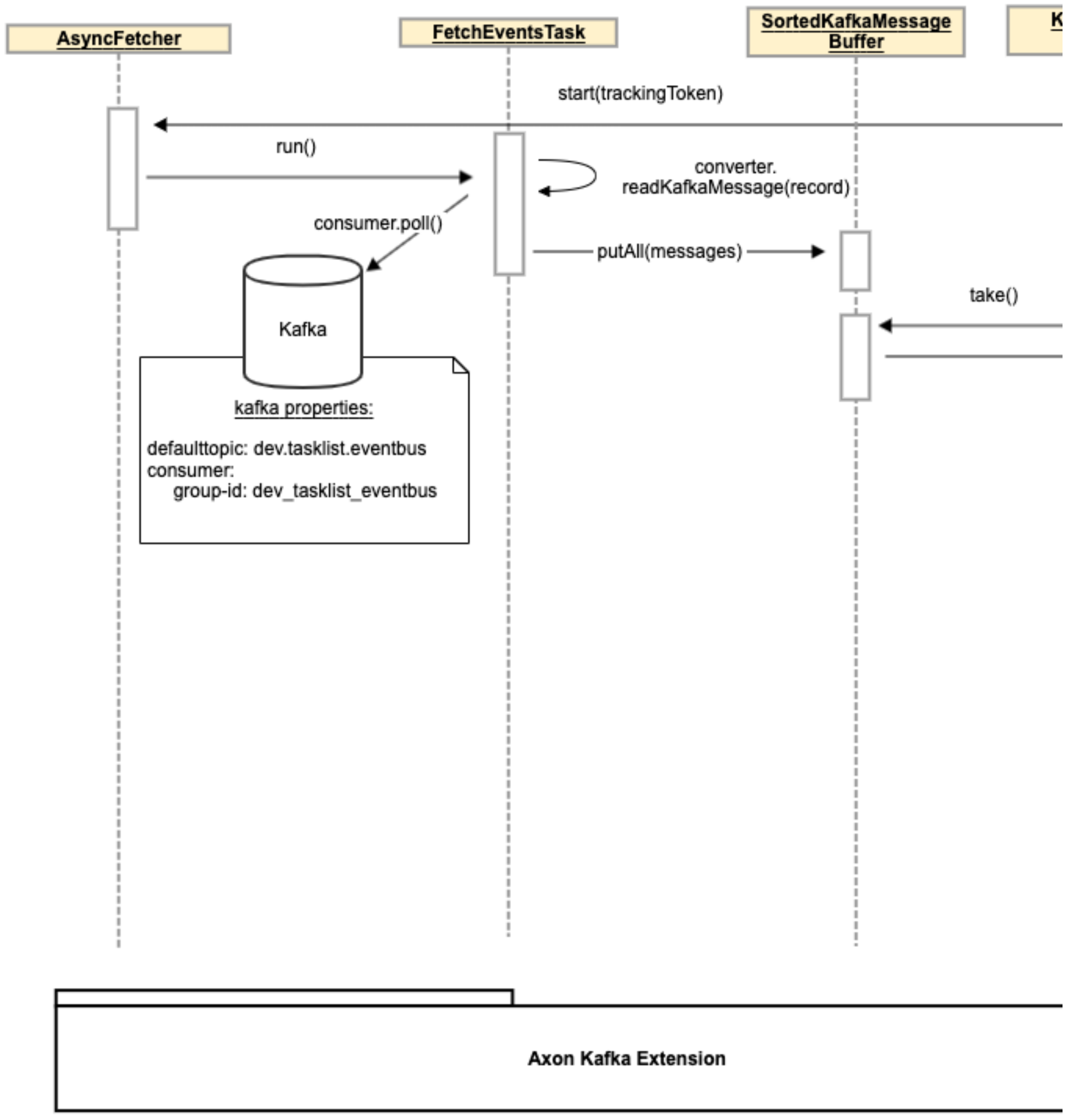


From Process Application to Kafka



From Kafka to Tasklist API

Kafka to Tasklist API Messaging (Detail View)


AsyncFetcher
FetchEventsTask
SortedKafkaMessage Buffer
K
start(trackingToken)
run()
converter.
readKafkaMessage(record)
consumer.poll()
putAll(messages)
take()
Kafka
kafka properties:
defaulttopic: dev.tasklist.eventbus
consumer:
group-id: dev_tasklist_eventbus
Axon Kafka Extension