User Guide

Table of Contents

Working Example Application

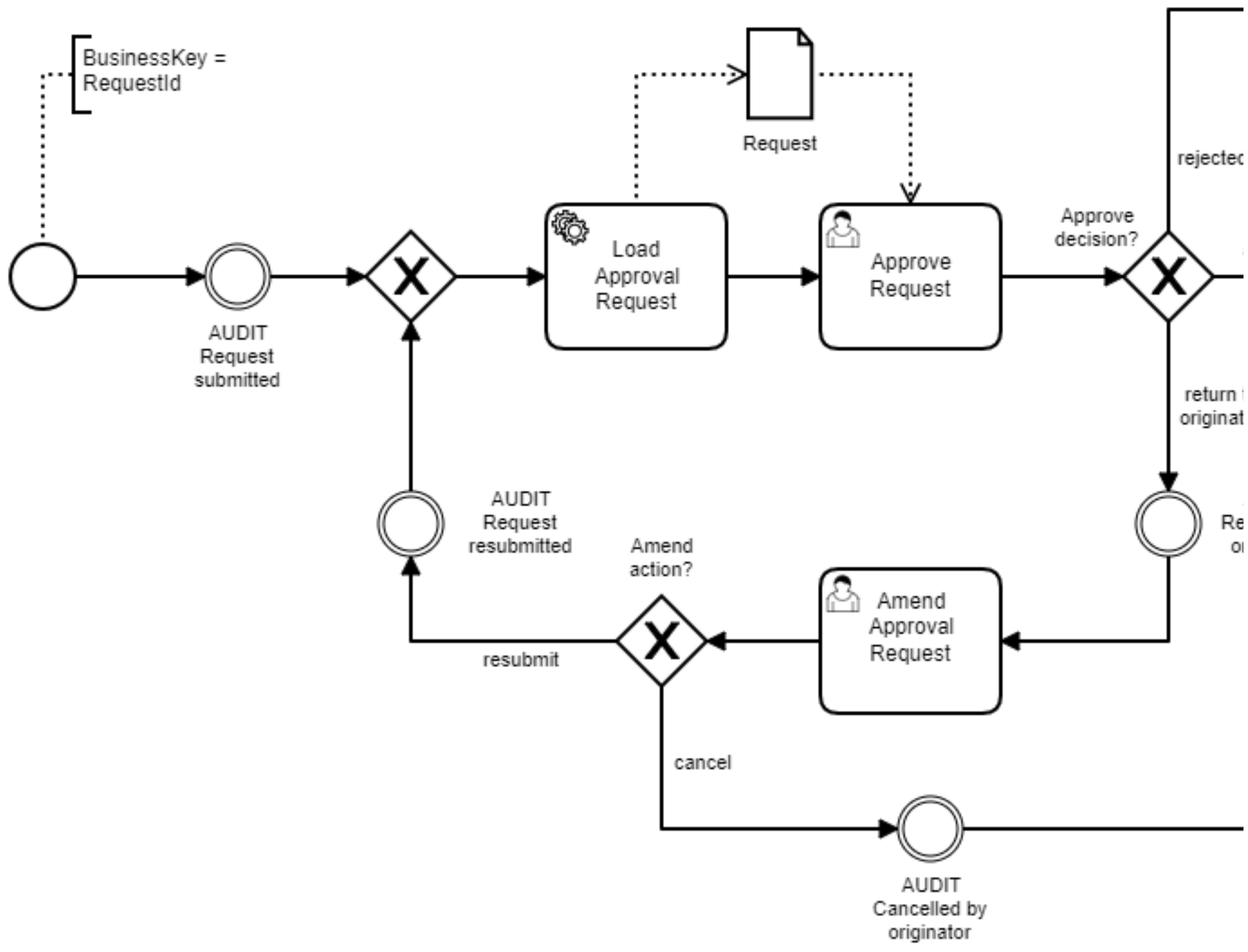
Components

Example Application

Example Application

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. They all share the same business process described in the next section.

Business context



Take a look on the process model above. Imagine you are building a system that responsible for management of all approval 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* for correction (that is the person, who submitted the request). Then, the *originator* can amend the request and resubmit it or cancel the request.

An approval request is modelled in the following way. The *subject* describes what the request is about, the *applicant* is the person whom it is about (can be different from *originator*). Finally, the *amount* and *currency* denote the costs of the request. All requests must be stored for compliance purposes.

The request is initially created 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.

Process Run

Let’s play the following run through this process model.

- `Ironman` submits an Advanced Training request on behalf of `Hulk`
- The request costs are provided in wrong currency and `Gonzo` returns the request to `Ironman` for correction (EUR instead of USD)
- `Ironman` changes the currency to USD and re-submits the request
- `Gonzo` is out of office, so `Fozzy` takes over and approves the request

Running Examples

To run the example please consult the Usage Scenarios section.

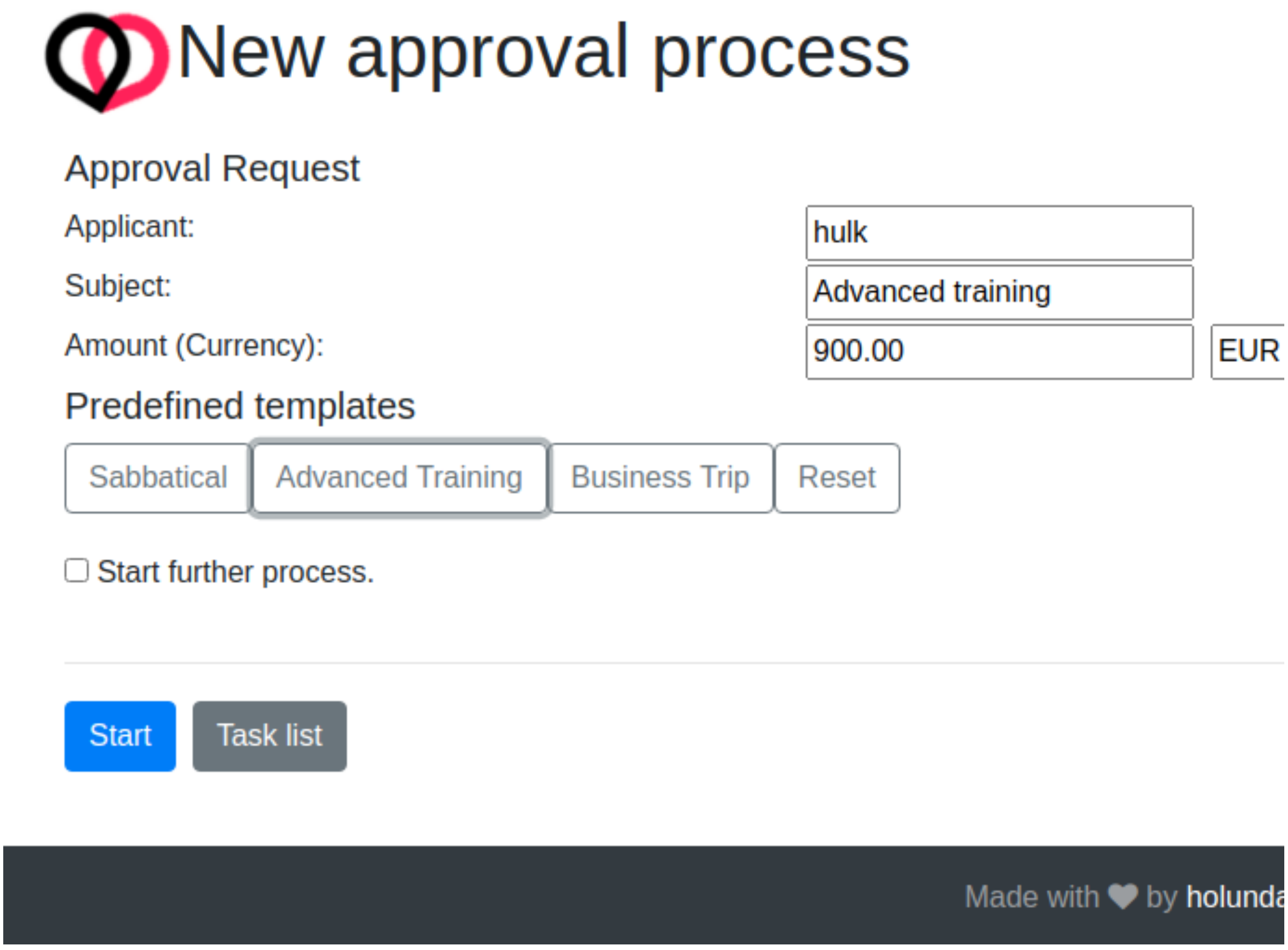
Tip Since the process application includes Camunda BPM engine, you can use the standard Camunda webapps by navigating to http://localhost:8080/camunda/app/. The default user and password are `admin / admin`.

Story board

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

Tip In this storyboard, we assume you started the single node scenario and the application runs locally on http://localhost:8080. Please adjust the URLs accordingly, if you started differently.

- To start the approval process for a given request open your browser and navigate to the `Example Tasklist`: http://localhost:8080/taskpool/. Please note that the selected user is `Ironman`.
- Open the menu (`Start new…`) in and select 'Request Approval'. You should see the start form for the example approval process.



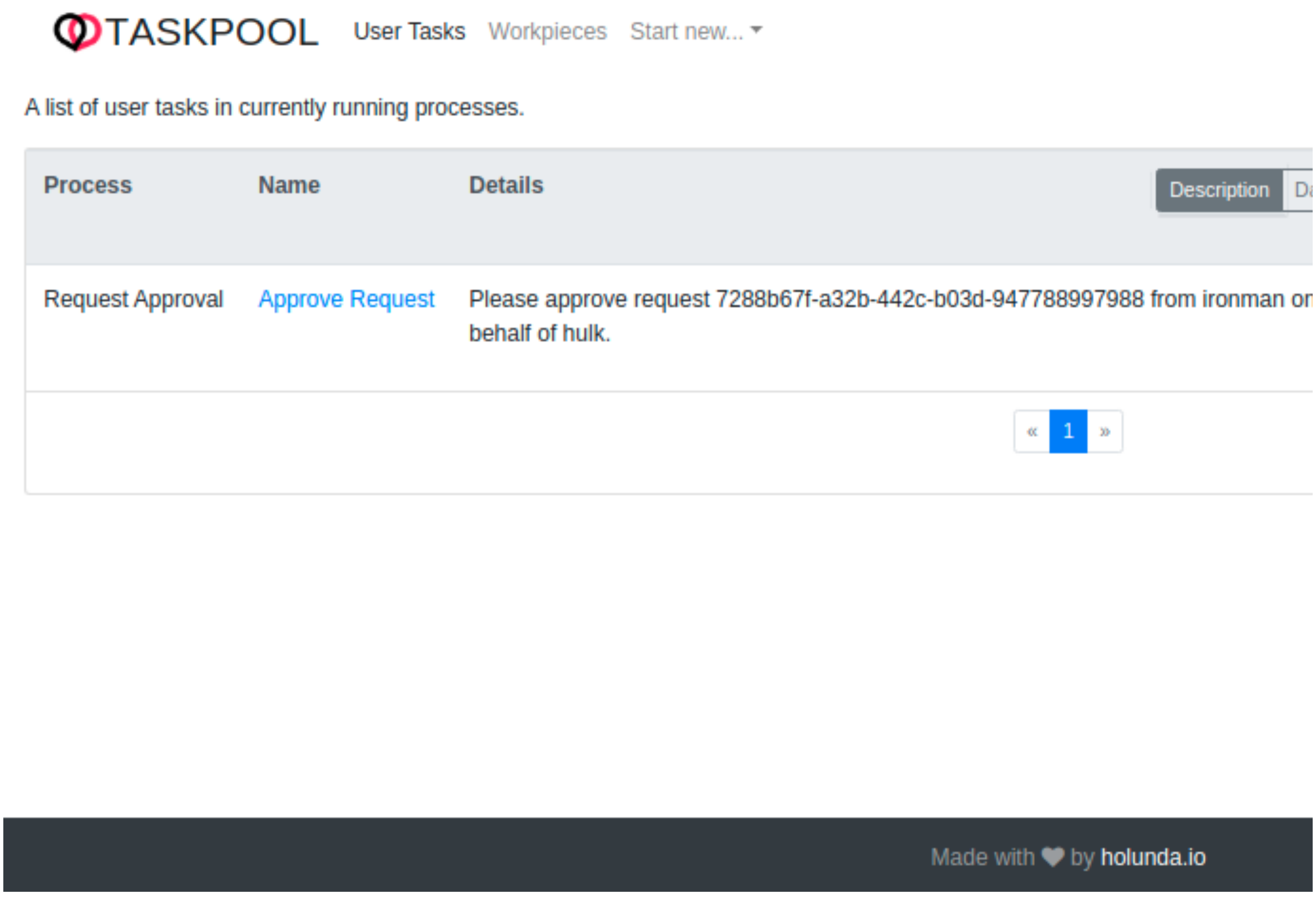
New approval process

Approval Request

Applicant:	hulk
Subject:	Advanced training
Amount (Currency):	900.00 EUR

Predefined templates

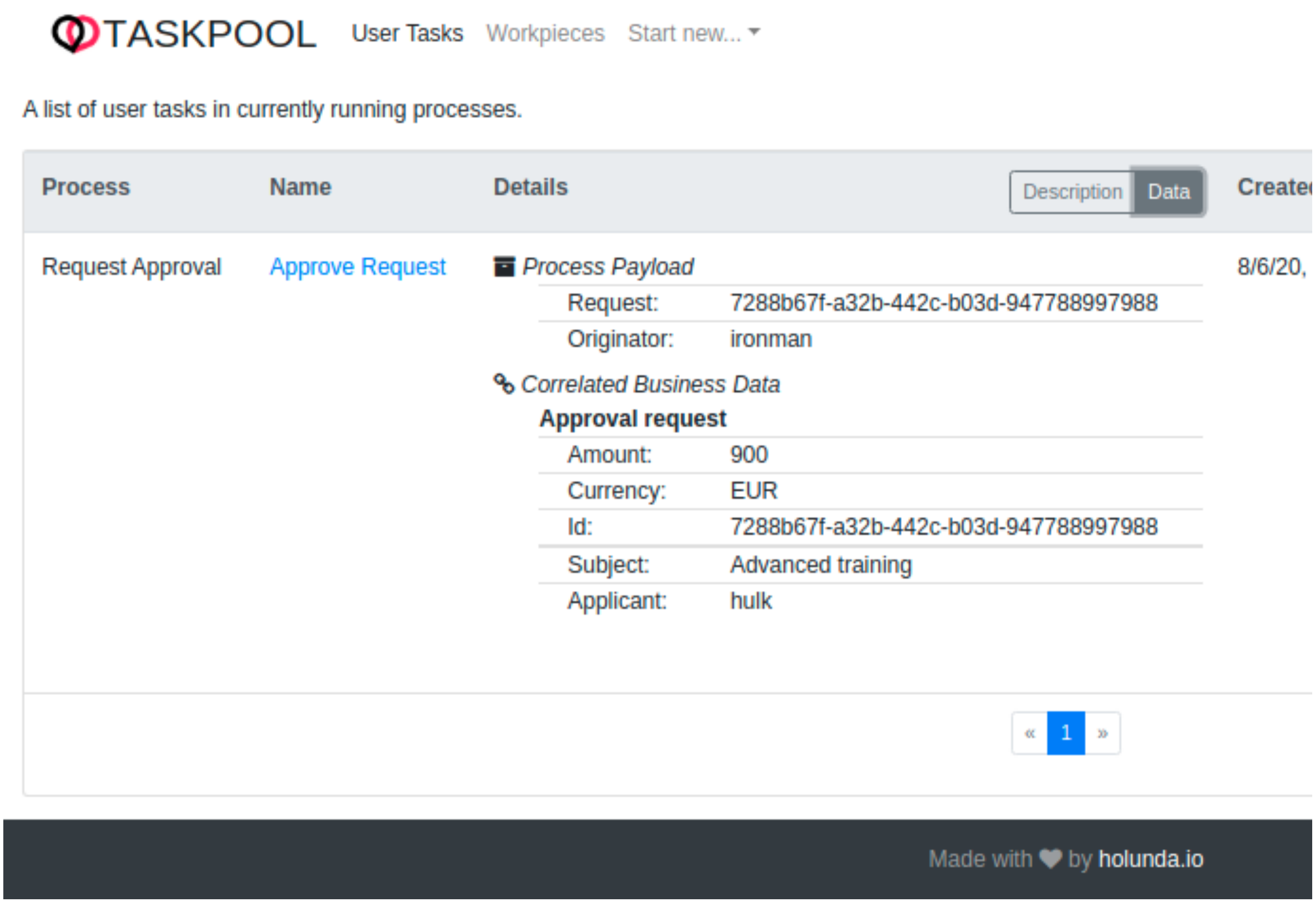
- Select `Advanced Training` from one of predefined templates and click *Start*. The start form will disappear and redirect back to the empty `Tasklist`.
- Since you are still acting as `Ironman` there are nothing you can do here. Please switch the user to `Gonzo` in the top right corner and you should see the user task `Approve Request` from process `Request Approval`.



A list of user tasks in currently running processes.

Process	Name	Details
Request Approval	Approve Request	Please approve request 7288b67f-a32b-442c-b03d-947788997988 from ironman on behalf of hulk.

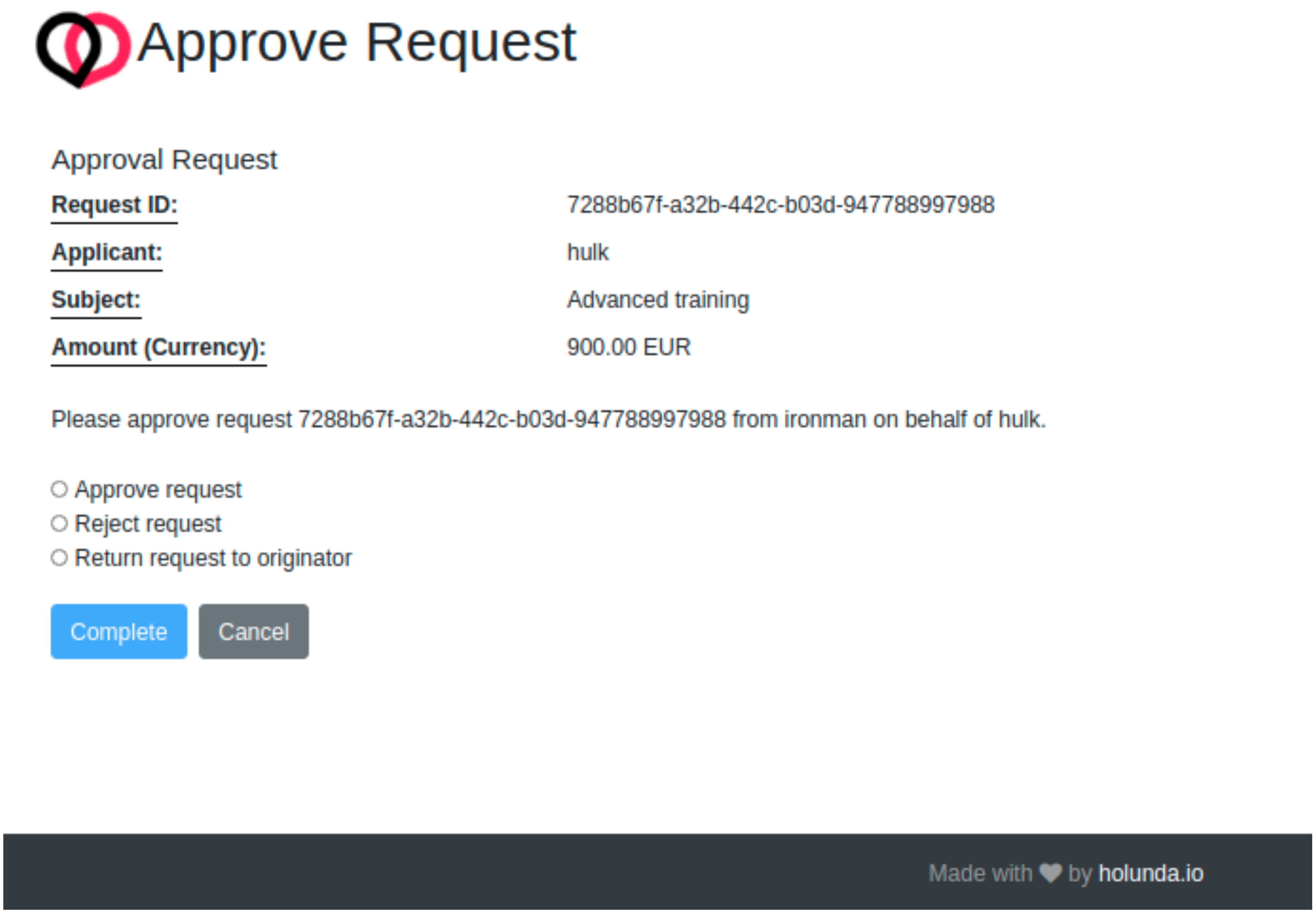
- Examine the task details by clicking *Data* tab in *Details* column. You can see the data of the request correlated to the current process instance.



A list of user tasks in currently running processes.

Process	Name	Details	Create
Request Approval	Approve Request	*Process Payload* Request: 7288b67f-a32b-442c-b03d-947788997988; Originator: ironman; *Correlated Business Data* **Approval request** Amount: 900; Currency: EUR; Id: 7288b67f-a32b-442c-b03d-947788997988; Subject: Advanced training; Applicant: hulk	8/6/20,

- Click on the task name and you will see the user task form of the `Approve Request` task. Select the option `Return request to originator` and click complete.



Approve Request

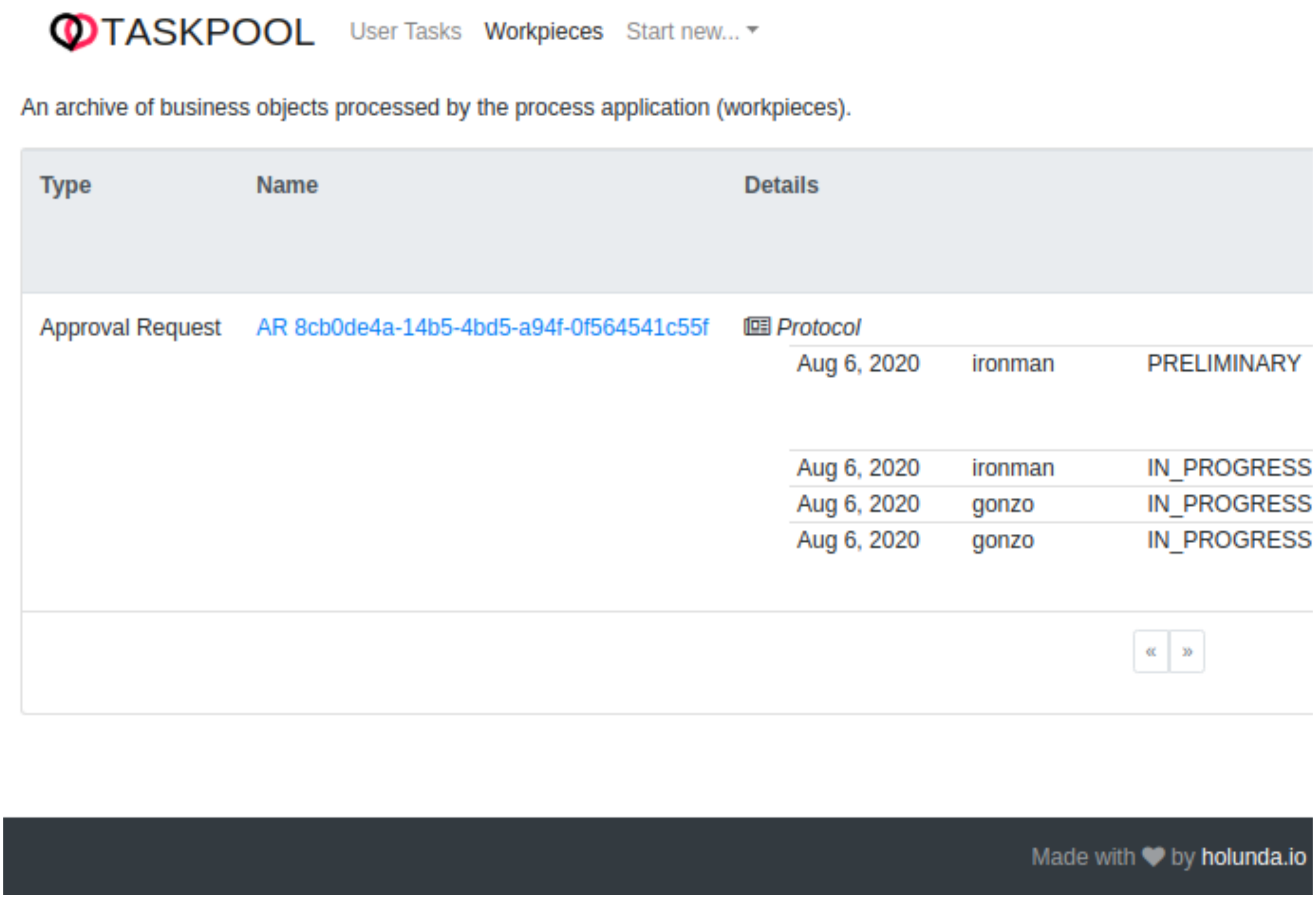
Approval Request

Request ID:	7288b67f-a32b-442c-b03d-947788997988
Applicant:	hulk
Subject:	Advanced training
Amount (Currency):	900.00 EUR

Please approve request 7288b67f-a32b-442c-b03d-947788997988 from ironman on behalf of hulk.

- ○ Approve request
- ○ Reject request
- ○ Return request to originator

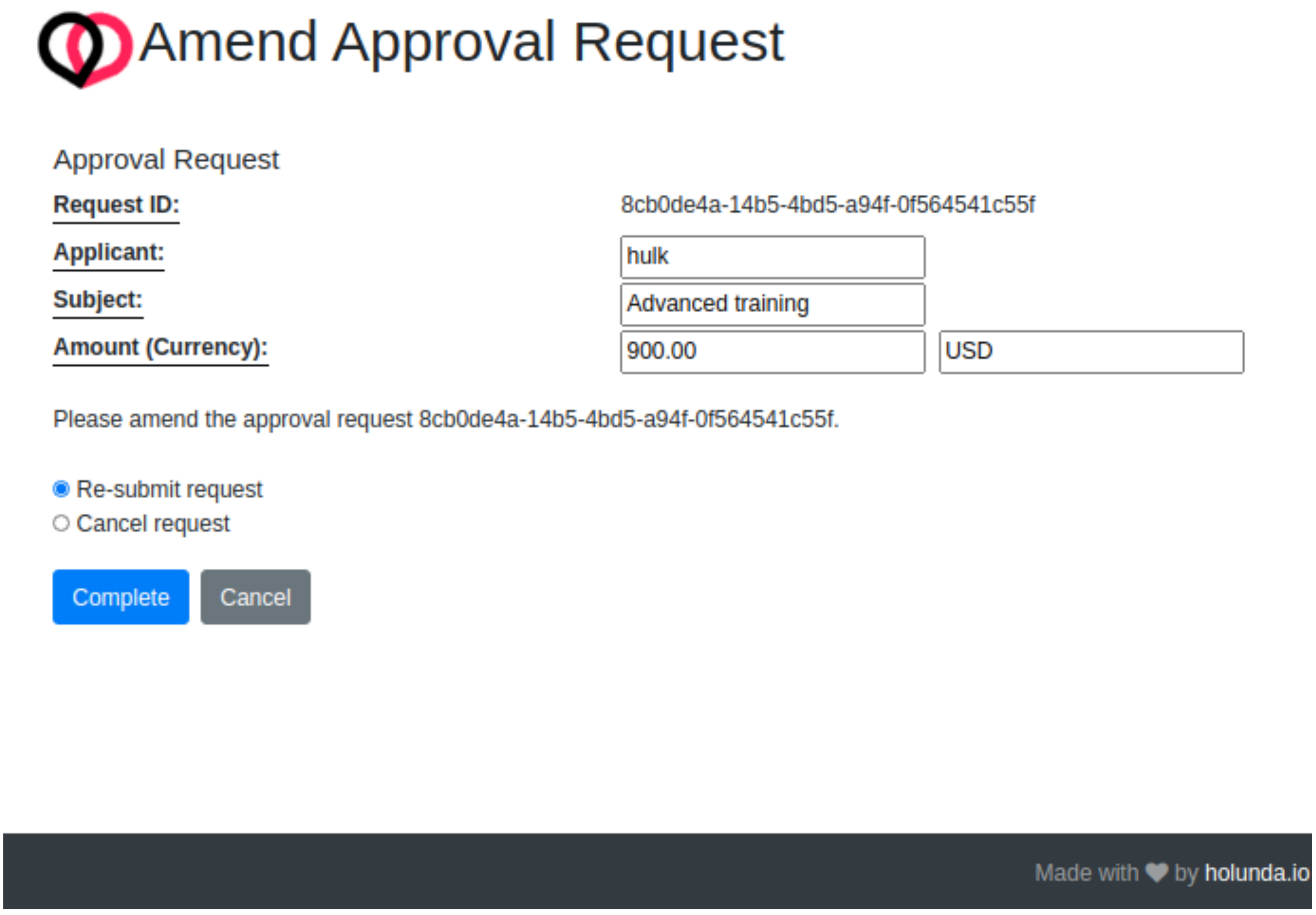
- Switch to `Workpieces` and you should see the request business object. Examine the approval request by clicking *Data*, *Audit* and *Description* tabs in *Details* column.



An archive of business objects processed by the process application (workpieces).

Type	Name	Details		
Approval Request	AR 8cb0de4a-14b5-4bd5-a94f-0f564541c55f	*Protocol*		
		Aug 6, 2020	ironman	PRELIMINARY
		Aug 6, 2020	ironman	IN_PROGRESS
		Aug 6, 2020	gonzo	IN_PROGRESS
		Aug 6, 2020	gonzo	IN_PROGRESS

- Change user back to `Ironman` and switch back to the `Tasklist` and open the `Amend request`task. Change the currency to `USD` and re-submit the request.


Amend Approval Request

Approval Request

Request ID:	8cb0de4a-14b5-4bd5-a94f-0f564541c55f	
Applicant:	hulk	
Subject:	Advanced training	
Amount (Currency):	900.00	USD

Please amend the approval request 8cb0de4a-14b5-4bd5-a94f-0f564541c55f.

[✓ Re-submit request]
Cancel request

- Change user back to `Fozzy`, open the `Approve Request` task and approve the request by selecting the appropriate option.
- Switch to `Workpieces` and you should still see the request business object, even after the process is finished. Examine the approval request by clicking *Data*, *Audit* and *Description* tabs in *Details* column.

Example Components

Example Components

For demonstration purposes we built several example components and reuse them to demonstrate the Example in different Usage Scenarios. These components are not part of a taskpool distribution and serve demonstration purposes only. They still show a trivial implementation of components which needs to be implemented by you and are not part of the provided library.

Process Application Example Components

To show the integration of `camunda-bpm-taskpool` components into a process application, the process discussed in Example has been implemented.

- Process Application Frontend
- Process Application Backend

Process Platform Example Components

To show the integration of `camunda-bpm-taskpool` components into a process platform, a simple task list and a workpieces view (archive view for business objects) has been implemented.

- Process Platform Frontend
- Process Platform Backend

Shared Example Components

Components used by other example components.

- User Management

Process Application Frontend

Process Application Frontend

The process application backend is implementing the user task forms and business object views for the example application. It is built as a typical Angular Single Page Application (SPA) and provides views for both user tasks and the business object. It communicates with process application backend via REST API, defined in the latter.

The user primarily interacts with the process platform which seamless integrate with the process applications. Usually, it provides integration points for user-task embedding / UI-composition. Unfortunately, this topic strongly depends on the frontend technology and is not a subject we want to demonstrate in this example. For simplicity, we built a very simple example, skipping the UI composition / integration entirely.

The navigation between the process platform and process application is just as simple as a full page navigation of a hyperlink.

Process Application Backend

Process Application Backend

The process application backend is implementing the process described in the Example. It demonstrates a typical three-tier application, following the Boundary-Control-Entity pattern.

Boundary

The REST API is defined using OpenAPI specification and is implemented by Spring MVC controllers. It defines of four logical parts:

- Environment Controller
- Request Controller
- Approve Task Controller
- Amend Task Controller

Control

The control tier is implemented using stateless Spring Beans and orchestrated by the Camunda BPM Process. Typical `JavaDelegate` and `ExecutionListener` are used as a glue layer. Business services of this layer are responsible for the integration with `Datapool Components` to reflect the status of the `Request` business entity. The Camunda BPM Engine is configured to use the `TaskCollector` to integrate with remaining components of the `camunda-bpm-taskpool`.

Entity

The entity tier is implemented using Spring Data JPA, using Hibernate entities. Application data and process engine data is stored using a RDBMS.

User Management

User Management

Usually, a central user management like a Single-Sign-On (SSO) is a part deployed into the process application landscape. This is responsible for authentication and authorization of the user and is required to control the role-based access to user tasks.

In our example application, we *disable any security checks* to avoid the unneeded complexity. In doing so, we implemented a trivial user store holding some pre-defined users used in example components and allow to simply switch users from the frontend by choosing a different user from the drop-down list.

It is integrated with the example frontends by passing around the user id along with the requests and provide a way to resolve the user by that id. Technically speaking, the user id can be used to retrieve the permissions of the user and check them in the backend.

Process Platform Frontend

Process Platform Frontend

The example process platform frontend provides example implementation of two views:

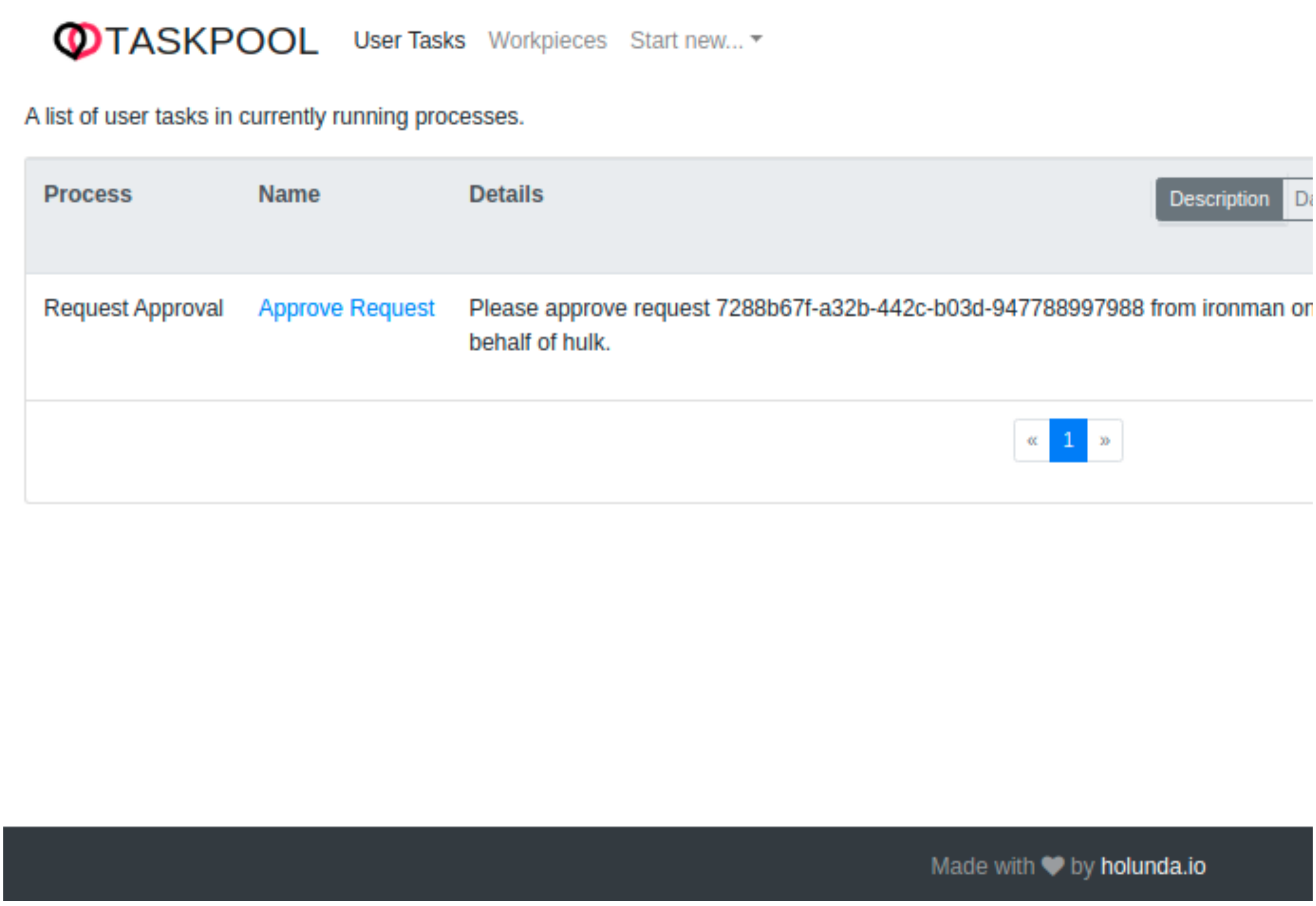
- Example Tasklist
- Example Workpieces List

Example Tasklist

Example Tasklist is a simple implementation of task inbox for a single user. The it provides the following features:

- Lists tasks in the system for selected user
- Allows for switching users (faking different user login for demonstration purposes)
- 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 (correlated via variable from process instance)
- The list of tasks is sortable
- The list of tasks is pageble (7 items per page)
- Allows claiming / unclaiming
- Provides a deeplink to the user task form provided by the process application
- Allows starting new process instances

Here is, how it looks like showing task descriptions:

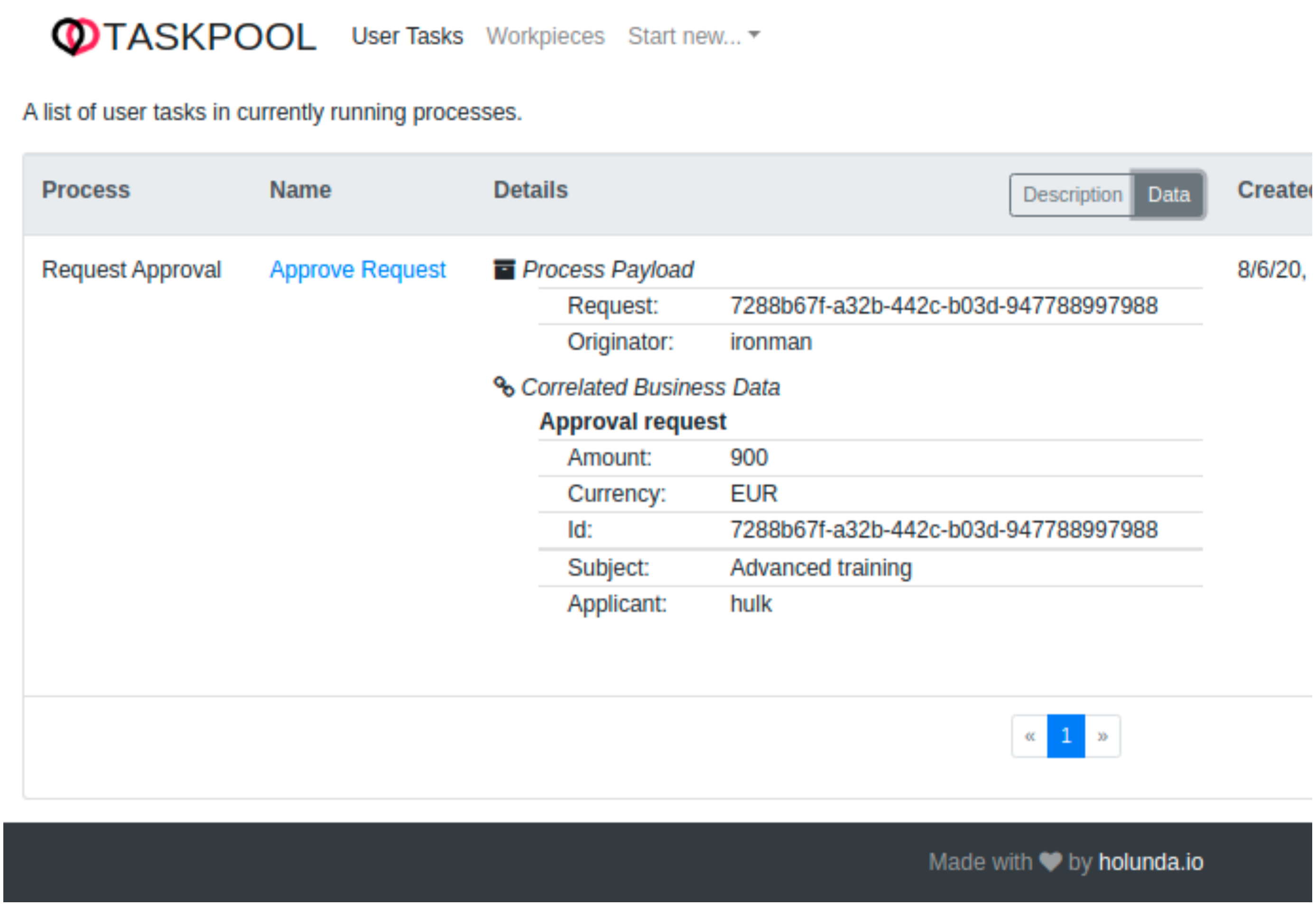


A list of user tasks in currently running processes.

Process	Name	Details
Request Approval	Approve Request	Please approve request 7288b67f-a32b-442c-b03d-947788997988 from ironman on behalf of hulk.

Made with ♥ by holunda.io

you can optionally show the business data correlated with user task:



A list of user tasks in currently running processes.

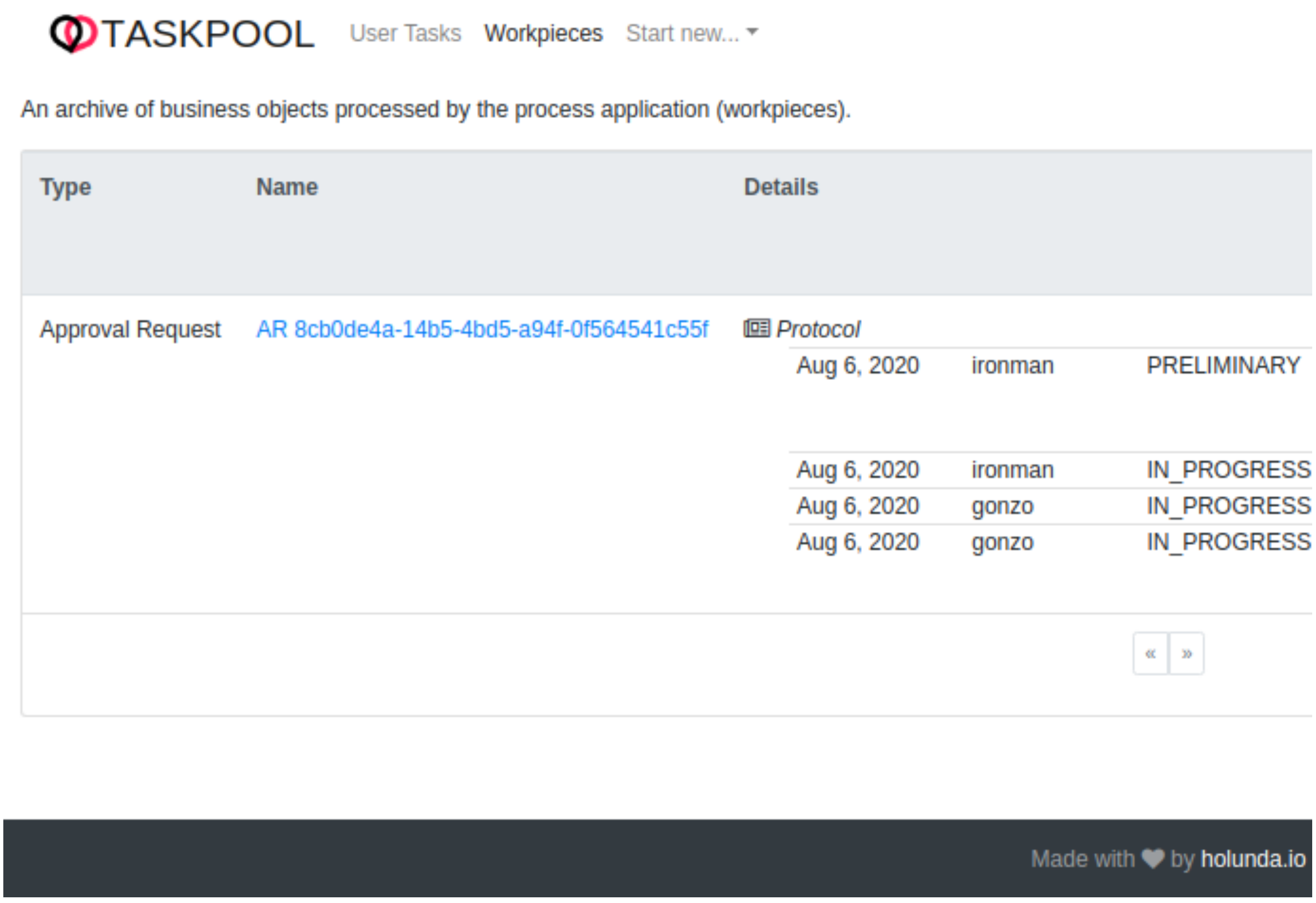
Process	Name	Details		Create
Request Approval	Approve Request	*Process Payload*		8/6/20,
		Request:	7288b67f-a32b-442c-b03d-947788997988	
		Originator:	ironman	
		Correlated Business Data		
		Approval request		
		Amount:	900	
		Currency:	EUR	
		Id:	7288b67f-a32b-442c-b03d-947788997988	
		Subject:	Advanced training	
		Applicant:	hulk	

Example Workpieces List

The example workpieces list is provides a list of business objects / workpieces that are currently processed by the processes even after the process has already been finished. It provides the following features:

- Lists business objects in the system for selected user
- Allows for switching users (faking different user login for demonstration purposes)
- 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 pageable (7 items per page)
- Business object view
- Provides a deeplink to the business object view provided by the process application

Here is, how it looks like showing the audit log:



An archive of business objects processed by the process application (workpieces).

Type	Name	Details
Approval Request	AR 8cb0de4a-14b5-4bd5-a94f-0f564541c55f	*Protocol*

Aug 6, 2020	ironman	PRELIMINARY
Aug 6, 2020	ironman	IN_PROGRESS
Aug 6, 2020	gonzo	IN_PROGRESS
Aug 6, 2020	gonzo	IN_PROGRESS

Process Platform Backend

Process Platform Backend

The purpose of the example process platform backend is to demonstrate how *process agnostic* parts of the process solution can be implemented.

Usage Scenarios

Usage Scenarios Overview

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 and connect them.

In doing so, 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 is 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

It is a good idea to understand the single node scenario first and then move on to more elaborated scenarios.

Scenario for running on a single node

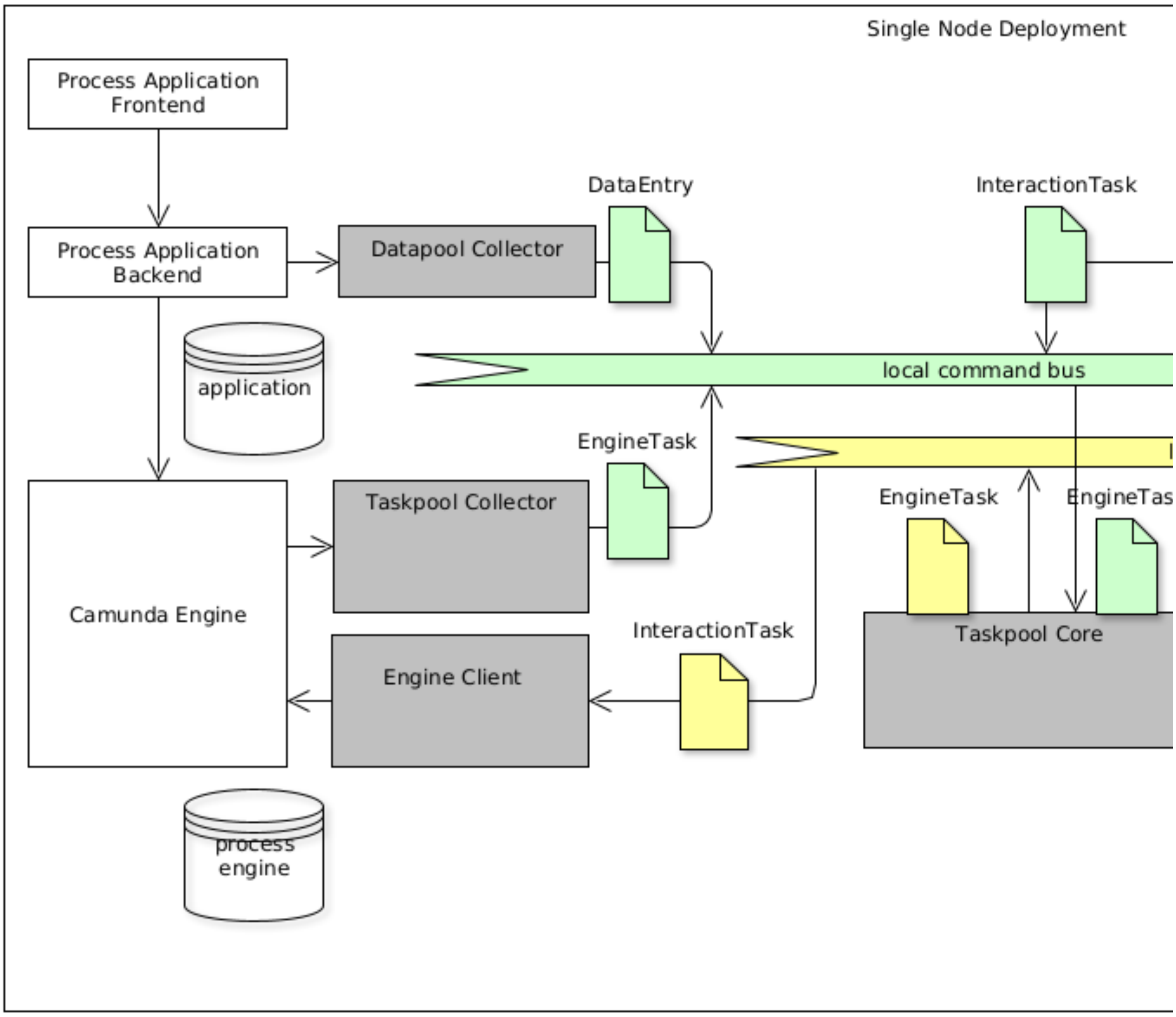
Scenario for running on a single node

In a single node scenario, the process application and the process platform components are deployed in a single node. In most production environments this scenario doesn’t make sense because of poor reliability. Still, it is valid for demonstration purpose and is ideal to play around with components and understand their purpose and interaction between them.

In a single mode scenario the following configuration is used:

- All buses are local (command bus, event bus, query bus)
- In-memory H2 is used as a database for:
  - Camunda BPM Engine
  - Axon Event Store (JPA-based)
  - Process Application Datasource
- In-memory transient projection view is used (`simple-view`)

Check the following diagram for more details:



Running Example

This example demonstrates the usage of the Camunda BPM Taskpool deployed in one single node and is built as a SpringBoot application.

System Requirements

- JDK 8

Preparations

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

Start

The demo application consists of one Maven module which can be started by running from command line in the `examples/scenarios/single-node` directory using Maven. Alternatively you can start the packaged application using:

```
java -jar target/*.jar
```

Useful URLs

- http://localhost:8080/taskpool/
- http://localhost:8080/swagger-ui/
- http://localhost:8080/camunda/app/tasklist/default/

Distributed Scenario using Axon Server

Distributed Scenario using Axon Server

A distributed scenario is helpful if you intend to build a central process platform and multiple process applications using it.

In general, this is the main use case for taskpool itself, but the distribution aspects adds technical complexity to the resulting architecture. Especially, following the architecture blueprint of Axon Framework, the three buses (command bus, event bus and query bus) needs to be distributed and act as connecting infrastructure between components.

Axon Server provides an implementation for this requirement leading to a distributed buses 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.

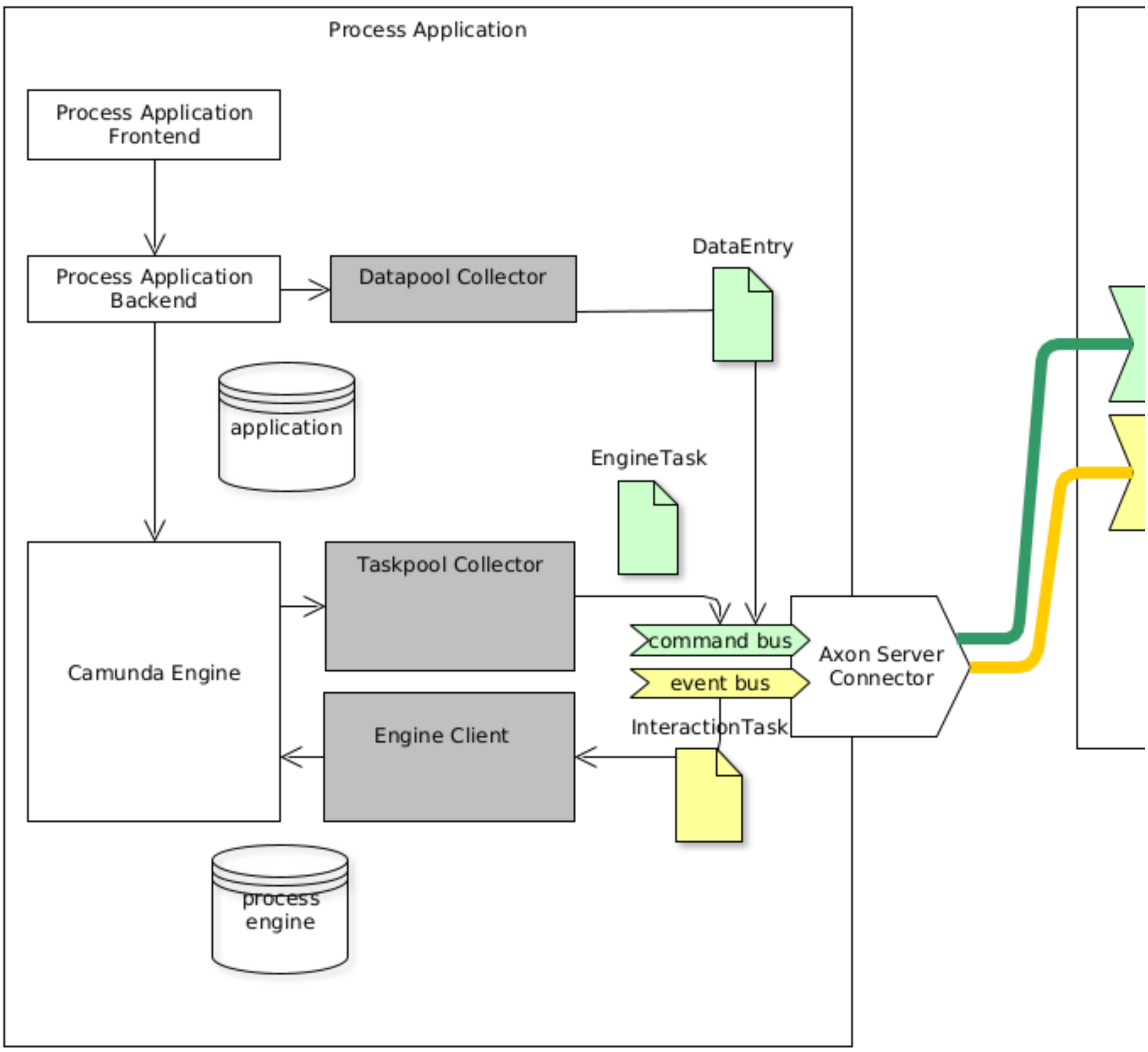
This scenario supports:

- central process platform components (including task pool and data pool)
- free choice for projection persistence (can be replayed)
- no direct communication between process platform and process application is required (e.g. via REST, since it is routed via command bus)

The following configuration is used in the distributes scenario with Axon Server:

- Bus distribution is provided by Axon Server Connector (command bus, event bus, query bus)
- Axon Server is used as Event Store
- Postgresql is used as a database for:
    - Camunda BPM Engine
    - Process Application Datasource
- Mongo is used as persistence for projection view (`mongo-view`)

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



Running Example

This example is demonstrating the usage of the Camunda BPM Taskpool with components distributed with help of Axon Server. It provides two applications for demonstration purposes: the process application and the process platform. Both applications are built as SpringBoot applications.

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 run a preparation script `.docker/setup.sh`. You can do it with the following code from your command line (you need to do it 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 (process platform)

2. process-application (example process application)

The modules can be started by running from command line in the `examples/scenarios/distributed-axon-server` directory using Maven or start the packaged application using:

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

URLs

Process Platform

- http://localhost:8081/taskpool/

- http://localhost:8081/swagger-ui/

Process Application

- http://localhost:8080/camunda/app/

- http://localhost:8080/swagger-ui/

Scenario without Axon Server

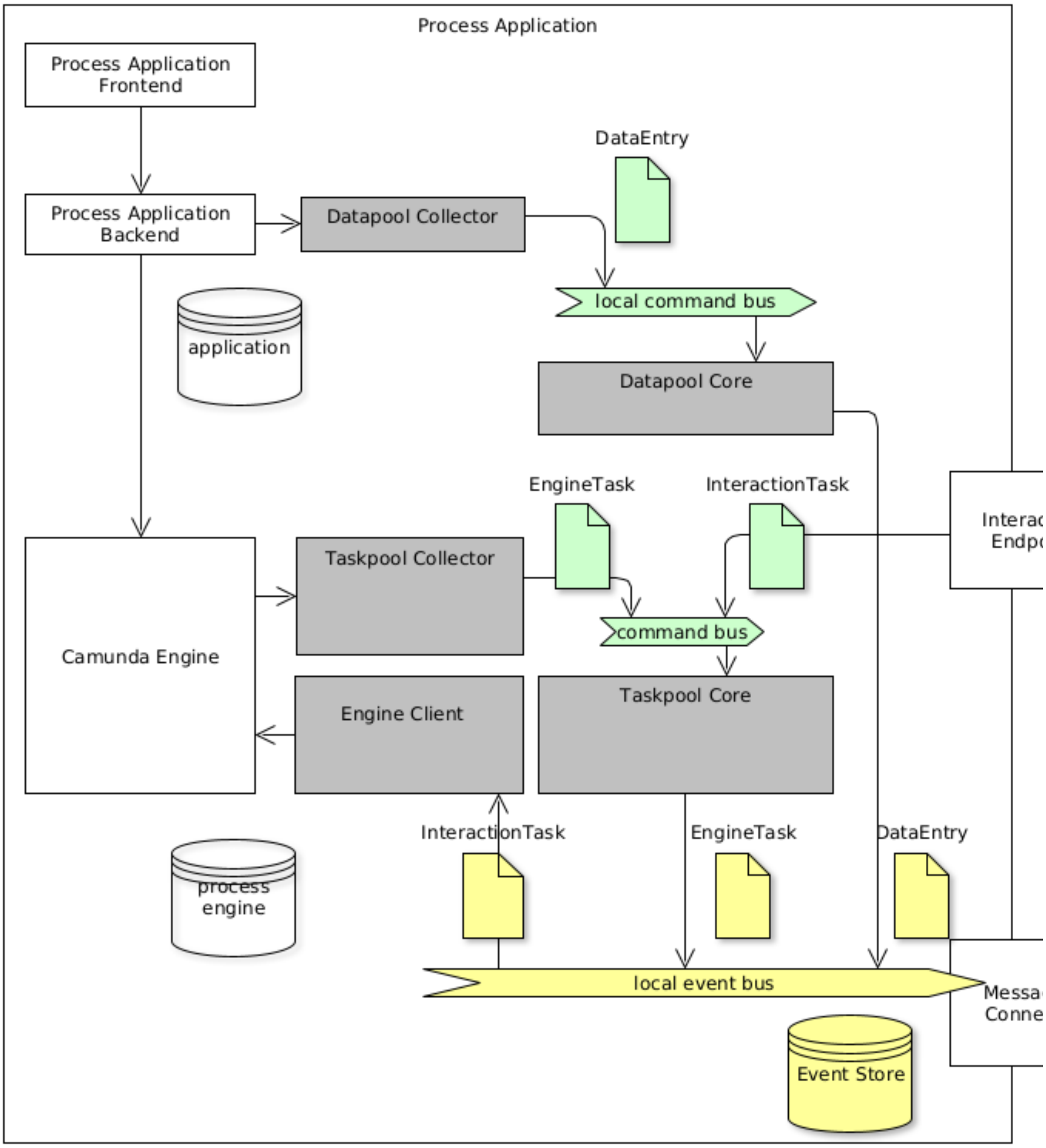
Scenario without Axon Server

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:

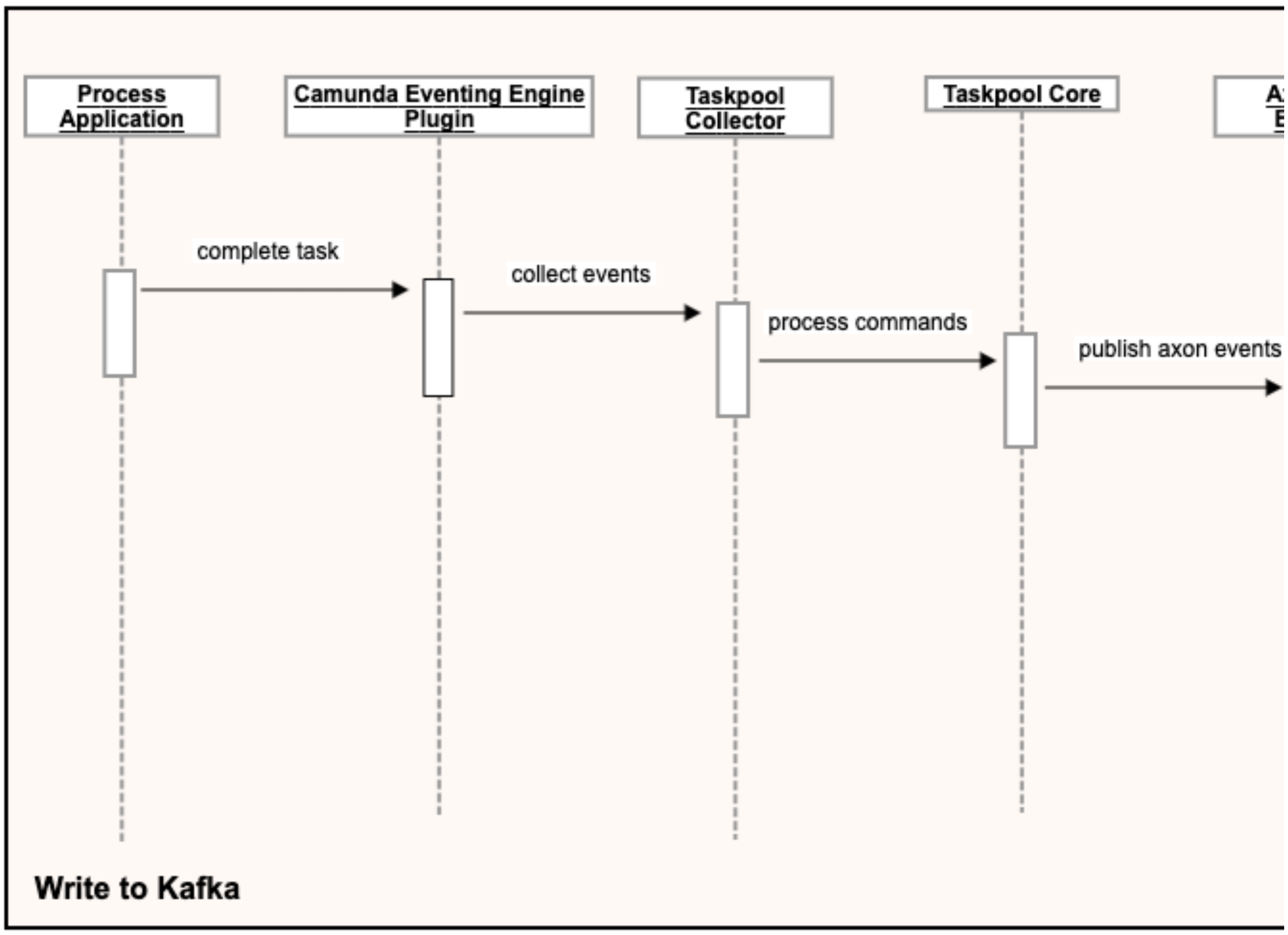
- distributed task pool / data pool
- view must be persistent
- direct communication between task list / engines required (addressing, routing)
- concurrent access to engines might become a problem (no unit of work guarantees)

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



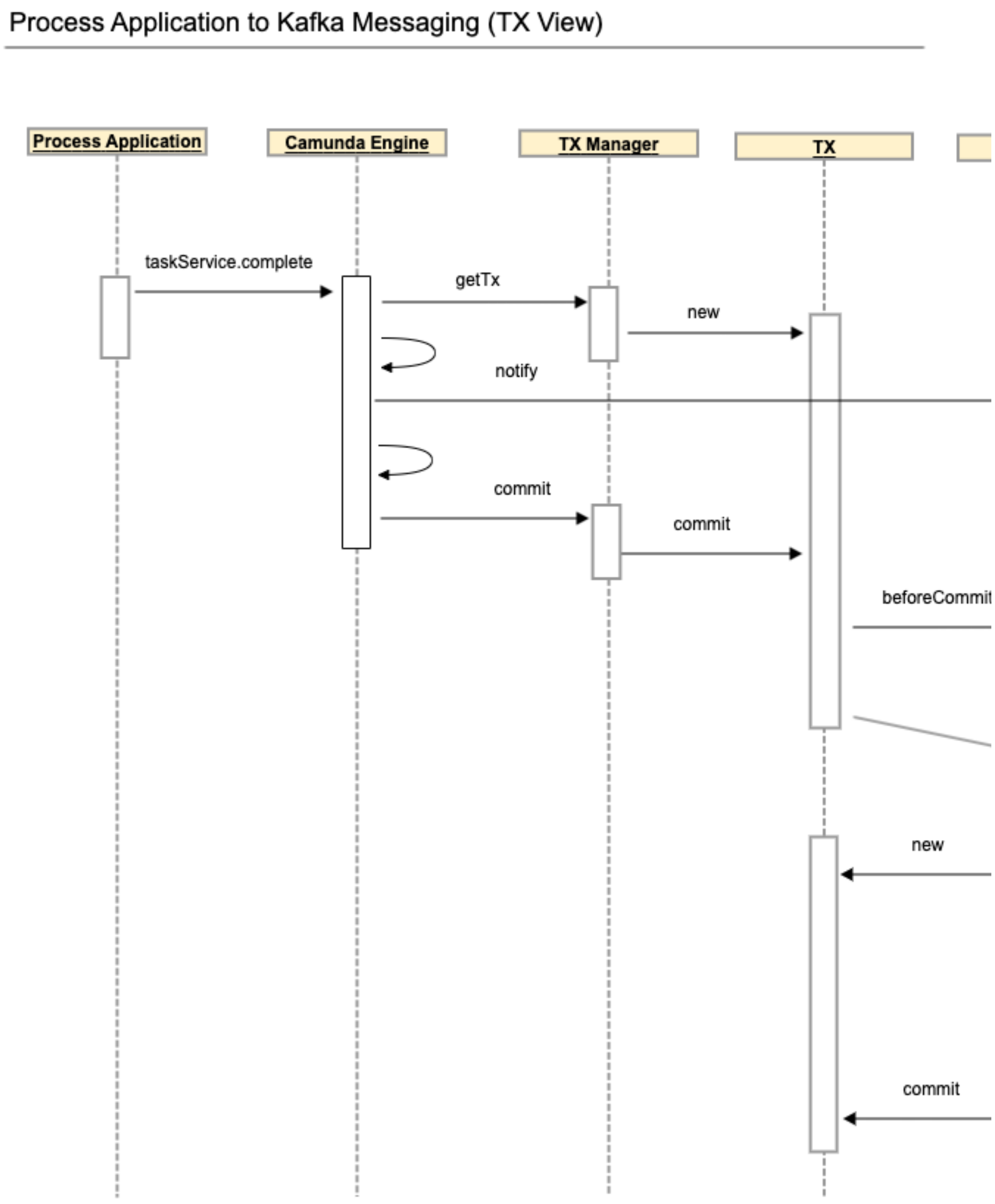
The following diagram depicts the task run from Process Application to the end user, consuming it via Tasklist API.

Process Application to Tasklist API Messaging (Top Level View)

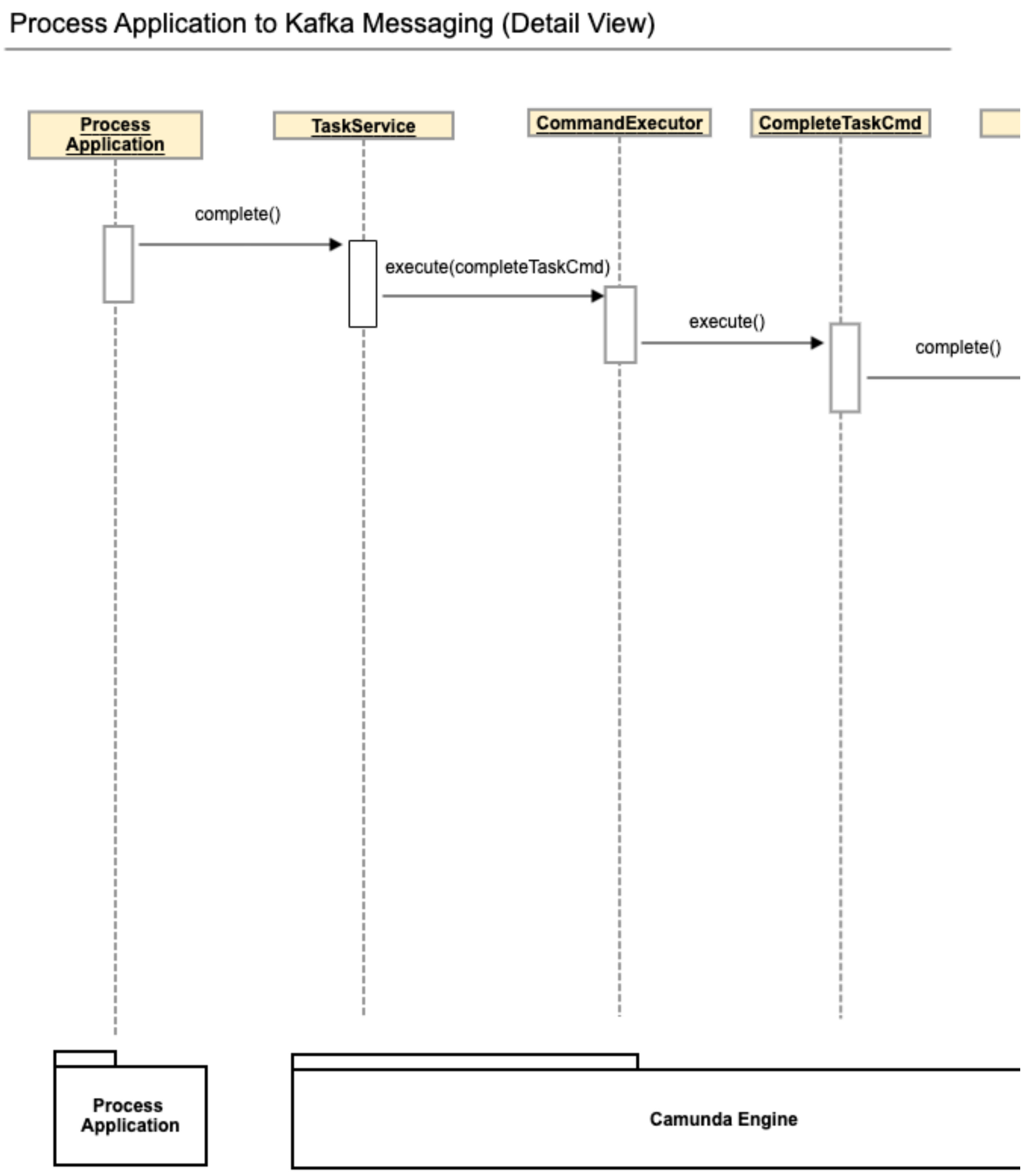


- The `CamundaEventingEnginePlugin` provided with the Taskpool tracks events in the Camunda engine (e.g. the creation, deletion or modification of a User Task) and makes them available as Spring events.
- The `Taskpool Collector` component listens to those events. It collects all relevant events that 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 `MongoViewService` 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.

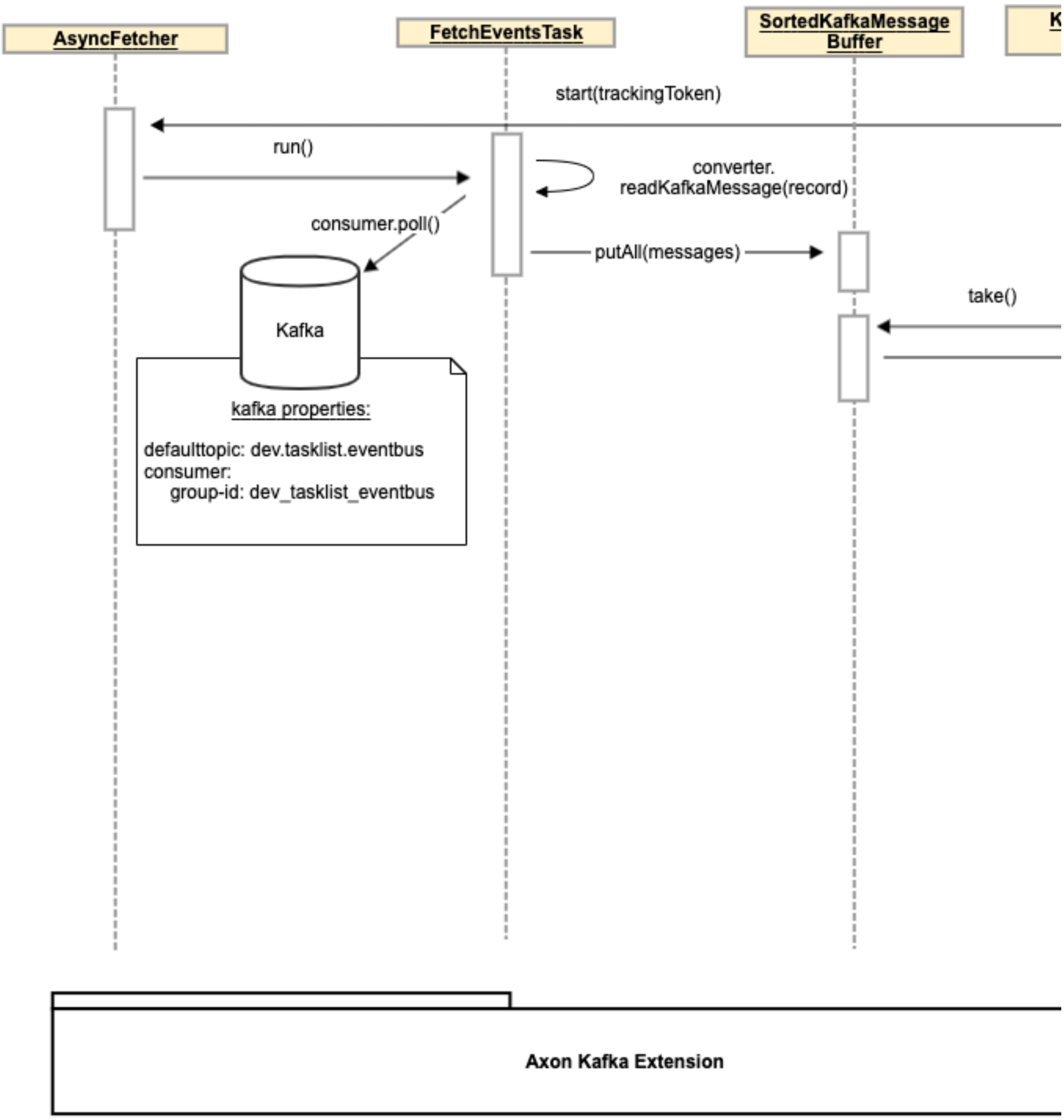


From Process Application to Kafka



From Kafka to Tasklist API

Kafka to Tasklist API Messaging (Detail View)



Integration Guide

Integration Guide

This guide is describing steps required to configure an existing Camunda Spring Boot Process Application and connect to existing Process Platform.

Configure your existing process application

Apart from the example application, you might be interested in integrating task pool into your existing application. To do so, you need to enable your Camunda BPM process engine to use the library. For doing so, add the `camunda-bpm-taskpool-engine-springboot-starter` library. In Maven, add the following dependency to your `pom.xml` :

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

Now, find your SpringBoot application class and add an additional annotation to it:

```
@SpringBootApplication
@EnableTaskpoolEngineSupport
public class MyApplication {

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

Finally, add the following block to your `application.yml`:

```
camunda:
  bpm:
    default-serialization-format: application/json
    history-level: full
  taskpool:
    collector:
      tasklist-url: http://localhost:8081/tasklist/
      process:
        enabled: true
      enricher:
        application-name: ${spring.application.name}  # default
        type: processVariables
      sender:
        enabled: true
        type: tx
    dataentry:
      sender:
        enabled: true
        type: simple
        applicationName: ${spring.application.name}  # default
    form-url-resolver:
      defaultTaskTemplate:  "/tasks/${formKey}/${id}?userId=%userId%"
      defaultApplicationTemplate: "http://localhost:${server.port}/${applicatioName}"
      defaultProcessTemplate: "/${formKey}?userId=%userId%"
```

Now, start your process engine. If you run into a user task, you should see on the console how this is passed to task pool.

For more details on the configuration of different options, please consult the Taskpool Components sections.

Taskpool Components

Components Overview

We decided to build a library as a collection of loose-coupled components which can be used during the construction of the process automation solution. In doing so, we provide *Process Engine Components* which are intended to be deployed as a part of the process application. In addition, we provide *Process Platform Components* which serve as a building blocks for the process platform.

Process Engine Components

Process Engine Components are designed to be a part of process application deployment and react on engine changes / interact with the engine. These are:

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

Process Platform Components

Process Platform Components are designed to build a process platform.

Core Components

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

- 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

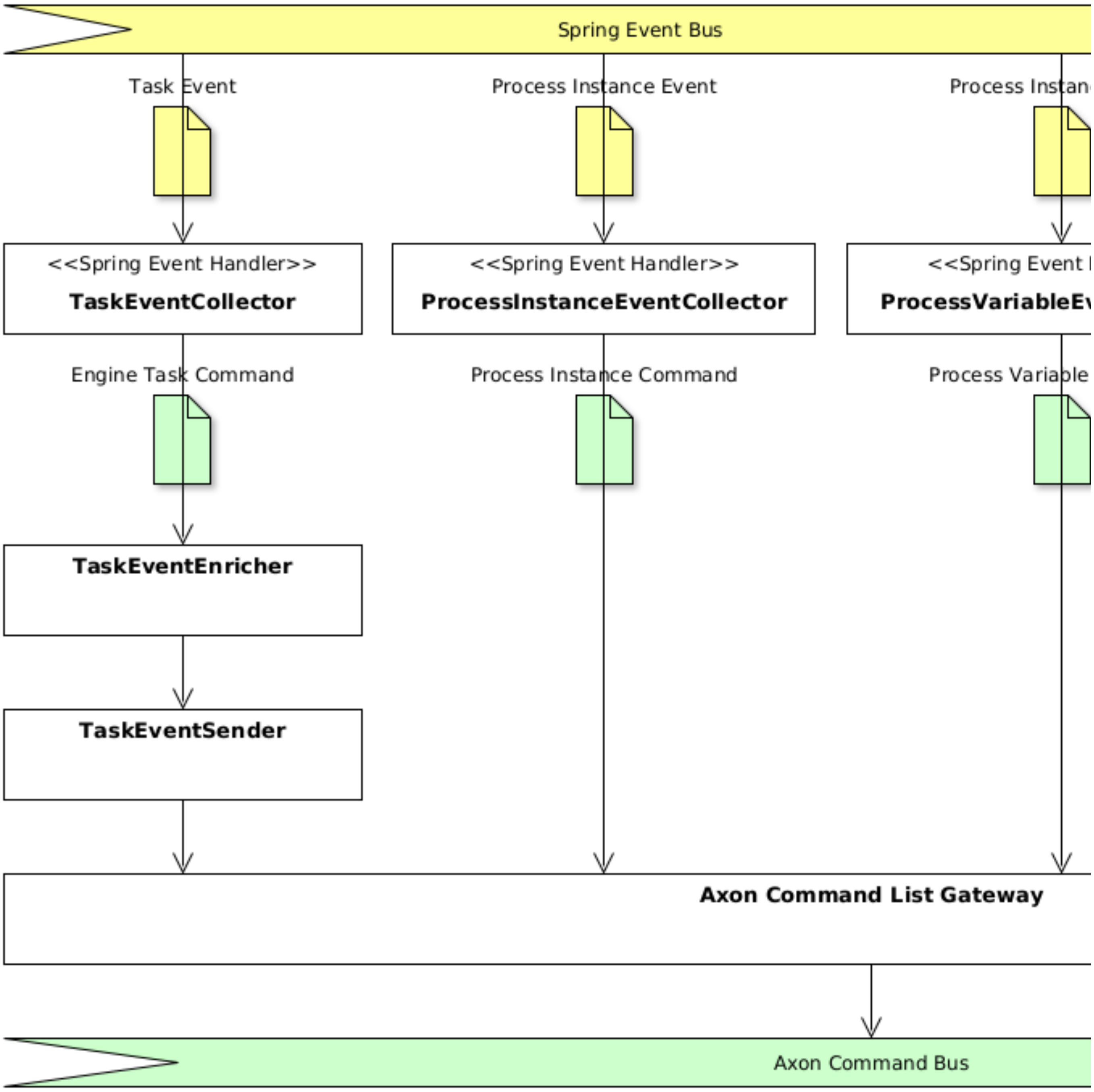
Taskpool Collector is a component deployed as a part of the process application (aside with Camunda BPM Engine) that is responsible for collecting information from the Camunda BPM Engine. It detects the *intent* of the operations executed inside the engine and creates the corresponding commands for the taskpool. The commands are enriched with data and transmitted to other taskpool components (via Axon Command Bus).

In the following description, we use the terms *event* and *command*. Event denotes an entity received from Camunda BPM Engine (from delegate event listener or from history event listener) which is passed over to the Taskpool Collector using internal Spring eventing mechanism. The Taskpool Collector converts the series of such events into a Taskpool Command - an entity carrying an intent of change inside of the taskpool core. Please note that *event* has another meaning in CQRS/ES systems and other components of the taskpool, but in the context of Taskpool collector an event alway originates from Spring eventing.

Features

- Collection of process definitions
- Collection of process instance events
- Collection of process variable change events
- Collection of task events and history events
- Creation of task engine commands
- Enrichment of task engine commands with process variables
- Attachment of correlation information to task engine commands
- Transmission of commands to Axon command bus
- Provision of properties for process application

Architecture



The Taskpool Collector consists of several components which can be devided into the following groups:

- Event collectors receive are responsible for gathering information and form commands
- Processors performs the command enrichment with payload and data correlation
- Command senders are 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
@EnableTaskpoolCollector
class MyProcessApplicationConfiguration {

}
```

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
  - HistoricProcessInstanceEventEntity
  - HistoricVariableUpdateEventEntity
  - HistoricDetailVariableInstanceUpdateEntity

The events are transformed into corresponding commands and passed over to the processor layer.

Task commands enrichment

Alongside with attributes received from the Camunda BPM engine, the engine task commands can be enriched with additional attributes.

There are three enrichment modes available controlled by the `camunda.taskpool.task.collector.enricher.type` property:

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

Process variable enrichment

In particular cases, the data enclosed into task attibutes is not sufficient for the 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, active it setting the property `camunda.taskpool.collector.task.enricher.type` to `process-variables` and the enricher will put process variables into the task payload.

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). If the filter is not provded, a default filter is used which is an empty `EXCLUDE` filter, resulting in all process variables being attached to the user task.

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 defined in the filter.
  - `EXCLUDE`: task-level exclude filter, denoting a list of variables to be ignored for the task defined in the filter. 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.task.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 data correlation allows to attach one or several references (that is a pair of values `entryType` and `entryId`) of business data entry(ies) to a task. In the projection (which is used for querying of tasks) this correlations is 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` bean. Here is an example how this can be done:

```
@Bean
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 for all tasks or for an individual taskby providing a so-called correlation map. A correlation map is keyed by the name of a process variable inside Camunda Process Engine and holds the type of business data entry 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 `io.my.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
fun processVariablesCorrelator() = ProcessVariablesCorrelator(

  ProcessVariableCorrelation("process_approval_process",
    mapOf(
      "task_approve_request" to mapOf(
        "varRequestId" to "io.my.approvalRequest" // process variable 'varRequestId' hol
      )
    )
  )
)
```

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 aggregation

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

In addition you can control by the property `camunda.taskpool.collector.task.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` for sending the result over to the Axon command gateway.

Tip If you want to implement a custom command sending, please provide your own implementation of the interface `EngineTaskCommandSender` (register a Spring Component of the type) and set the property `camunda.taskpool.collector.task.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 component and enricher is performed synchronously, 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 gateway. The property `camunda.taskpool.collector.task.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 this purpose, please provide a Spring Bean implementing the `CommandSuccessHandler`and `CommandErrorHandler` accordingly.

Here is an example, how such a handler may look like:

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

Configuration properties overview

Table 1. Title

Name	Purpose	Default
Cell in column 1, row 1	Cell in column 2, row 1	Cell in column 3, row 1
Cell in column 1, row 2	Cell in column 2, row 2	Cell in column 3, row 2

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.

Configuration

Component activation

In order to activate Datapool Core component, please include the following dependency to your application

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

and activate its configuration by adding the following to a Spring configuration:

```
@Configuration
@EnableDataPool
class MyConfiguration
```

Revision-Aware Projection

The in-memory data entry projection is supporting revision-aware projection queries. To activate this, you need to activate the correlation of revision attributes between your data entries commands and the data entry events. To do so, please activate the correlation provider by putting the following code snippet in the application containing the Datapool Core Component:

```
@Configuration
@EnableDataPool
class MyConfiguration {

  @Bean
  fun revisionAwareCorrelationDataProvider(): CorrelationDataProvider {
    return MultiCorrelationDataProvider<CommandMessage<Any>>(
      listOf(
        MessageOriginProvider(),
        SimpleCorrelationDataProvider(RevisionValue.REVISION_KEY)
      )
    )
  }

}
```

By doing so, if a command is sending revision information, it will be passed to the resulting event and will be received by the projection, so the latter will deliver revision information in query results. The use of `RevisionAwareQueryGateway` will allow to query for specific revisions in the data entry projection, see documentation of `axon-gateway-extension` project.

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 gor
    "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"
}
```