UNIVERSITY OF GRONINGEN

FACULTY OF SCIENCE AND ENGINEERING
DEPARTMENT OF COMPUTING SCIENCE

Design and Implementation of a Process-Aware System

for Flexible Batch Processing

Author
Dimitrios LASKARATOS

Supervisor
pr. dr. Dimka KARASTOYANOVA

university of groningen

faculty of science and engineering

Contents

List of Figures

List of Tables

Listings

Abbreviations

Abbreviation	**Meaning**
BPM	Business Process Management
BPMN	Business Process Model and Notation
BPMS	Business Process Management System

Abstract

Business Process Management (BPM) is important to the day-to-day operations of large organizations that constantly seek to optimize their workflow. BPM systems (BPMS) are numerous and offer many diverse services to these organizations, based on their needs. However, these systems all have one thing in common: they do not perform batch processing. Even though there is an abundance of literature approaching this problem theoretically along with several implementations, they are hindered by the fact that they are static and limited in their flexibility. This project attempts an implementation of a flexible batch processing that is independent of both the modelling and the execution phase of BPM. This implementation is a system on its own, involving multiple components that interact with each other, as well as the engine. The results after modifying the target BPMS are promising, showing that the flexibility aspect discussed is certainly possible. Concurrently, it offers a view on extensive future work on several other aspects, like further extending the batch components in order to generalize, i.e., to apply to all types of activities in a process.

1 Introduction

I Models, Process Engines and Batch Processing

Business Process Management is an integral part of large organizations, whose operations often require multiple actions from several different departments to carry out a complex task. This complexity is not always visible from a high-level view of the task, which hides the intertwined activities needed for its completion. In time, along with the growth of the organization, the task may evolve and become even more complex. This may lead to several sub-activities to need to change or even become obsolete, but the scale of the task itself might make it hard to detect minor details. Eventually, the time and cost of the task will increase for the organization. The goal of BPM is to not only document the processes that are crucial for the completion of the task and analyze their behaviour, but also discover processes that are running in the background, and require additional resources, potentially preventing the task from completing faster [4]. The main artifact of BPM is the conceptual model, depicting the life cycle of the target process from the moment it is initiated, to the moment it is completed. This model is constructed using the Business Process Model and Notation (BPMN) format. BPMN is based on graphical flowcharts and consists of five core elements, namely the "start" and "end" symbols, the "gateway" symbol, the "activity" symbol and the "sequence flow" symbol. A brief BPMN example, greatly resembling UML, is visible in Figure 1, depicting a simple ordering and shipping of a product. The "start" and "end" events are the moment the order is placed and the moments the order is either rejected or accepted, respectively. The rectangles represent the activities that take place during the process, which might as well be processes themselves, and the arrows dictate the sequence of events in the process. Lastly, the diamond shape is the "gateway" that can be either a decision making point or a point where two separate activities must take place in parallel. In both cases, there are multiple execution paths possible. As mentioned, the complexity of the model depends on the complexity of the process itself, but can also depend on the level of detail that the modeler requires. As a result, a process model may scale up to hundreds of internal activities , sub-processes and possible execution paths. Additionally, other entities (processes, external services) that communicate with the process might also have to be included, adding to the total complexity of the model. This can be compared to a program with multiple execution paths, that can spawn procedures, each with their own execution paths, eventually resulting in several different outputs.

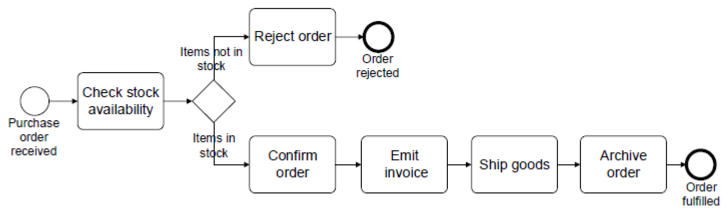
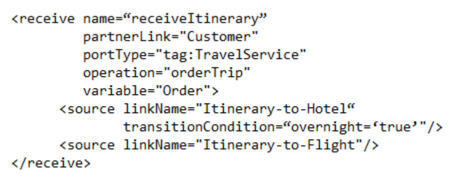


Figure 1: BPMN process model



```
<receive name=“receiveItinerary”
        partnerLink="Customer"
        portType="tag:TravelService"
        operation="orderTrip"
        variable="Order">
    <source linkName="Itinerary-to-Hotel“
            transitionCondition=“overnight=‘true’"/>
    <source linkName="Itinerary-to-Flight"/>
</receive>
```

Figure 2: BPEL format

The process model can only offer static analysis of the processes and resources involved, which is why the next step is to submit this model to a business process engine for execution. The modelling tool, which in most cases is engine specific, generates an XML-like file that can be parsed by the engine to create executable code out of its semantics. It is worth mentioning that in some cases, engines generate a similar representation of the elements called Business Process Execution Language (BPEL) which is visible in Figure 2. BPEL is used to dictate the interaction of specific elements (eg. an activity) of the process workflow with external web services.

Batch Processing

An important part of BPM is the notion of batch processing [5] which is very common in day-to-day operations of organizations. Batch processing stems from the idea that several instances of a process may exist at the same time, which, by definition, require the same resources for their operation. As a result, these instances will eventually have to be executed at the same activity, provided a normal execution (i.e. no faults during execution that can result in different activities) takes place. In such a case, the multiplicity of the process will result in the commitment of a large number of these resources, as well as increase the time needed to conclude the execution of all the processes [6]. That is, if the execution is conducted sequentially. By using batch processing, the process instances are aggregated in groups called "batches" right before a certain activity starts, in order to execute them as a group. This reduces the cost and processing time compared to when these processes are handled individually [7]. Consider the example of blood testing in a medical center. The blood samples taken from the patients must be carried to the laboratory for testing. It doesn’t make sense for the employee to take each new sample to the lab individually because that would be a waste or time. Instead, the employee waits until a number of samples have been accumulated before submitting them to the lab. The same idea applies to BPM where the main process execution halts until the required conditions have been met to begin group processing [2].

From a modelling perspective it is sufficient to only denote the batch activity in the process model in BPMN as shown in Figure 3 [8]. However, the BMP engine that parses the process models in preparation for execution, have to also be extended accordingly to parse the batch region notation. Additionally, the activation of a batch cluster, meaning, the actual execution of the batch needs to meet the conditions mentioned above. [7] and [2] have proposed as a threshold either a time constraint, eg. one hour before the batch is executed anyway, or the number of instances needed to be aggregated before execution, eg. 10 or 20. Further information on that work can be found in the Related Work section.

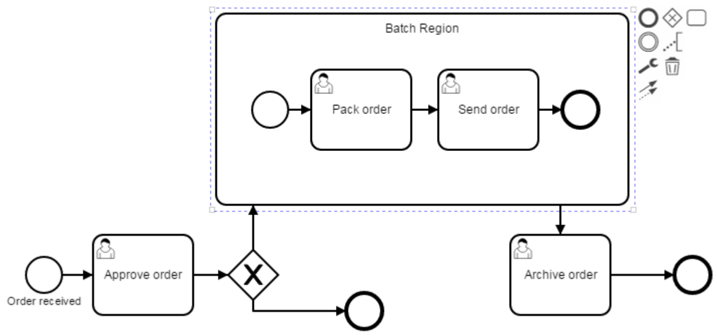


Figure 3: Batch region in Camunda Modeller modeller batch extension by Pufahl et al. [1]

II Related Work

Existing BPM engines do not actually implement batch processing and are limited to a static and sequential execution and resource handling. While the idea of batch processing has been around for quite long [6], only in recent years several attempts have been made to introduce batch processing in BPM engines, either on a theoretical level or on a practical one.

Weske at al. [2018] [6] describes in depth the prototypical implementation of a batch extension by [1], from both a theoretical and practical point of view. Several requirements that need to be satisfied in order for the extension to be successful are identified:

1. **Batch processing over several activities**: Batching must be able to be applied to a set of connected activities instead of just one.

2. **Grouping**: Cases must be able to be grouped in batches based on a common characteristic.

3. **Resource capacity**: A restriction must be put on the number of process instances that can be aggregated before batched execution.

4. **Batch assignment**: Process instances whose number has exceeded the maximum allowed for batch execution must be assigned to different batches.

5. **Activation mechanism**: A mechanism is needed to define the optimal activation time for a batch job to start.

6. **Execution strategy**: A decision has to be made about whether the manner of execution of the batch is parallel or sequential.

Karastoyanova et al. [2018] [2] serves as the basis for this work, as it provides the theoretical background of the batch processing extension. This work discusses three strategies that can support batch processing on an engine and then proposes three flexibility needs that the extension should implement. These three strategies require certain artifacts in order to work. These artifacts are the process model, the batch model and the connector, the latter in some cases. The batch model is a file that specifies the conditions on which a batch cluster is activated and the connector defines the relations between a batch model and a process activity. Lastly, a system architecture to further support the previous is provided.

The three strategies are the *Modelling strategy*, the *Deployment strategy* and the *Execution strategy*. The Modelling strategy states that the batch region of the process is defined during the modelling phase on the process model, as shown on Figure 4. The model is then deployed on the engine for execution and once control reaches the batch activity, it is executed based on the predefined conditions of the batch model. This strategy is the simplest of the three since the requirements it imposes on the system are minimal compared to the other two. On the other hand, any change in the batch configuration requires the process to be modelled again and redeployed.

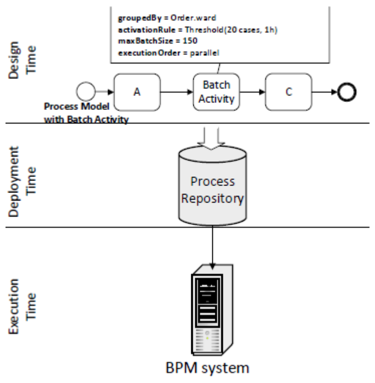


Figure 4: Illustration of the modelling strategy [2]

In the Deployment strategy the batch region is no longer defined in the process model file, but is instead specified in the connector artifact, completely separate from the model file. The process model, the connector and the batch model all have to be stored in the BPM repository as a group as shown in Figure 5. The requirements on the system are more extensive compared to the Modelling strategy, but using the Deployment strategy, the modularity, reusability and flexibility of the artifacts are improved. For example, the modeller needs not be concerned with specifying batch regions or any of the batch configuration since these are stored in separate files. Per contra, changes to the batch model require it to be modified and redeployed which eventually decreases the flexibility of the system and has to be avoided.

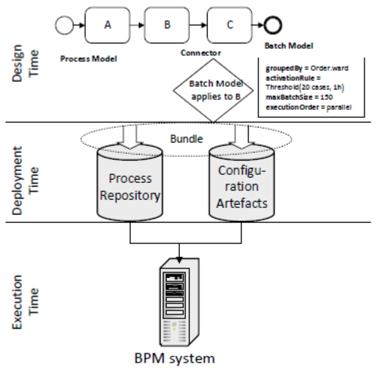


Figure 5: Illustration of the deployment strategy [2]

In the Execution strategy, as with the Deployment strategy, the process model, the batch model and the connector have to be generated by the modeller, but are not deployed on the BPM system as a bundle during deployment. Instead, each artifact can be submitted separate from the others and/or at different times. For example, a batch model can be assigned to a process after the process has been deployed on the BPM, or even during its execution. An illustration can be found in Figure 6. Since a batch model can be assigned and connected to multiple activities across several process instances in real time, it negates the previous disadvantage of the Deployment strategy. Additionally, it is possible to remove said connections during execution. The system requirements for this strategy are again more extensive, but the three flexibility needs are supported.

Note that the connector artifact presented above might seem as an entity that needs to be handled as the batch model. In reality, it is an abstract concept that is merely as a link between a process, an activity and a batch model. It's role will become more clear in Section 2.

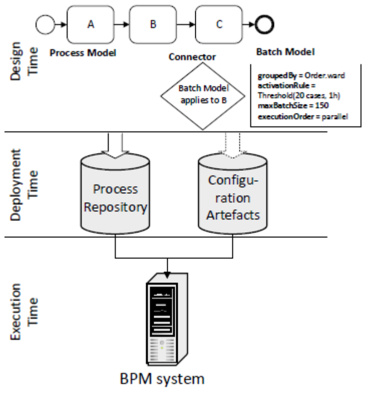


Figure 6: Illustration of the execution strategy [2]

The flexibility needs for batch processing as identified in the work of [2] are as follows:

- **Evolution**: Either the process model and/or the batch configuration must be able to change (eg. due to business changes) without the need of redeployment of all artifacts.

- **Looseness**: Batch configuration must be decoupled from the modelling phase to allow certain aspects of batch work to remain undefined if need be.

- **Adaptation of Batch Activity**: Batch configuration elements must be adaptive to react to exceptions during process execution or other reasons.

In the context of [2], a system architecture to support the batch extension is provided in Figure 7. This architecture will be used as a basis of this work. Note that this architecture contains a high level view of the components involved and the role of each will be thoroughly discussed in Section 2

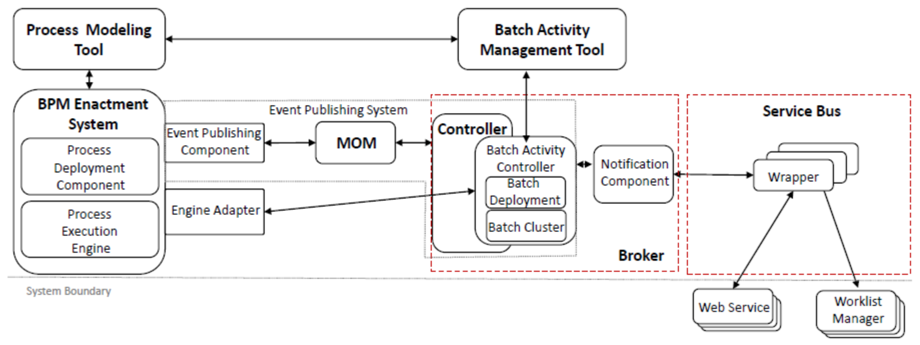


Figure 7: High level architecture of batch processing extension [2]

In **Pufahl [2018] [1]** there exists a prototypical concrete implementation of a batch extension that largely follows the Modelling strategy presented in [2] and briefly discussed previously. In this work the batch properties have to be specified in the process model and deployed together on the Camunda Engine. This implementation applies only to activities of type User Task, where the process instances are aggregated based on the user assigned to the task.

III Present Work

The purpose of this thesis is to extend a BPM engine so that it can support flexible batch processing based on the work of [1] and [2]. More specifically, the goal is to combine the theoretical background of [2] and, possibly, elements of the Java implementation of batch processing of [1] to enable the BPM engine to support real-time manipulation of the batch artifacts, by taking into account the flexibility needs, strategy and architecture that have been discussed previously. Because of time constraints this implementation will only take into account activities that execute an automated script. In the context of Camunda this is equivalent to invoking a web service.

In Section 2 we will deal with the platform selection criteria, followed by a brief explanation of the workflow components, and lastly, the architecture and the implementation details of the extension. In Section 3 we will present the results based on case studies conducted with the new implementation and subsequently discuss them in Section 4. Finally, in Section 5 we will discuss possible future research opportunities based on the results obtained and the current landscape of BPM.

2 Architecture and Implementation

I Platform Selection

There are several Workflow Management Systems (WfMS) that are currently supporting the latest version of BPMN (version 2.0) [9]. Of the ones whose source code is publicly available, two are selected with which the writer has had substantial experience. These are Camunda BPM Community version and Apache ODE. Apache ODE proved difficult to use because of the limited support and documentation that was available online and, additionally, needed Eclipse as a platform which is widely considered outdated and unreliable. Considering the fact that Camunda was the basis with which Dr. Pufahl [1] developed the batch extension and that a large and active community of users is maintained online that can provide support, is was decided that Camunda will also be used for the flexible batch processing implementation.

I.1 Camunda BPM System Architecture

Since Camunda will serve as a host platform for the extension, we will describe the high-level architecture of the system, component-wise, before moving on to explain the extension components.

The first part of the process life cycle is the modelling tool, which, in most cases produces an XML file with the .bpmn extension. Other possible extensions include CMMN (Case Management Model and Notation) and DMN (Decision Model and Notation). Camunda comes with its own modelling tool called Camunda Modeller that offers additional features, such as specifying user or automated tasks, that are specific to the engine and the applications of the system. In Figure 8 this applies to the "Modeller".

A distinct characteristic of the Camunda Modeller is that it enables the user to specify the behaviour of the activities during the modelling phase as well as any variables that are manipulated during the process workflow. A notable example, which will be described in detail later, involves the creation of an activity that requires the execution of Java code. This activity is called a "Script Task". In order for Camunda to execute the Java code specified by this activity, both the process model and the Java class have to be built together and packaged in a .war file. This procedure is called Process Definition Deployment.

As stated in Section 1, this .war file is then uploaded to a File Repository which is accessed by the Engine itself, which parses the .bpmn file, translating the BPMN artifacts and other semantics into Java objects, internal to the engine. The engine is a Java application that uses the REST API, on top of an Apache Tomcat server to communicate with external applications. The engine handles all aspects of the process life-cycle, including starting and stopping a process, and provides an API for such manipulations by the users. The engine uses the H2 database to store process entities by default, but can also be configured to use additional ones like DB2, MySQL, and others.

Finally, user, technical and administrative tasks are accessible from a range of Angular.js web pages, provided through the Tomcat server. These applications offer a variety of tools like a list of tasks for the end user, details regarding the current state of process instances, the ability to start multiple process instances, adding or deleting users and many others. These applications, along with any other external Java application (eg. web service) can invoke engine routines through the REST API.

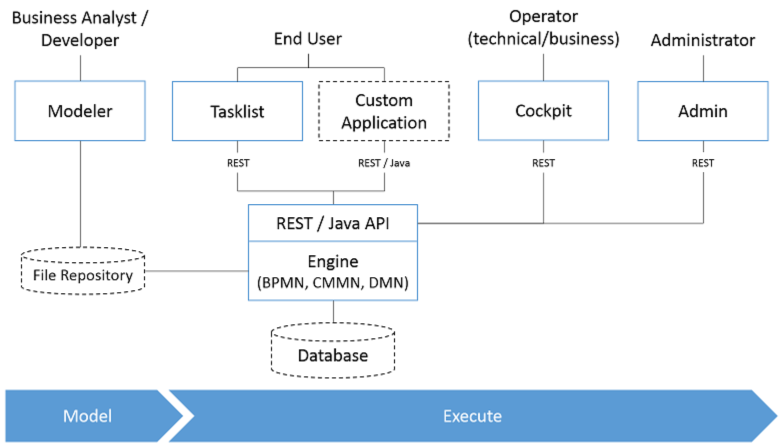


Figure 8: Camunda components [3]

I.2 Camunda Basic Workflow

In Figure 12 presented is the basic workflow for the creation, deployment and execution of a process in Camunda Engine. The first step is to create the process in the modelling tool as shown in (1), using the BPM Notation discussed previously. Optionally, there is the possibility to specify the starting variables by selecting the start event (2.1) and subsequently add variables by specifying their name and type (2.2). Lastly, when the process contains script activities as in the Figure, it is mandatory to bind a script to it. In this case a Java class is bound to activity "Calculate Value", by selecting the activity (3.1) and then providing the full path of the Java class (3.2).

By building the process model classes bound to the activities in an IDE using the Camunda maven archetypes, a .war file is generated as shown in (4.1). When this file is copied to the "webapps" folder of the running Apache Tomcat server, the engine deploys the application as shown in (4.2).

By opening the Cockpit web application of Camunda, the deployed process definition is now visible (5). In order to start an instance of this process, the user must then go to the Tasklist application and select the process from the list as shown in (6.1). If the process specified starting variables, the values have to be provided in the form in (6.2). By pressing the "Start" button, the engine starts an instance of the process.

Going back to Cockpit and selecting the process from the list, the started instance is now visible, along with its current state and variables (7).

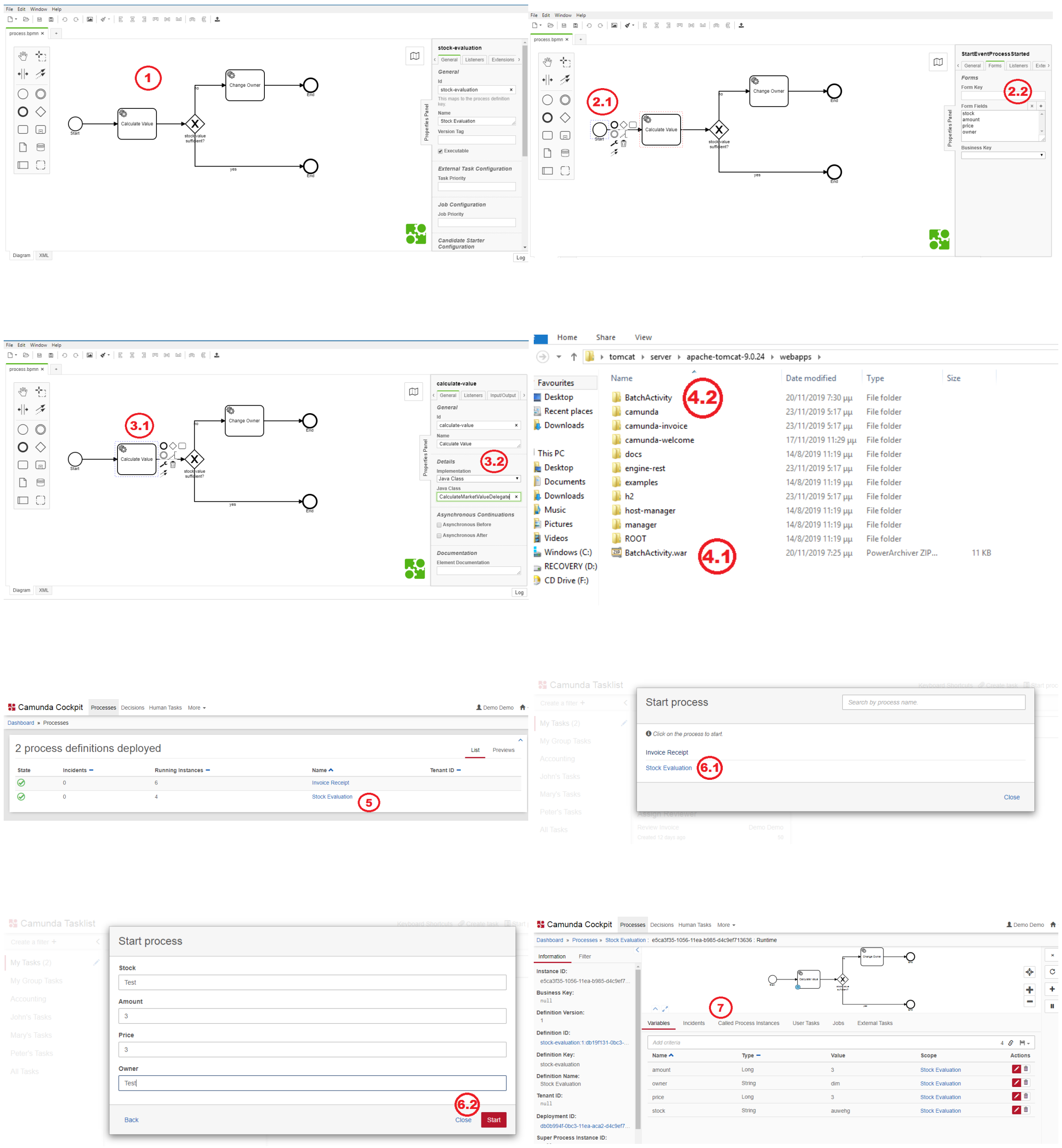


Figure 12: (1): Process model - (2.1): Start event - (2.2): Start variables - (3.1): Activity - (3.2): Script binding - (4.1): Generated .war file - (4.2): Process deployed - (5): Process in Cockpit - (6.1): Deployed process in Tasklist - (6.2): Process instantiation - (7): Process instance

II Extension Architecture

In this section the architectural components will be discussed in greater detail.

II.1 Overview

As stated in Section II, the high-level architecture presented in Figure 7 will be the backbone of the component architecture described in detail in this section. Since the architecture presented in Figure 7 does not refer to a specific BPMN engine, certain adjustments are necessary in order for this work's architecture to apply to Camunda. The proposed and implemented architecture is visible in Figure 13. It consists of three major components, namely the Camunda engine itself, the Batch Controller and the User Interface. It is complimented by a message queue that is implemented by a RabbitMQ node, and a MySQL instance database for storing the batch properties and conditions. The Web Service at the right of the figure is not part of the architecture, but is merely there to infer that the Batch Controller may potentially interact with third-party services.

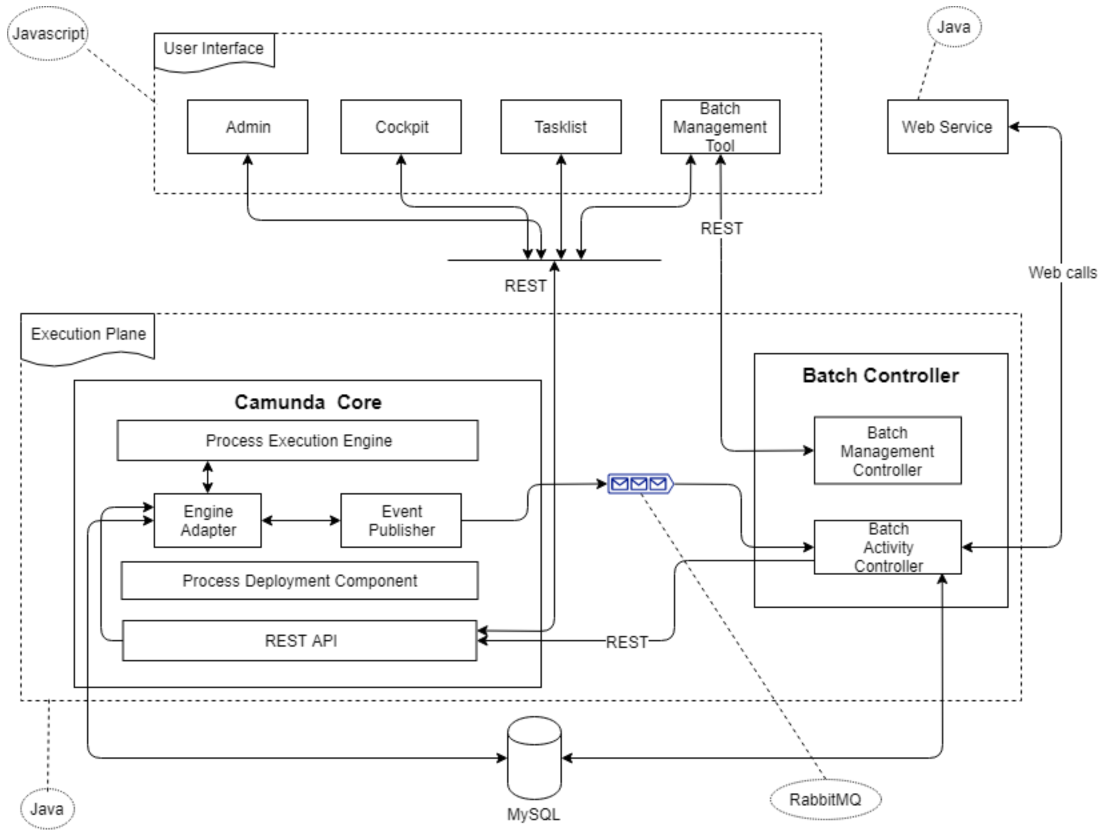


Figure 13: Extension Architecture

II.2 Engine

The software that Camunda is made up of, consists of multiple packages, the most important of which are the Process Engine and REST API. In Figure 13 it may seem that Camunda Core contains many different modules that interact with each other. In reality, only the Process Execution Engine and REST API are separate modules, while the remaining are included in the Process Execution Engine part. This separation in the figure is only done for illustration purposes, as there is no physical distinction between these entities.

The **Process Execution Engine** is responsible for virtually every operation that is needed in Camunda. It parses newly deployed processes, starts, runs or suspends processes, tracks the state of every running process and manages their entire life-cycle. It is a complex and large piece of Java code that conducts many internal calls and interacts with databases and the underlying server (Apache Tomcat, JBoss, Wildfly). Since Camunda does not implement batch processing or anything closely related, and it is also not possible to use its

REST API to halt process instances, modification of the engine code is necessary.

The **Engine Adapter** is a sub-module of the Process Engine that contains the functionality to handle process instances before and after batch execution. The Engine Adapter directly interacts with Process Engine elements, the REST API and the Event Publisher. A brief explanation of the Engine Adapter's function is presented in Figure 14. A more detailed explanation is available in Section III.

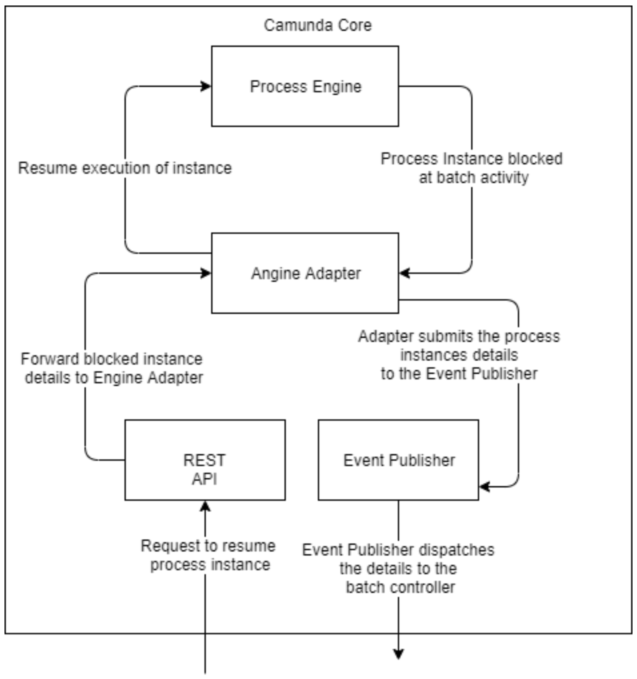

Camunda Core
Process Engine
Resume execution of instance
Process Instance blocked at batch activity
Angine Adapter
Forward blocked instance details to Engine Adapter
Adapter submits the process instances details to the Event Publisher
REST API
Event Publisher
Request to resume process instance
Event Publisher dispatches the details to the batch controller

Figure 14: Engine Adapter function within the architecture

Figure 14 shows how the Engine Adapter interacts within the scope of Camunda Core. Once a process instance reaches an activity that has been flagged for batching, its execution is halted. Its current state and the activity are stored in an Engine Adapter object internally, and the Adapter then proceeds to send the details to the Event Publisher. In turn, the Event Publisher sends these details to the message queue. The instance remains in this halted state until an external signal is sent to the REST API which then calls the Adapter to resume its execution. Once the instance starts executing again, the Process Engine is solely responsible for its life-cycle.

The **Event Publisher** is a very simple message queue client, whose sole purpose is to publish messages to the message queue server, a third party software running locally.

The **REST API** is, as mentioned, a separate module that contains all the classes that handle HTTP communication with external components. It routes requests from Camunda's web page applications, as well as the Batch Controller, as can be seen in Figure 13. After the requests are routed to the right method, Process Engine functions are then called to process them. As a result, external applications do not have explicit access to Process Engine functions.

II.3 Batch Controller

The Batch Controller is similar to Camunda Core, in that, there is no actual distinction between the components depicted in Figure 13, but is rather a piece of code with multiple functionalities. Additionally, Batch Management Controller and Batch Activity Controller correspond to REST API and Process Execution Engine respectively, since their role in the component is quite similar to their counterparts. The Batch Controller is a play! application written in Java. It is essentially a web server that uses an underlying REST system for HTTP communication.

A more accurate view of how the various modules of the Batch Controller interact with their environment is available in Figure 15. In Section III, each module's role and function will be discussed in more detail.

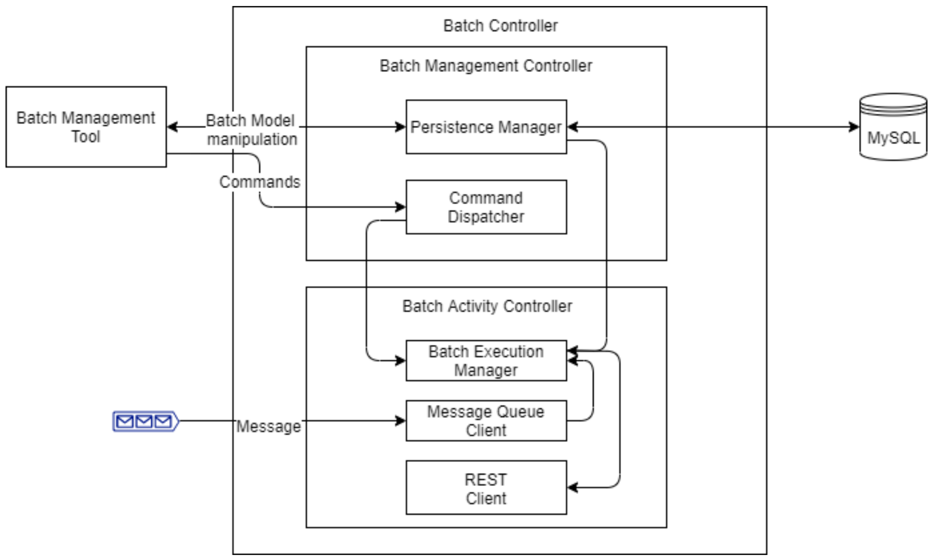


Figure 15: Batch Controller within the architecture

As with Camunda Core, the parts of Batch Controller in Figure 15 are not separate modules, but elements of the same piece of code. In that regard, the **Batch Management Controller** corresponds to the REST API of Camunda Core, as it contains the functionality to handle HTTP requests from the Batch Management Tool. The **Persistence Manager** manages the storage of the request data to the MySQL database, as well as their retrieval, when needed. The **Command Dispatcher** is simply a loose term to describe methods that unpackage the data of the HTTP request from the web page and call functions from the Batch Activity Controller to handle them.

The **Batch Activity Controller** is more complex than Batch Management Controller, operation-wise. It manages the life-cycle of a batch, from the moment the first process instance is halted, to the moment the batch concludes it's execution and the instances resume their execution again. It use the **Message Queue Client** to receive updates, halted instances or exceptions related to these from Camunda Core. Lastly, the **REST Client** makes HTTP requests to the REST API of Camunda in order to resume process instance executions.

II.4 Web Applications

The web applications of the system, namely Cockpit, Tasklist, Admin and Batch Management are the user interface of the system, written in HTML CSS and JavaScript. They provide the means to manipulate process instances, process definitions, users, batch models, monitoring tools etc. Every communication from and to the applications is done via HTTP requests to Camunda (through the web server) or the Batch Controller.

The **Batch Management Tool** is a very simple interface that supports basic operations on a batch model: creation, update and deletion. In Figure 16, at the top of the page, there is a list presenting all available batch models (1). Additionally, the fields below (2) correspond to the batch properties as suggested by [2]. From first to last, these are: the name of the model, the maximum number of instances in a batch, the minimum number of instances before a batch is executed, the time (in hours, minutes, or seconds) that has to pass before the batch is executed, the manner of execution (parallel or sequential), the process name, the activity name and the grouping attribute. Additionally, there is a second tab which lists all currently blocked process instances with information on the process they belongs and which activity they are blocked on. The instances tab in Figure 17.

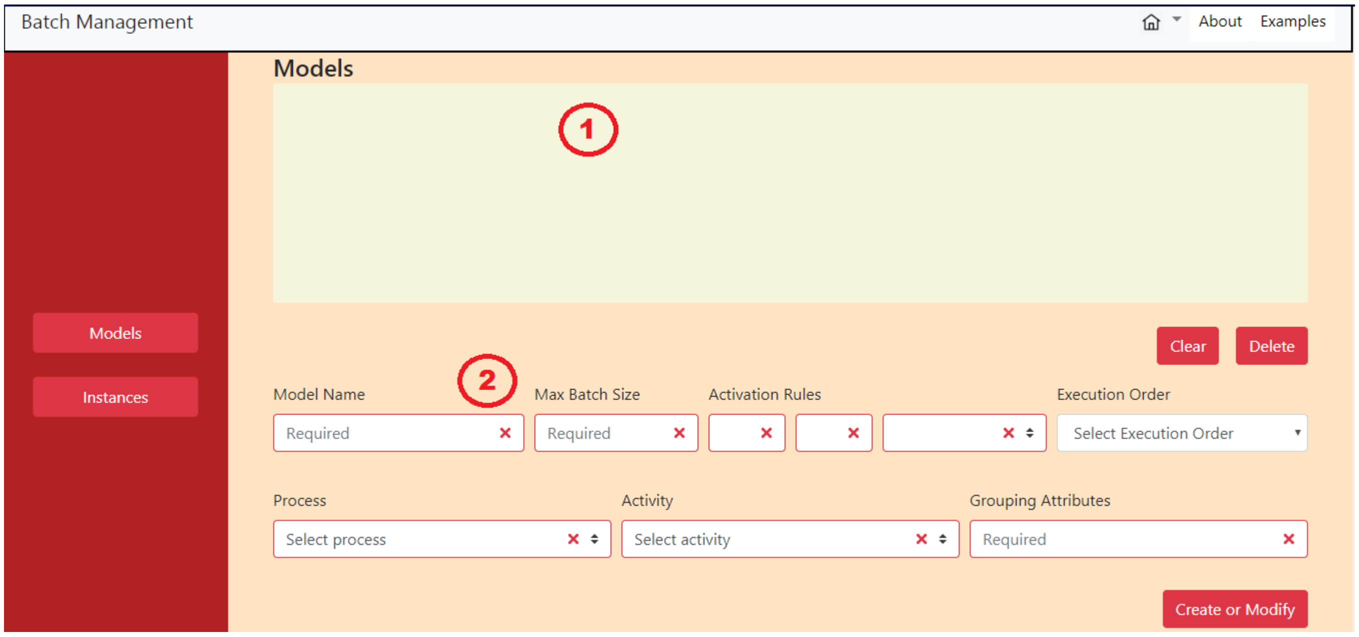


Figure 16: Batch Management Tool

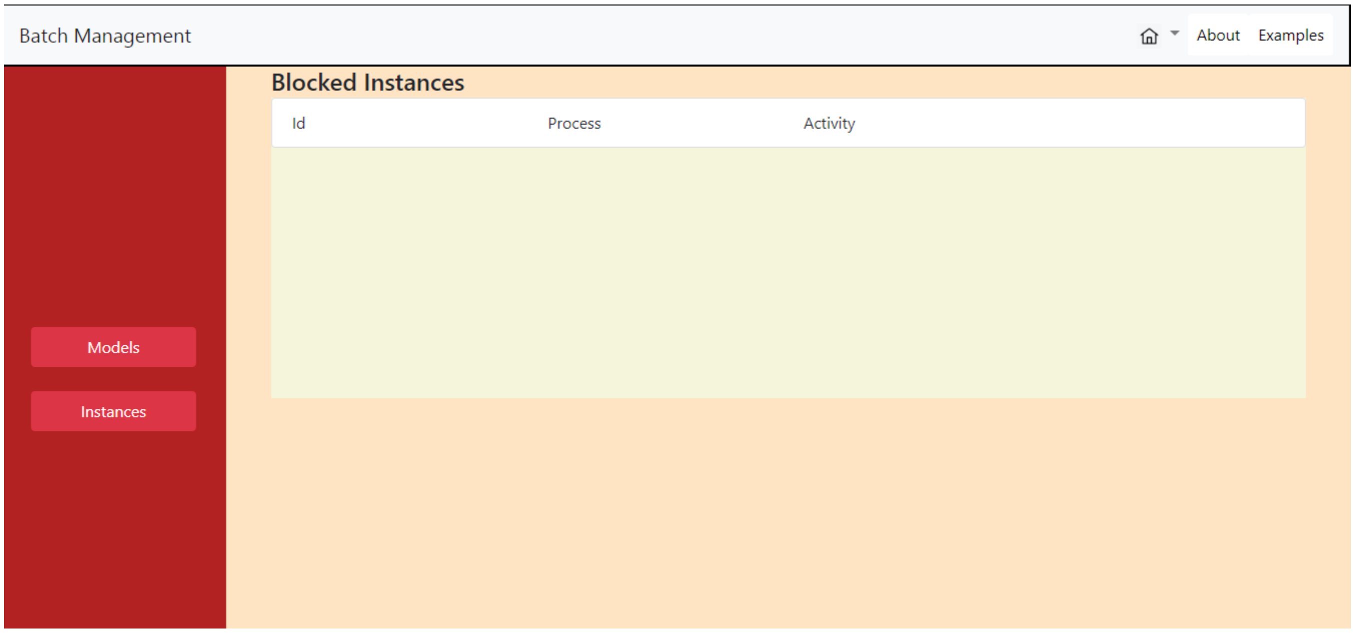


Figure 17: Blocked Instances tab

III Implementation

This section will discuss how the batch extension of the system is implemented, with regard to the three major components presented in Section II. As already mentioned in the Introduction, this system will attempt an implementation of the most flexible batch strategy described in [2], that is, the Execution Strategy. According to this strategy and the architecture discussed in the preceding sections, the implementation will involve the following steps: the process engine modification, the batch controller development and the Batch Management interface development. Additionally, the strategy and architecture impose certain requirements to the system, that the implementation will have to address. These requirements can be grouped into two categories: requirements stemming from the strategy, and requirements stemming from the batch processing theory itself. As such the identified requirements are the following:

1. Fulfillment of the three flexibility needs as presented in Section II (strategy).
2. Interruption of the process may not alter its normal execution. This effectively means that after resuming, the process must continue its execution as it would normally do without the interruption (theory).
3. The batching mechanism must be able to aggregate processes, execute them, and submit them to the engine with the new data (theory).
4. Processes must be able to continue their execution from the interrupted activity, if the batch is canceled (theory).

III.1 Process Engine Modification

Arguably, the most challenging part of this work, was the Process Engine Modification, which required extensive source code investigation, in order to identify places that code

RELATED REQUIREMENTS	1, 2, 4

can be injected or existing code must be modified. According to the theory and requirements presented earlier, the modification of the engine must implement the following functionality of Table 1.

#	**Functionality**	**Requirement**	**Notes**
1	The engine must load information regarding existing batch models on startup	1	Flexibility Need 1 & 2 (II)
2	The engine must flag batch activities during runtime	1	Flexibility Need 1
3	A blocking mechanism is necessary to halt the execution of a process when a batch activity is reached.	2, 4	
4	The state of a blocked process must be stored internally.	2, 4	
5	The engine must be able to send blocked processes and updates to the Batch Controller.	2, 4	
6	Process instances that were blocked, must be able to continue their execution from the batch activity, or the next activity in the process.	2, 4	
7	The engine must provide the necessary information for the specification of a batch model.	1	Flexibility Need 2 & 3

Table 1: Process Engine modification functionality

Functionality #7 was the first to be implemented, as it was decided that batch models have to be present before implementing the blocking mechanism. Since batch models are specified in the Batch Management Tool, certain properties of the model, specifically the Process and the Activity, have to be provided by the engine through an HTTP request from the page in Figure 16.

As mentioned already, the REST API of Camunda is a separate module, which made the decision of where to put the code to handle the aforementioned HTTP request easy. The idea of how to implement said functionality is as follows: 1) Retrieve all process definitions currently deployed on Camunda, 2) Retrieve all activities that each process contains, 3) Package each process with its activities in a new object, 4) Serialize all new objects into a JSON object and send as response to browser. In Listing 2 is the implementation of steps 1 and 2. The list of *ProcessDefinitionExtract* objects is then serialized into a JSON object and sent back to the requester (browser).

```
@GET
@Path("/activities")
```

```java
@Produces(MediaType.APPLICATION_JSON)
List<ProcessDefinitionExtract> getProcessActivityNames(@Context
 UriInfo uriInfo);
```

Listing 1: Method declaration with REST path

```java
 @Override
 public List<ProcessDefinitionExtract> getProcessActivityNames(
UriInfo uriInfo){

     List<ProcessDefinitionDto> list  = getProcessDefinitions(
uriInfo, null, null);
     List<ProcessDefinitionExtract> extractList = new ArrayList<>();

     for(ProcessDefinitionDto def: list){
         String name = def.getName();
         String id = def.getId();
         List<ProcessActivityExtract> activityList =
getProcessActivitiesByProcessId(id);
         extractList.add(new ProcessDefinitionExtract(id, name,
activityList));
     }

     return extractList;
 }

 private List<ProcessActivityExtract>
getProcessActivitiesByProcessId(String processId){

     List<ProcessActivityExtract> list = new ArrayList<>();
     Collection<ModelElementInstance> taskInstances;
     InputStream processModelIn = null;

     try {
         processModelIn = getProcessEngine().getRepositoryService().
getProcessModel(processId);
         byte[] processModel = IoUtil.readInputStream(processModelIn
, "processModelBpmn20Xml");
         InputStream stream = new ByteArrayInputStream(processModel)
;
         BpmnModelInstance modelInstance = org.camunda.bpm.model.
bpmn.Bpmn.readModelFromStream(stream);
         ModelElementType taskType =  modelInstance.getModel().
getType(Task.class);
         taskInstances = modelInstance.getModelElementsByType(
taskType);
     } catch (AuthorizationException e) {
         throw e;
     } catch (ProcessEngineException e) {
         throw new InvalidRequestException(Status.BAD_REQUEST, e, "
No matching definition with id " + processId);
     } finally {
         IoUtil.closeSilently(processModelIn);
     }

     for(ModelElementInstance task: taskInstances){
         list.add(new ProcessActivityExtract(task.getAttributeValue(
 "id"), task.getAttributeValue("name")));
     }

     return list;
 }
```

Listing 2: Method definition

Something of note is that the model of each process definition must be retrieved from storage and streamed as a byte array before converted to usable Java objects, as can be seen in Listing 2. Other methods were tried but did not permit filtering process elements by type and were also considerably slower.

The next step was to update the engine about existing batch models or newly created/updated/deleted ones (#1 and #2). For this purpose, a global Set<String> variables was created in the Engine Adapter to hold all activity Ids that have a batch model assigned. For the sake of simplicity and reduced latency in identifying batch activities during runtime, two assumptions were made:

1. Each activity in a process has a unique name and Id.
2. No two activities with the same Id exist across all processes currently deployed to the engine.

The premise of the second assumption may, of course, be permitted, if an activity Id is correlated with a process Id, with the latter being definitely unique in the engine. In this case a different container would be required, eg. a Map<String, Set<String» to hold a list of process Ids, with each having a set of unique activities Ids. While this would indeed work in a possible production environment, it was not deemed necessary for the needs of this project (extra memory etc.).

The set of activity Ids is modified in two occasions: on engine startup and during runtime, when a batch model is create, deleted, or updated. In the first case, code has be included in a startup script in order to initialize the set as soon as possible. In the code of Camunda there is an initialization script that executes operations steps and is suitable for such an occasion. In Listing 3, the step MySQLBatchModelRetrievalStep retrieves all activity Ids from a MySQL database and inserts them in the set. The implementation of the step in Listing 4.

```
    protected void deployBpmPlatform(LifecycleEvent event) {

    final StandardServer server = (StandardServer) event.getSource();

    containerDelegate.getServiceContainer().createDeploymentOperation("
   deploy BPM platform")
      .addAttachment(TomcatAttachments.SERVER, server)
      .addStep(new TomcatParseBpmPlatformXmlStep())
      .addStep(new DiscoverBpmPlatformPluginsStep())
      .addStep(new StartManagedThreadPoolStep())
      .addStep(new StartJobExecutorStep())
      .addStep(new PlatformXmlStartProcessEnginesStep())
    .addStep(new MessageProducerStep())
```

```
      .addStep(new MysqlBatchModelRetrievalStep())
      .execute();

    LOG.camundaBpmPlatformSuccessfullyStarted(server.getServerInfo());

  }
```

Listing 3: Initialization script

```
    public class MysqlBatchModelRetrievalStep extends
   DeploymentOperationStep {

    private static Connection connection;

    public String getName() {
        return "Retrieving batch models";
    }

    @Override
    public void performOperationStep(DeploymentOperation
   operationContext){

        connect();
        EngineAdapter.INSTANCE.setList(getData());
    }

    public void connect(){

        final String url = "jdbc:mysql://localhost:3306/test";

        try{
            connection = DriverManager.getConnection(url, "root", "root
   ");
        }
        catch(Exception e){
            MessageProducer.publish("Connection error: "+e.toString(),
   Queue.exception);
        }
    }

    public Set<String> getData(){

        Set<String> list = new HashSet<>();

        try{
            Statement stmt = connection.createStatement();
            ResultSet rs = stmt.executeQuery("select activityId from
   models");

            while(rs.next()){
                list.add(rs.getString("activityId"));
            }
        }
        catch(Exception e){
            MessageProducer.publish("Data retrieval error: "+e.toString
   (), Queue.exception);
        }

        return list;
    }
}

  }
```

Listing 4: Batch Activity Retrieval Step Implementation

Before moving to the implementation of the second case, it is important to explain the two ways with which Camunda starts a process instance. In a given engine session, an internal storage service called DeploymentCache containing process definitions that have already been parsed and stored with their internal Java object representation. This means that a process that has been instantiated once during the session, odes not need to be parsed again every time it needs to be instantiated. When a process definition model is parsed for the first time during the session, an ActivityImpl object is created representing an activity in the model. The code was modified to check whether there is a batch model for this activity, by searching for the activity Id in the Set discussed previously. If the Id exists, then a flag "block", which is added to the ActivityImpl class as a field, is set to true. The modification in Listing 5 (lines 4-6).

```
1       public ActivityImpl createActivity(String activityId) {
2       ActivityImpl activity = new ActivityImpl(activityId,
       processDefinition);
3       if (activityId!=null) {
4         if(EngineAdapter.INSTANCE.findBatchActivityById(activityId)){
5           activity.setBlock(true);
6         }
7         if (processDefinition.findActivity(activityId) != null) {
8           throw new PvmException("duplicate activity id '" + activityId +
         "'");
9         }
10        if (BACKLOG.containsKey(activityId)) {
11          BACKLOG.remove(activityId);
12        }
13        namedFlowActivities.put(activityId, activity);
14      }
15      activity.flowScope = this;
16      flowActivities.add(activity);
17
18      return  activity;
19    }
```

Listing 5: Setting batch flag to activity

In case the process definition exists in the DeploymentCache, meaning ActivityImpl objects already exist for each activity, then it is sufficient to retrieve the definition from the DeploymentCache, then retrieve the specific activity from the definition and set the flag to true. Moreover, as a flexibility aspect, the administrator may decide that a specific process instance that has been halted should be removed from the batch and resumed independently from the rest. This is why the object ExecutionEntity, representing an active process instance, should also have a "block" flag. The code in Listing 6 sets the flags in both the ActivityImpl and ExecutionEntity of already running instances. If the instance has already reached and executed the batch activity, then it is no longer affected by the flags and continues its execution. If another instance has not yet reached the batch activity, then it is halted when it does. This way the flexibility need Evolution (II) is satisified. it should be noted that the function in Listing 6 is a REST function that receives data from the batch controller such as the value of the flag (true/false) and activity Id.

```
    public void setBatchActivity(BatchActivityDto dto){

    String activityId = dto.getActivityId();
    String definitionId = dto.getProcessDefinitionId();
    boolean block = dto.isBlock();

    ProcessEngineConfigurationImpl config = EngineAdapter.getConfiguration();
    Context.setProcessEngineConfiguration(config);
    Context.setCommandContext(new CommandContext(config));
    DeploymentCache cache = config.getDeploymentCache();
    ProcessDefinitionEntity process = cache.findDeployedProcessDefinitionById(definitionId);

    if(process != null){
        try{
            process.getActivities()
                    .stream()
                    .filter(item -> item.getActivityId().equals(activityId))
                    .findFirst().ifPresent(batchActivity -> batchActivity.setBlock(block));

            List<ExecutionEntity> executions = EngineAdapter.getExecutionList()
                    .values()
                    .stream()
                    .filter(item-> item.getProcessDefinitionId().equals(definitionId))
                    .collect(Collectors.toList());

            for(ExecutionEntity item : executions){
                item.getProcessInstance().setBlock(block);
            }
        }
        catch(Exception e){
            MessageProducer.publish(this.getClass().getName()+"->"+e.toString(), Queue.exception);
        }
        finally{
            Context.removeProcessEngineConfiguration();
            Context.removeCommandContext();
        }
    }
    else{
        if(block){
            EngineAdapter.addBatchActivity(activityId);
        }
        else{
            EngineAdapter.removeBatchActivity(activityId);
        }
    }
    }
```
Listing 6: Batch flag set during runtime

The blocking mechanism at #3 also required extensive investigation which revealed that when a process reaches an activity is executes and operation called "PvmAtomicOperationActivityExecute" that implements the execute() method of the interface "CoreAtomicOperation". The code of this method, visible in Listing **??**, contains the blocking mechanism at lines 5-15. The idea is that when this activity is reached, the flags of the activity and the instance are checked, and if both are true, the general execution of the instance is interrupted by a "return" statement. The engine behaves as if the instance has finished and then removes entities needed for further execution, namely the CommandInvocationContext, the ProcessEngineConfiguration and the ContextClassLoader. As a result, these entities have to be stored separately along with the general information of the process instance. This information is stored in an object called BatchEntity which is defined in Listing 8. This satisfies functionality #4 of Table 1.

As an additional note, the Id of the instance is used to retrieve the BatchEntity in order to resume the instance, but sometimes the Id of the ExecutionEntity may refer to a child execution of the parent, meaning the process instance. Retrieving this ExecutionEntity and attempting to resume it would fail, since the child does not contain the necessary information that is contained in the father. The solution is in line 11 of the listing, which is to check whether the parentId of the ExecutionEntity is null. If it is null, then the ExecutionEntity is the parent and we can use the Id for the BatchEntity. If parentId is not null, then the ExecutionEntity is a child execution, and the parentId needs to be used in the BatchEntity. In any case, the correlation later, always uses the Id of the parent.

```
1      public void execute(PvmExecutionImpl execution) {
2
3      final ActivityImpl activity = execution.getActivity();
4
5      if(activity.isBlock() && execution.getProcessInstance().isBlock()){
6        ProcessEngineConfigurationImpl config = Context.
     getProcessEngineConfiguration();
7        CommandInvocationContext commandInvocationContext = Context.
     getCommandInvocationContext();
8        ClassLoader executionContextClassLoader = Thread.currentThread().
     getContextClassLoader();
9        execution.setContextClassLoader(executionContextClassLoader);
10       EngineAdapter.getBatchList().add(new BatchEntity(execution,
     config, commandInvocationContext));
11       String id = execution.getParentId() != null ? execution.
     getParentId() : execution.getId();
12       EngineAdapter.INSTANCE.prepareProcessInstanceForBatch((
     ExecutionEntity)execution);
13       MessageProducer.publish("Activity '"+activity.getName()+"' is
     blocked from executing... Process Instance id is: "+id, Queue.info);
14       return;
15     }
16
17     //If this process instance was blocked before, then set the context
       class loader
18     if(activity.isBlock()){
      ClassLoaderUtil.setContextClassloader(execution.
   getContextClassLoader());
    }

    execution.activityInstanceStarted();

    execution.continueIfExecutionDoesNotAffectNextOperation(new
   Callback<PvmExecutionImpl, Void>() {
      @Override
      public Void callback(PvmExecutionImpl execution) {
        if (execution.getActivity().isScope()) {
          execution.dispatchEvent(null);
        }
        return null;
      }
    }, new Callback<PvmExecutionImpl, Void>() {

      @Override
      public Void callback(PvmExecutionImpl execution) {

        ActivityBehavior activityBehavior = getActivityBehavior(
   execution);

        ActivityImpl activity = execution.getActivity();
        LOG.debugExecutesActivity(execution, activity, activityBehavior
   .getClass().getName());

        try {
          MessageProducer.publish("Now executing activity: "+activity.
   getName(), Queue.info);
          activityBehavior.execute(execution);
        } catch (RuntimeException e) {
          throw e;
        } catch (Exception e) {
          throw new PvmException("couldn't execute activity <" +
   activity.getProperty("type") + " id=\"" + activity.getId() + "\"
   ...>: " + e.getMessage(), e);
        }
        return null;
      }
    }, execution);
  }
```

Listing 7: Blocking mechanism

```
    public class BatchEntity {

    private PvmExecutionImpl execution;
    private ProcessEngineConfigurationImpl config;
    private CommandInvocationContext commandInvocationContext;

    public BatchEntity(PvmExecutionImpl execution,
   ProcessEngineConfigurationImpl config,
                      CommandInvocationContext
   commandInvocationContext){

        this.execution = execution;
```

```java
        this.config = config;
        this.commandInvocationContext = commandInvocationContext;
    }

    public String getId(){
        return this.execution.getId();
    }

    public String getParentId(){
        return this.execution.getParentId();
    }

    public PvmExecutionImpl getExecution(){
        return this.execution;
    }

    public ProcessEngineConfigurationImpl getProcessEngineConfiguration
    (){
        return this.config;
    }

    public CommandInvocationContext getCommandInvocationContext(){
        return this.commandInvocationContext;
    }

    protected void compareVariables(Map<String, Object> newVariables){

        Map<String, Object> currentVariables = execution.getVariables()
    ;
        Object o = null;
        for(String str : currentVariables.keySet()){
            Class<?> clazz = currentVariables.get(str).getClass();
            Class<?> newClazz = newVariables.get(str).getClass();
            if(clazz == Long.class && newClazz == Integer.class){
                o = newVariables.get(str);
            }
            try{
                o = (long) (int) o;
                newVariables.put(str, o);
            }
            catch(Exception e){
                MessageProducer.publish(this.getClass().getName()+": "+
    e.toString(), Queue.exception);
            }
        }

    }

}
```

Listing 8: BatchEntity definition

Functionality #5 is enabled with the use of the MessageProducer, a RabbitMQ client that pushes messages to three different queues based on the message type. These queue are: the information queue, which stores messages regarding general information about the engine, process instance states, activities that are being executed etc, the exception queue which stores exceptions and error during engine runtime and the block queue which stores JSON object of blocked processes. The definition of the MessageProducer is available in Listing 9. Serialization is accomplished using the Jackson library.

```java
    public class MessageProducer {

    private ConnectionFactory factory;
    private Connection connection;
    private static Channel channel;
    private static final Logger LOGGER = Logger.getLogger(
   MessageProducer.class.getName());
    private static MessageProducer instance = new MessageProducer();

    private MessageProducer(){

        factory = new ConnectionFactory();
        factory.setHost("localhost");

        try{
            connection = factory.newConnection();
            channel = connection.createChannel();
            channel.exchangeDeclare("batching", "direct", false, false,
     null);
            channel.queueDeclare("info", false, false, false, null);
            channel.queueDeclare("block", false, false, false, null);
            channel.queueDeclare("exception", false, false, false, null
   );
            channel.queueBind("info", "batching", "info");
            channel.queueBind("block", "batching", "block");
            channel.queueBind("exception", "batching", "exception");
        }
        catch(Exception e){
            LOGGER.severe(e.getMessage());
        }
    }

    public static MessageProducer getInstance(){
        return MessageProducer.instance;
    }

    public Channel getChannel(){
        return this.channel;
    }

    public static void publish(String message, Queue queue){

        try{
            channel.basicPublish("batching", queue.getName(), null,
   message.getBytes("UTF-8"));
        }
        catch(Exception e){
            LOGGER.severe(e.getMessage());
        }
    }
}
```

Listing 9: MessageProducer definition

Finally, functionality #6 is realized in two steps: The Batch Controller has to make an HTTP request containing the process instance(s), the variables and a boolean value indicating how the instance(s) may proceed with execution. The function handling the HTTP method is visible in Listing 10. After deserializing the JSON object, the operation that has to be executed is determined by the boolean value. If true, the instance resumes execution from the batch activity with the operation ACTIVITY_EXECUTE which executes the activity the instance is at. If false, the operation ACTIVITY_LEAVE is executed, which leaves the current (batch) activity and goes to the next.

The last step requires the Engine Adapter to retrieve the BatchEntity based on the Id from the HTTP request and proceed to execute the operation. However, deserialization of the JSON yielded differences between the variables types sent from the Batch Controller and the variable types in Camunda, Integer and Long respectively. The solution was to repair this disparity resulting in exceptions, by converting Integer to Long. The code for resuming the method is visible in Listing 11.

```
  public void resumeProcessInstanceById(String json){

  Map<String, Object> variables;
  PvmAtomicOperation operation;

  try{
    ResumeProcessInstanceDto dto = mapper.readValue(json,
ResumeProcessInstanceDto.class);
    String processInstanceId = dto.getProcessInstanceId();
    variables = dto.getVariables();
    if(dto.getResumeAtCurrentActivity()){
      operation = PvmAtomicOperation.ACTIVITY_EXECUTE;
    }
    else{
      operation = PvmAtomicOperation.ACTIVITY_LEAVE;
    }

    EngineAdapter.INSTANCE.resumeProcessInstanceById(
processInstanceId, variables, operation);
  }
  catch(Exception e){
    MessageProducer.publish(this.getClass().getName()+":"+e.toString
(), Queue.exception);
  }
}
```

Listing 10: REST method handling the resume command

```
  public void resumeProcessInstanceById(String processInstanceId, Map
<String, Object> variables, PvmAtomicOperation operation){

      BatchEntity entity = getBatchEntity(processInstanceId);
      PvmExecutionImpl execution = getExecution(entity);

      ProcessEngineConfigurationImpl config =
getProcessEngineConfiguration(entity);
      Context.setProcessEngineConfiguration(config);
      Context.setCommandContext(new CommandContext(config));
```

```
        Context.setCommandInvocationContext(getCommandInvocationContext
   (entity));

        if(variables!= null && !variables.isEmpty()){
            entity.compareVariables(variables);
            if(execution.getProcessInstance() != null){
                execution.getProcessInstance().setVariables(variables);
            }
            else{
                execution.setVariables(variables);
            }
        }

        execution.getProcessInstance().setBlock(false);

        try{
            MessageProducer.publish("Resuming process instance with id:
    "+ execution.getProcessInstance().getId(), Queue.info);
            operation.execute(execution);
        }
        catch(Exception e){
            MessageProducer.publish(this.getClass().getName()+": "+e.
   getMessage(), Queue.exception);
        }
        finally{
            Context.removeProcessEngineConfiguration();
            Context.removeCommandContext();
            Context.removeCommandInvocationContext();
            ProcessEngineContextImpl.set(false);
        }
    }
```

Listing 11: Engine Adapter method to resume instance

It should be noted that the same principle applies for resuming multiple instances (eg. when a batch has finished), the only difference being that the HTTP request now containing a list of process instance Ids with their variables, instead of a single one.

III.2 Batch Controller

RELATED REQUIREMENTS	1, 3, 4

The Batch Controller component roughly corresponds to the Controller in Figure 7, presented in [2]. The Batch Controller is responsible for facilitating the management of batch models, and to a lesser extend halted processes, following operations from the Batch Management Tool. These include: creating, updating and deleting models, assigning halted processes to batch clusters and managing the clusters' life-cycle, managing the execution of a batch and acting as a broker between the Batch Management Tool and Camunda in limited cases.
Similarly to the Camunda Process Engine, the following functionality can be identified, that has to be implemented by the Controller.

The Batch Controller uses a script to initialize components early on start-up. These include the BatchClusterManager, the MessageReceiver, the DBManager and the list of

#	Functionality	Requirement	Notes
1	Create/Update/Delete batch models	1	Flexibility need 2 - Looseness (II)
2	Assignment of processes to batches	3	
3	Execution of a batch	3	
4	Send batch processes with variables back to Camunda	3, 4	
5	Cancel batch	1, 4	Flexibility need 1 - Evolution
6	Remove individual instance from batch	4	Flexibility need 1 - Evolution

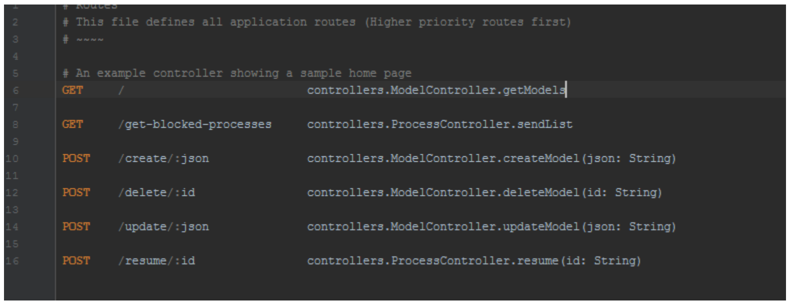
Table 2: Batch Controller functionality

models and blocked processes. The script is available in Listing 12.

```
    public class Startup {

    public static DBManager db;
    public static List<Model> models;
    public static final Logger log = LoggerFactory.getLogger("BATCH
   CONTROLLER");
    public static MessageReceiver receiver;
    public static List<ProcessInstance> globalList;

    public Startup(){

        //init components
        new BatchClusterManager();
        models = new ArrayList<>();
        db = new DBManager();
        receiver = new MessageReceiver();
        globalList = new ArrayList<>();
        db.fillLists();
    }
}
```

Listing 12: Startup script

As stated in previous sections, the Batch Controller uses an underlying REST component to route HTTP requests to the right handler. Play! framework provides a simple way to specify the request handler, by writing a line in the "routes" file of the project with format: [Http method] [handler url path] [path of the method]. In Figure 18, is the routes file of the Batch Controller. All HTTP request from the Controller are handled by functions which return a Result object indicating whether the request has succeeded. Next, the method makes an internal call to the corresponding method of the DBManager class which makes direct contact with the database to execute the appropriate queries. For each operation of functionality #1 in Table 2, the two methods are listed in Listings [], [] and []. First is the method of the request handler, followed by the DBManager method.



```
# Routes
# This file defines all application routes (Higher priority routes first)
# ~~~~

# An example controller showing a sample home page
GET     /                         controllers.ModelController.getModels

GET     /get-blocked-processes    controllers.ProcessController.sendList

POST    /create/:json             controllers.ModelController.createModel(json: String)

POST    /delete/:id               controllers.ModelController.deleteModel(id: String)

POST    /update/:json             controllers.ModelController.updateModel(json: String)

POST    /resume/:id               controllers.ProcessController.resume(id: String)
```

Figure 18: Routes file

```
    public Result createModel(String json){

        Result result = null;
        String success;

        try{
            Model model = (Model)JsonBuilder.fromJson(json, Model.class);
            success = Boolean.toString(Startup.db.create(model.createValueString()));
            if(success.equals("true")){
                Startup.models.add(model);
                result = ok("Model successfully created");
                String jsonRest = "{\"processDefinitionId\" : \""+model.getProcessId()+"\", \"activityId\" : \""+model.getActivityId()+"\", \"block\" : \"true\"}";
                String method = "process-definition/set-batch-activity";
                RESTClient.newRequest(jsonRest, method);
            }
        }
        catch(Exception e){
            result = internalServerError("Server error: Could not create model\n Reason: "+e.toString());
        }

        return result;
    }

    public boolean create(String modelValues){

        boolean success = false;

        try{
            Statement stmt=connection.createStatement();
            int rs=stmt.executeUpdate("insert into models (name, maxBatchSize, minBatchSize, timeLimit, time," +
                    " executionOrder, process, processId, activity, activityId, attributes, Id)" +
                    " values"+modelValues);
            success = (rs==1);
        }
        catch(Exception e){
            Startup.log.error("Create: " +e.getMessage());
```

```java
        }

        return success;
    }
```

Listing 13: Create model

```java
    public Result updateModel(String json){

        Result result = null;
        String success;
        try{
            Model model = (Model)JsonBuilder.fromJson(json, Model.class
);
            success = Boolean.toString(Startup.db.update(model));
            if(success.equals("true")){
                listIt = Startup.models.listIterator();
                while(listIt.hasNext()){
                    Model m = listIt.next();
                    if(m.getId().equals(model.getId())) listIt.set(
model);
                }

                result = ok("Model successfully updated");
            }

            String jsonRest = "{\"processDefinitionId\" : \""+model.
getProcessId()+"\", \"activityId\" : \""+model.getActivityId()+"\",
\"block\" : \"true\"}";
            String method = "process-definition/set-batch-activity";
            RESTClient.newRequest(jsonRest, method);
        }
        catch(Exception e){
            result = internalServerError("Server error: Could not
update model\n Reason: "+e.toString());
        }

        return result;
    }

     public boolean update(Model model){

        boolean success = false;

        try{
            PreparedStatement ps = connection.prepareStatement("update
models set " +
                        "name='"+model.getName()+"'," +
                        "maxBatchSize="+model.getMaxBatchSize()+","+
                        "minBatchSize="+model.getMinBatchSize()+","+
                        "timeLimit="+model.getTimeLimit()+","+
                        "time='"+model.getTime().toString()+"',"+
                        "executionOrder='"+model.getOrder().toString()+"',"
+
                        "process='"+model.getProcess()+"',"+
                        "processId='"+model.getProcessId()+"',"+
                        "activity='"+model.getActivity()+"',"+
                    "attributes='"+model.getAttributes()+"'  "+
                    "where Id='"+model.getId()+"';");
            success = (ps.executeUpdate()==1);
        }
        catch(Exception e){
            Startup.log.error(e.getMessage());
        }

        return success;
    }
```

Listing 14: Update model

```
     public Result deleteModel(String id){

        Result result = null;

        try{
            Model m = null;
            String response = Boolean.toString(Startup.db.delete(id));
            if (response.equals("true")){
                modelsIt = Startup.models.iterator();
                while(modelsIt.hasNext()){
                    m = modelsIt.next();
                    if(m.getId().equals(id)){
                        modelsIt.remove();
                        break;
                    }
                }

                if(m != null){
                    result = ok("Model successfully deleted");
                    String jsonRest = "{\"processDefinitionId\" : \""+m
    .getProcessId()+"\", \"activityId\" : \""+m.getActivityId()+"\", \"
    block\" : \"false\"}";
                    String method = "process-definition/set-batch-
    activity";
                    RESTClient.newRequest(jsonRest, method);
                    BatchClusterManager.removeAndFinalize(m);
                }

            }
            else{
                result = internalServerError("Server error: Could not
    delete model");
            }
        }
        catch(Exception e){
            result = internalServerError("Server error: Could not
    delete model\n Reason: "+e.toString());
        }
        return result;
    }
}

public boolean delete(String id){

        boolean success;
        try{
            Statement stmt=connection.createStatement();
            int rs=stmt.executeUpdate("delete from models where id='"+
    id+"'");
            success = (rs==1);
        }
        catch(Exception e){
            success = false;

            e.printStackTrace();
        }
        return success;
    }
```

Listing 15: Delete model

Create: The createModel() method receives a JSON object containing the model information specified from the Batch Management Tool. This String is deserialized into a Model object using the Jackson library and the create() method of the DBManager inserts it into the MySQL database. If the operation is successful, the new model is also inserted to the global model list defined in Listing 12. Subsequently, the Batch Controller sends an HTTP request to Camunda informing it of the new batch model as explained in Listing 6.

Update: The updateModel() method operates similarly to the createModel(), with a difference being that after the successful database query, the batch model has to be retrieved from model list in order to be updated locally.

Delete: the deleteModel() method only receives a single String with the Id of the model to be deleted. After the successful database query, the models list has to be iterated again in order to remove the model.

As mentioned in Section II, the system uses three different message queues for information, exceptions and blocked processes from Camunda to the Batch Controller. The MessageReceiver of the Controller listens to three queues and uses a different handler to handle the messages from each of them. While for the "information" and "exception" queues the handler only print the message in the Log, the handler of the "block" queue is functioning differently. In Listing [] is the receiving end of the "block" queue and the handling of he message by the BlockMessageHandler.

```
     channel.basicConsume("block", false, "BlockConsumer",
                    new DefaultConsumer(channel){
                        @Override
                        public void handleDelivery(String consumerTag,
                                                   Envelope envelope,
                                                   AMQP.BasicProperties
    properties,
                                                   byte[] body)
                                throws IOException
                        {
                            String routingKey = envelope.getRoutingKey
    ();
                            String contentType = properties.
```

```
getContentType();
                          long deliveryTag = envelope.getDeliveryTag
();
                          String message = new String(body, "UTF-8");
                          BlockMessageHandler.handle(message);
                          channel.basicAck(deliveryTag, false);
                      }
                  });

    public static void handle(String message){
        ProcessInstance instance;
        try{
            instance = (ProcessInstance) JsonBuilder.fromJson(message,
ProcessInstance.class);
            instance.repairVariables();
            ProcessController.addToList(instance);
            Startup.globalList.add(instance);
            BatchClusterManager.assignToCluster(instance);
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
```

Listing 16: Block Message handling

For the implementation of functionality #2, the String message from the queue has to be deserialized into a ProcessInstance object and its variables repaired (see mod. of Process Engine). In order to assign the new instance to a batch cluster, the BatchClusterManager implements the following procedure. The String identifier has to be constructed from the instance and the corresponding batch model. This identifier called a "cluster key identifier", is synthesized by three parts: the process Id, the activity name and the value of the grouping attribute. They key then has the form '[process_Id]/[activity_name]:[attribute_value]'. This string is then compared against a list of type Map<String, Integer> containing the keys of existing batch clusters. If the key is not present in the list, a new batch cluster is created. The Integer value of each entry in the Map signifies the cluster order, meaning the number of cluster of the same type that exist. This is because several cluster may exist that have the same key. For example if 5 process instances have been started and blocked, with value "test" in the grouping attribute, but the batch model only allows 3 in every batch, then these 5 instances will be split in two batches with 3 and 2 instances each respectively. When the first cluster is full, the second must be created to accommodate the remaining two instances. Additionally, every cluster has a unique id which is comprised of the cluster identifier, plus the cluster number. Lastly, the received instance is inserted to the newly created cluster or the existing one if possible. The implementation of this procedure in Listings 17 and 18.

```
      public static void assignToCluster(ProcessInstance instance){
        BatchCluster cluster = findOrCreate(instance);
        cluster.tryAdd(instance);
        instance.setClusterId(cluster.getId());
    }

    private static BatchCluster findOrCreate(ProcessInstance instance){
        BatchCluster cluster = null;
        Model model = instance.getBatchModel();
        String key = constructClusterKeyIdentifier(instance, model);
        int clusterOrder = clusterKeys.getOrDefault(key, 0);

        if(clusterOrder == 0){
            cluster = new BatchCluster(model);
            cluster.setKey(key);
            addCluster(cluster);
            clusterKeys.put(key, ++clusterOrder);
            startBatchTimer(cluster);
            Startup.log.info("New batch with id: "+cluster.getId()+"
    started");
        }
        else{
            String clusterId = key+"-"+clusterOrder;
            cluster = batchClusters.get(clusterId);
        }

        return cluster;
    }

    public static String constructClusterKeyIdentifier(ProcessInstance
    instance, Model model){
        String key;
        String processDefinitionComponent;
        String processDefinitionIdParts[] = model.getProcessId().split(
    ":");

        //If the process definition id contains the process version,
    include it in the processDefinitionComponent string
        if(processDefinitionIdParts.length == 3){
            processDefinitionComponent = processDefinitionIdParts[0]+":
    "+processDefinitionIdParts[1];
        }
        else{
            processDefinitionComponent = processDefinitionIdParts[0];
        }
        String activityComponent = model.getActivity();
        String attribute = model.getAttributes();
        String attributeValue = (String)instance.variables.get(
    attribute.toLowerCase());
        key = processDefinitionComponent+"/"+activityComponent+":"+
    attributeValue;

        return key;
    }
```

Listing 17: Assigning an instance to a cluster - BatchClusterManager class

```
    public void tryAdd(ProcessInstance instance){

        if(this.currentCapacity < this.max){
            this.instances.add(instance);
            increaseCurrentCapacity();
        }
        else{
            BatchCluster newCluster;
            try{
                newCluster = new BatchCluster(this.batchModel);
                newCluster.setKey(this.key);
                newCluster.instances.add(instance);
                BatchClusterManager.addCluster(newCluster);
                int nextBatchNumber = BatchClusterManager.getCLusterKeys().get(this.key);
                BatchClusterManager.getCLusterKeys().put(this.key, nextBatchNumber);
                BatchClusterManager.startBatchTimer(newCluster);
            }
            catch(Exception e){
                System.out.println(e.toString());
            }
        }
    }
```

Listing 18: Assigning an instance to a cluster - BatchCluster class

When a batch cluster is created, a countdown begins using a Timer object. The duration of this countdown is specified in the batch model and is common for all batches stemming from the model. Once the countdown ends, the batch is executed if the minimum number of instances has been reached. Otherwise, the batch is canceled. The Timer and TimerTask are visible in Listings [] and [].

```
      public static void startBatchTimer(BatchCluster cluster){
        Timer timer = new Timer();
        timers.put(cluster.getId(), timer);

        try{
            long delay = cluster.transformToLong();
            timer.schedule(cluster.startTimer(), delay);
        }
        catch(Exception e){
            Startup.log.error(e.toString()+" : "+e.getMessage());
        }
    }
```

Listing 19: Timer - BatchClusterManager class

```
      public TimerTask startTimer(){

        BatchCluster cluster = this;
        TimerTask task;
        task = new TimerTask() {
            @Override
            public void run() {
                if(currentCapacity < min){
                    Startup.log.info("Minimum number of instances has not been reached. Resuming individual execution...");
                    BatchClusterManager.cancelBatch(cluster);
                    cancel();
                }
                else{
                    Startup.log.info("Execution for Batch "+cluster.id+" has started");
                    cluster.setRunning(true);
```

```java
                    try{
                        if(cluster.getModel().getOrder() == Model.executionOrder.PARALLEL){
                            CountDownLatch latch = new CountDownLatch(cluster.instances.size());
                            BatchExecutionManager.executeParallel(cluster, latch);
                            latch.await();
                        }
                        else{
                            BatchExecutionManager.executeSequential(cluster);
                        }
                        Startup.log.info("Execution for Batch "+cluster.id+" has finished");
                        cluster.finalizeBatch(false);
                    }
                    catch(Exception e){
                        e.printStackTrace();
                    }
            }
        }
    };

    return task;
}
```

Listing 20: TimerTask - BatchCluster class

In case the execution order in the model is specified as parallel, each instance is executed on a different thread. The batch is concluded once all thread have finished their execution, which is determined by a CountDownLatch object. Right before the execution, the activity to execute is determined on runtime. The code in Listings 21 and 22.

```java
  public static void executeParallel(BatchCluster cluster, CountDownLatch latch) {
    String activity = cluster.getModel().getActivity().replace(" ", "");

    try{
        Class<?> clazz = Class.forName(delegatePackage+activity);
        Constructor<?> delegateConstructor = clazz.getConstructor(Map.class);
        for(ProcessInstance instance : cluster.getList()){
            new InstanceExecutor(instance.getVariables(), delegateConstructor, latch);
        }
    }
    catch(Exception e){
        Startup.log.error(e.toString());
    }
}
```

Listing 21: Loading activity to execute and thread spawning - BatchExecutionManager class

```java
    public InstanceExecutor(Map<String, Object> variables,
  Constructor<?> constructor, CountDownLatch latch){
        this.constructor = constructor;
        this.variables = variables;
        this.latch = latch;
        thread = new Thread(this);
        thread.start();

    }

    @Override
    public void run() {

            try{
                this.constructor.newInstance(variables);
                latch.countDown();
            }
            catch(Exception e){
                e.printStackTrace();
            }
    }
```

Listing 22: Thread Execution - InstanceExecutor class

The activity that is determined in real-time is of course identical to the activity behaviour that is deployed to Camunda along with the process model. In the Batch Controller, the activity script changes the process variables, which are then ready to be serialized again and sent back to the Camunda Process Engine.

When the batch has finished execution, or is canceled, then it is up for finalization, meaning, to send the process instances back to the engine to continue their normal flow of execution. In finalization, the process instances of the batch are then packaged into a list which is then serialized and sent to Camunda through HTTP, as explained in the previous section. The finalization method takes a boolean parameter indicating how the instances are to be resumed. The finalization and REST method are available in Listing 23 and 24.

```java
    public void finalizeBatch(boolean resumeAtCurrentActivity) {
        Map<String, Map<String, Object>> map = new HashMap<>();
        this.setRunning(false);

        for (ProcessInstance instance : this.instances) {
            Startup.globalList.remove(instance);
            map.put(instance.processInstanceId, instance.variables);
        }

        BatchClusterManager.removeCluster(this);

        int oldOrder = BatchClusterManager.getCLusterKeys().get(this.
  key);
        int newOrder = --oldOrder;
        if(newOrder != 0){
            BatchClusterManager.getCLusterKeys().put(this.key, newOrder
  );
        }
        else{
            BatchClusterManager.getCLusterKeys().remove(this.key);
        }


        String json = JsonBuilder.toJson(new
    ProcessInstanceListRestExtract(map, resumeAtCurrentActivity));
        String method = "process-instance/resumeMany";
        RESTClient.newRequest(json, method);
    }
```

Listing 23: Batch finalization - BatchCluster class

```
      public static int newRequest(String payload, String method){

        int code = 0;

        try{
            StringEntity jsonEntity = new StringEntity(payload,
    ContentType.APPLICATION_JSON);
            HttpPost request = new HttpPost(url+method);
            request.addHeader("content-type", "application/json");
            request.setEntity(jsonEntity);
            CloseableHttpResponse response = httpclient.execute(request
    );
            code = response.getStatusLine().getStatusCode();

        }
        catch(Exception e){

        }

        return code;
    }
```

Listing 24: REST method

Canceling a batch means preventing the batch from executing because, eg. the minimum number of instances has not been reached. Considering the implementation of the Timer-Task in Listing 20, the only operation needed is to cancel the timer itself and finalize the batch, setting the boolean value to true (the batch is not executed, so the instances have to continue from the same blocked activity). The code is visible in Listing **??**.

```
      public static void cancelBatch(BatchCluster cluster){
        Timer timer = timers.get(cluster.getId());
        try{
            timer.cancel();
        }
        catch(Exception e){
            Startup.log.error(e.toString()+" : "+e.getMessage());
        }
        finally{
            timers.remove(cluster.getId());
            cluster.finalizeBatch(true);
        }
    }
```

Listing 25: REST method

Lastly, for reasons of increased flexibility in the system, suppose that the administrator would require a way to remove a process instance from the batch, in which case the instance would have to be resumed at the blocked activity. Assuming that a process instance cannot be removed from a running batch, the method can be written as in Listing 26. An HTTP request is sent from the Batch Management Tool with the Id of the process that needs to be removed. The method then searches for it in te global list of startup (see Listing 12) and then sends its own HTTP request to Camunda.

```
      public Result resume(String processInstanceId){

        Result result;
        int code;
        ProcessInstance instance = null;

        try{
            instance = Startup.globalList
                    .stream()
                    .filter(item -> item.getProcessInstanceId().equals(processInstanceId) || (item.getParentId() != null && item.getParentId().equals(processInstanceId)))
                    .findAny()
                    .orElse(null);
            if(instance != null){
                instance.removeFromCluster();
                Startup.globalList.remove(instance);
                instance.setResumeAtCurrentActivity(true);
                ProcessInstanceRestExtract extract = new ProcessInstanceRestExtract(instance);
                String json = JsonBuilder.toJson(extract);
                String method = "process-instance/resumeOne";
                code = RESTClient.newRequest(json, method);

                if(code > 299){
                    throw new NoHttpResponseException("Server error");
                }
            }
            else{
                throw new ProcessInstanceNotFoundException("Process instance not found");
            }

            result = ok("Process successfully resumed");
        }
        catch(Exception e){
            result = internalServerError(e.toString());
        }

        return result;
    }
}
```

Listing 26: Resume single instance method

```
      public static void removeCluster(BatchCluster cluster){

        try{
            if(cluster.isRunning()){
                throw new RemoveRunningBatchException();
            }
            cluster.decrease();
            batchClusters.remove(cluster.getId());
        }
        catch(Exception e){
            Startup.log.error(e.toString());
        }
    }
```

Listing 27: Cannot remove instance from running batch

III.3 Batch Management Tool

RELATED REQUIREMENTS	1, 4

The implementation of the Batch Management Tool is not directly involved with the four requirements defined earlier in the section, as it mainly deals with the user interface functionality. Its principal functions are the listing, creation, update and deletion of models, as well as the listing of blocked processes in a separate tab as shown in Section II.4. For the implementation of the above functions, the tool makes several HTTP requests to the Batch Controller, with the exception of the list of process and list of activities which are requested from Camunda. In most cases, if the request succeeds, the result will be visualized in the browser along a pop-up message stating as such. Otherwise, the message will display an error message (eg. "Internal Server Error"), with no change in the browser. Two examples of such requests can be seen in Listings 28 and 29.

```
      function deleteModel(){
    if(selectedId !== null){
        if (confirm("Delete this model?")) {
            var xhr = new XMLHttpRequest();
            xhr.open("POST", "http://localhost:9000/delete/"+selectedId
    , true);
            xhr.onload = function (e) {
                if (xhr.readyState === 4 && xhr.status === 200) {
                    var item = document.getElementById(selectedId);
                    item.parentNode.removeChild(item);
                    var idx = modelList.indexOf(modelList.find(x => x.
    id === selectedId));
                    delete modelList[idx];
                    clearPropertyFields();
                    alert(xhr.responseText);
                }

                else {
                    alert("Server error");
                }
            }
            xhr.onerror = function (e) {
                alert("Could not reach server");
            };
            xhr.send(null);
        }
        }
    else{
        alert("No model is selected!");
    }
}
```

Listing 28: Delete model request

```
      function resumeProcess(processId){

    var request = new XMLHttpRequest();
    request.open("POST", "http://localhost:9000/resume/"+processId,
   true);
    request.onload = function() {
        if (request.status === 200 && request.status <= 299) {
            var process = document.getElementById(processId);
            process.parentNode.removeChild(process);
      alert(request.responseText);
        }
        else{
        var process = document.getElementById(processId);
            process.parentNode.removeChild(process);
            alert(request.responseText);
        }
    }
  request.onerror = function (e) {
            alert("error: " +request.status);
    };
  request.send(null);

}
```

Listing 29: Resume process instance request

3 Results

For the needs of demonstrating the system's operations, two sample processes have been created in figures 19 and 20.

The first process has two activities, Calculate Value and Change Owner and also an OR gateway. Both activities are of type "Script" and their implementation can be found in Listings 30 and 31 respectively. The following start variables were also set: stock (String), amount (Long), price (Long), owner (String). The Calculate Value activity calculates the value of the stock based on the amount, the individual price and a random multiplier and stores the result in newValue. newValue is then inserted into the process variables. The OR gateway checks whether newValue is sufficiently high, and if so, the process ends. Otherwise, a new owner is randomly selected from a predefined pool of names. Lastly, both newValue and owner variables are then printed to Tomcat log and the process ends.

The second process in Figure 20 was created to demonstrate that the flexibility need Looseness has been satisfied. As described in previous sections, this flexibility aspect dictates that the batch model can be specified during process enactment. In Camunda, one way to show this is to create a user task before the activity that we want to batch over. When engine control reaches the user task, the rest of the process instance operation is put on hold until the a human agent completes the task. During this time, the process instance is considered active and running, providing the opportunity to define a batch model.

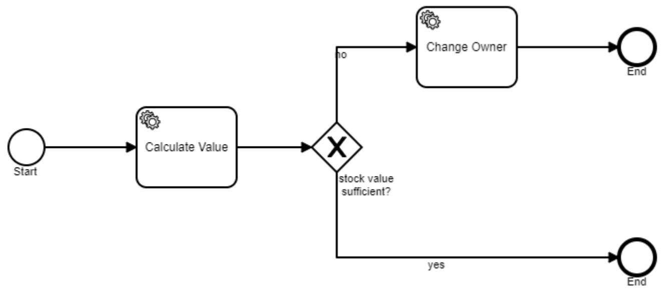


Figure 19: Sample process #1 for demonstration

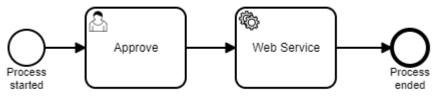


Figure 20: Sample process #2 for demonstration

```
    public class CalculateMarketValueDelegate implements JavaDelegate
  {

```

```
  @Override
  public void execute(DelegateExecution execution) throws Exception {

    long amount = (long)execution.getVariable("amount");
    long price = (long)execution.getVariable("price");
    double random = Math.random();
    long newPrice = (long)random*price + (price+40);
    long newValue = amount*newPrice;

    execution.setVariable("newValue", newValue);

  }

}
```
Listing 30: Caclulate Value implementation

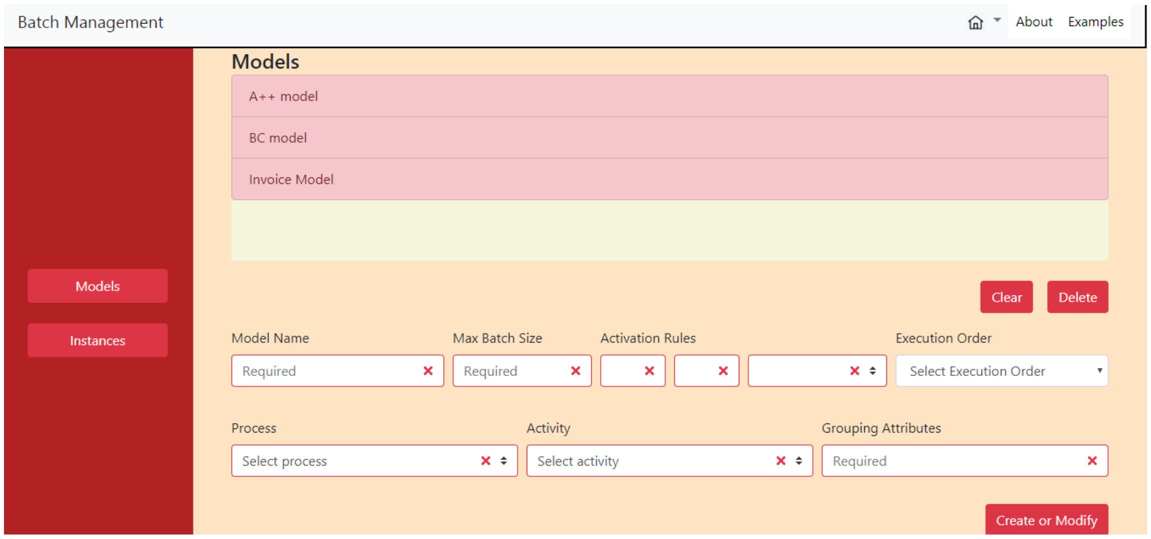
```
    public class ChangeOwnerDelegate implements JavaDelegate {

  private final Logger LOGGER = Logger.getLogger(ChangeOwnerDelegate.
   class.getName());

  @Override
  public void execute(DelegateExecution execution) throws Exception {

    String namePool[] = {"Olof Palme", "James McRandom", "Elia  S.
   Whatever", "Natalie Doe", "Albert Einstein"};
    Random random = new Random();
    int index = random.nextInt(5);
    String newOwner = namePool[index];
    execution.setVariable("owner", newOwner);

    LOGGER.info("New value of stock is: "+execution.getVariable("
   newValue"));
    LOGGER.info("New owner is: "+execution.getVariable("owner"));
  }

}
```
Listing 31: Change Owner implementation

After starting Camunda BPM platform and the Batch Controller, the Batch Management Tool looks like Figure 21. Without specifying a batch model for the above process, starting an instance as shown in Figure 22, the result is visible in Figures 23 and 24. As is evident, the process starts and finishes immediately as there is not batch activity specified. The final result in Figure 24 show that the owner changed from "test" to "Natalie Doe".

By specifying a sample batch model for this process as shown in Figure 25, and setting the minimum number of instances at 3, the countdown duration at 30 seconds and the grouping attribute to "owner", it's time to start multiple instances.

Test Case 1

As a first case, two instances are started with the same owner, let that be "random". Both are now blocked at activity "Calculate Value" and also assigned to the same cluster as they



Models

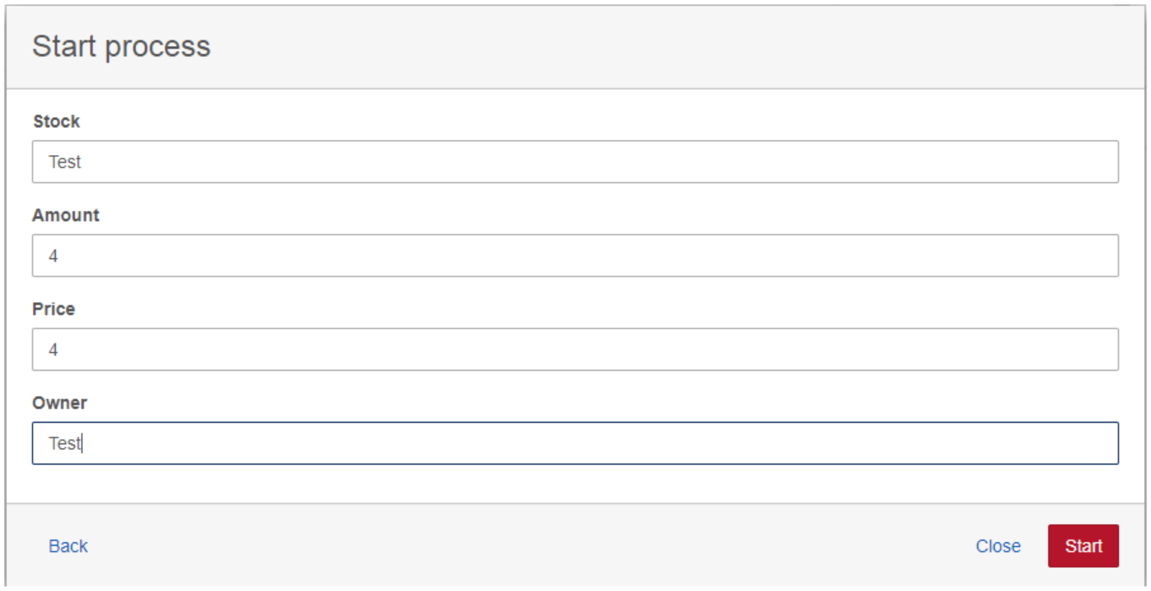
A++ model

BC model

Invoice Model

Model Name: Required
Max Batch Size: Required
Activation Rules
Execution Order: Select Execution Order
Process: Select process
Activity: Select activity
Grouping Attributes: Required

Figure 21: Batch Management Tool



Start process

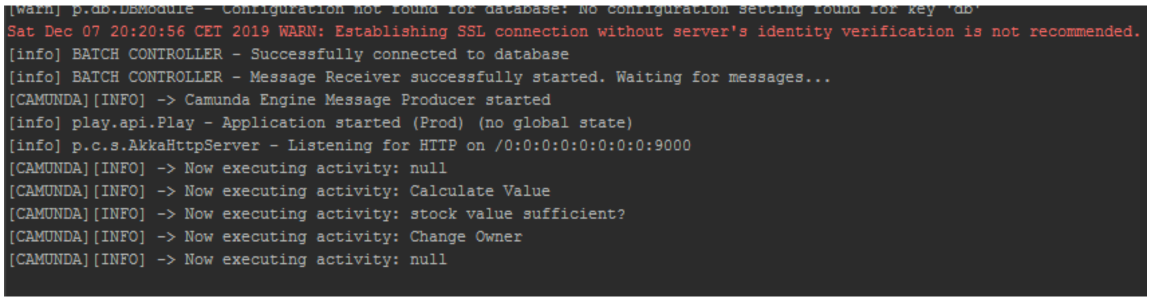
Stock: Test

Amount: 4

Price: 4

Owner: Test

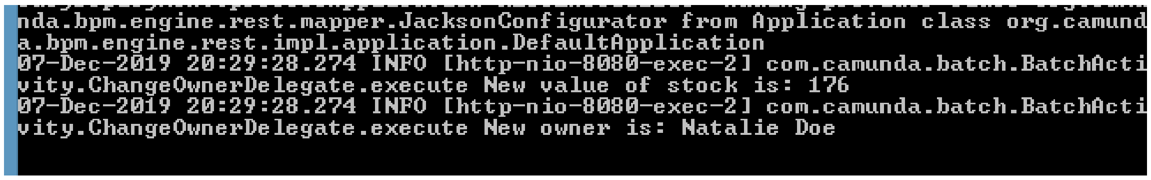
Figure 22: Starting an instance



```
[warn] p.db.DBModule - Configuration not found for database: No configuration setting found for key 'db'
Sat Dec 07 20:20:56 CET 2019 WARN: Establishing SSL connection without server's identity verification is not recommended.
[info] BATCH CONTROLLER - Successfully connected to database
[info] BATCH CONTROLLER - Message Receiver successfully started. Waiting for messages...
[CAMUNDA][INFO] -> Camunda Engine Message Producer started
[info] play.api.Play - Application started (Prod) (no global state)
[info] p.c.s.AkkaHttpServer - Listening for HTTP on /0:0:0:0:0:0:0:0:9000
[CAMUNDA][INFO] -> Now executing activity: null
[CAMUNDA][INFO] -> Now executing activity: Calculate Value
[CAMUNDA][INFO] -> Now executing activity: stock value sufficient?
[CAMUNDA][INFO] -> Now executing activity: Change Owner
[CAMUNDA][INFO] -> Now executing activity: null
```

Figure 23: Result of test instance in the Batch Controller

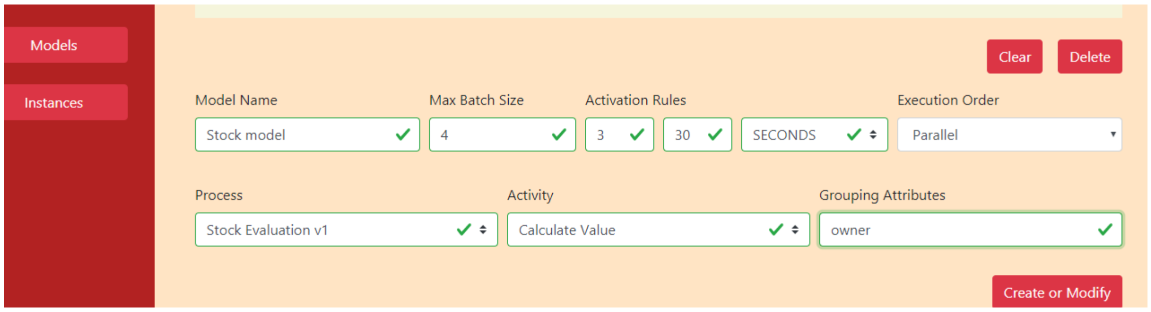
should, as can be seen in Figure 26. Additionally, the Instances tab at the web page now looks like Figure 27. After a wait of 30 seconds without starting a third process, the batch



```
nda.bpm.engine.rest.mapper.JacksonConfigurator from Application class org.camund
a.bpm.engine.rest.impl.application.DefaultApplication
07-Dec-2019 20:29:28.274 INFO [http-nio-8080-exec-2] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New value of stock is: 176
07-Dec-2019 20:29:28.274 INFO [http-nio-8080-exec-2] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New owner is: Natalie Doe
```

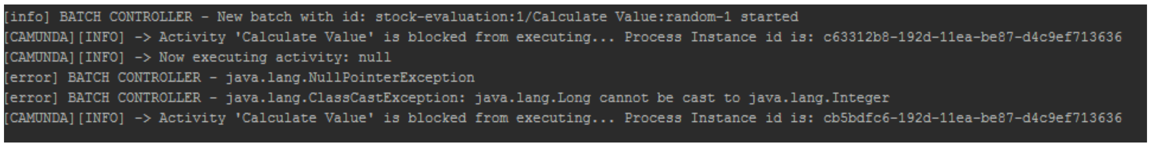
Figure 24: Result of test instance in Apache Tomcat log

is canceled and the processes sent back to Camunda, where their execution resumes at "Calculate Value" activity, as can be seen in Figures 28 and 29.



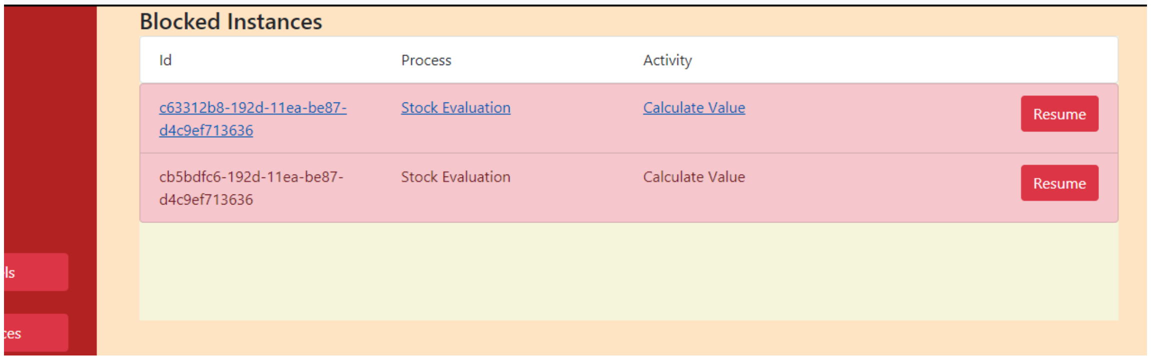
Model Name: Stock model
Max Batch Size: 4
Activation Rules: 3 | 30 | SECONDS
Execution Order: Parallel
Process: Stock Evaluation v1
Activity: Calculate Value
Grouping Attributes: owner

Figure 25: Creating model for process "Stock Evaluation"



```
[info] BATCH CONTROLLER - New batch with id: stock-evaluation:1/Calculate Value:random-1 started
[CAMUNDA][INFO] -> Activity 'Calculate Value' is blocked from executing... Process Instance id is: c63312b8-192d-11ea-be87-d4c9ef713636
[CAMUNDA][INFO] -> Now executing activity: null
[error] BATCH CONTROLLER - java.lang.NullPointerException
[error] BATCH CONTROLLER - java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer
[CAMUNDA][INFO] -> Activity 'Calculate Value' is blocked from executing... Process Instance id is: cb5bdfc6-192d-11ea-be87-d4c9ef713636
```

Figure 26: Two instances blocked

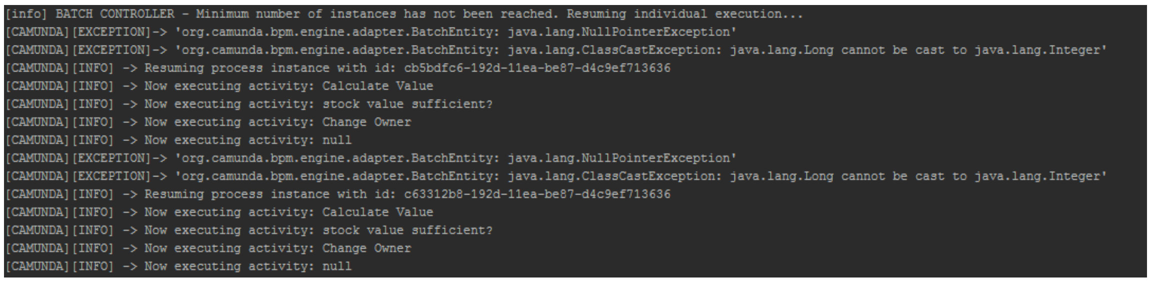


Id	Process	Activity	
c63312b8-192d-11ea-be87-d4c9ef713636	Stock Evaluation	Calculate Value	Resume
cb5bdfc6-192d-11ea-be87-d4c9ef713636	Stock Evaluation	Calculate Value	Resume

Figure 27: Blocked instances at the Instances tab

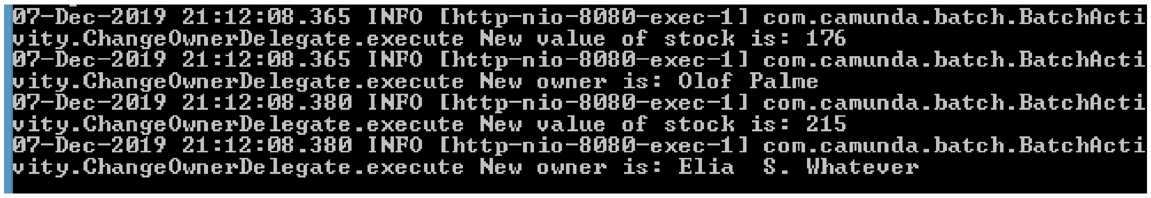
Test Case 2

By starting three instances, which is the minimum in order to execute the batch, and resuming one of them from the Instances tab of the Management Tool before the countdown ends, the instance resumes normally with rest following shortly after since the batch is



```
[info] BATCH CONTROLLER - Minimum number of instances has not been reached. Resuming individual execution...
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.NullPointerException'
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer'
[CAMUNDA][INFO] -> Resuming process instance with id: cb5bdfc6-192d-11ea-be87-d4c9ef713636
[CAMUNDA][INFO] -> Now executing activity: Calculate Value
[CAMUNDA][INFO] -> Now executing activity: stock value sufficient?
[CAMUNDA][INFO] -> Now executing activity: Change Owner
[CAMUNDA][INFO] -> Now executing activity: null
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.NullPointerException'
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer'
[CAMUNDA][INFO] -> Resuming process instance with id: c63312b8-192d-11ea-be87-d4c9ef713636
[CAMUNDA][INFO] -> Now executing activity: Calculate Value
[CAMUNDA][INFO] -> Now executing activity: stock value sufficient?
[CAMUNDA][INFO] -> Now executing activity: Change Owner
[CAMUNDA][INFO] -> Now executing activity: null
```

Figure 28: Batch Controller log showing both instances are resumed



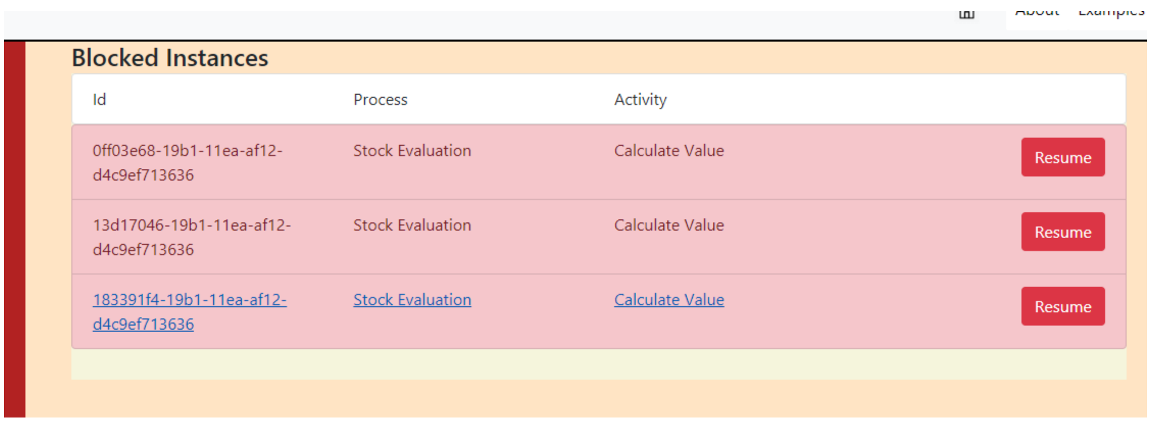
```
07-Dec-2019 21:12:08.365 INFO [http-nio-8080-exec-1] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New value of stock is: 176
07-Dec-2019 21:12:08.365 INFO [http-nio-8080-exec-1] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New owner is: Olof Palme
07-Dec-2019 21:12:08.380 INFO [http-nio-8080-exec-1] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New value of stock is: 215
07-Dec-2019 21:12:08.380 INFO [http-nio-8080-exec-1] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New owner is: Elia  S. Whatever
```

Figure 29: Apache Tomcat log showing both instances are resumed

canceled. Figure 30 shows all three instances are blocked. After resuming one of them, two are left in the batch (Figure 31) which is not enough to execute it. After 30 seconds have passed, the batch is canceled, and the remaining two are also individually executed by Camunda (Figures 32 and 33).

Test Case 3

Lastly, starting again three instances and waiting 30 seconds, the batch is executed and the instances are then returned to Camunda where the execution continues at the OR gateway as expected. The result in Figures 34 and 35.



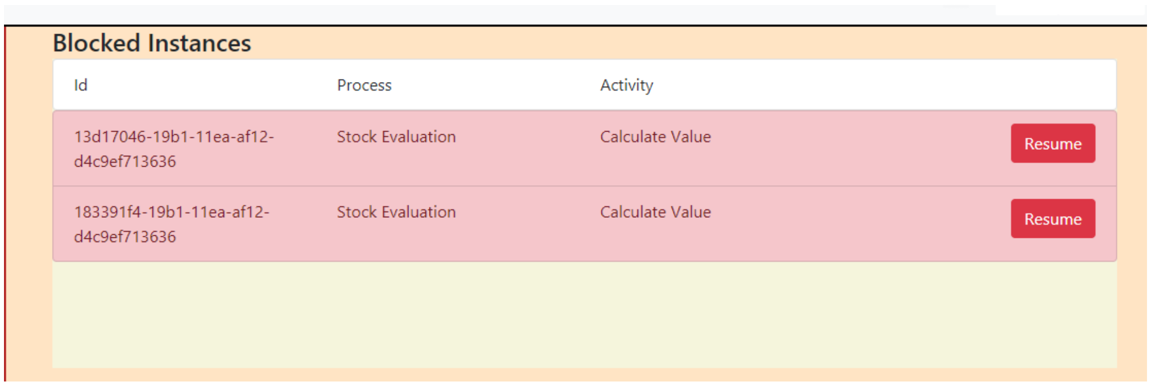
Blocked Instances

Id	Process	Activity	
0ff03e68-19b1-11ea-af12-d4c9ef713636	Stock Evaluation	Calculate Value	Resume
13d17046-19b1-11ea-af12-d4c9ef713636	Stock Evaluation	Calculate Value	Resume
183391f4-19b1-11ea-af12-d4c9ef713636	Stock Evaluation	Calculate Value	Resume

Figure 30: All three instances are blocked

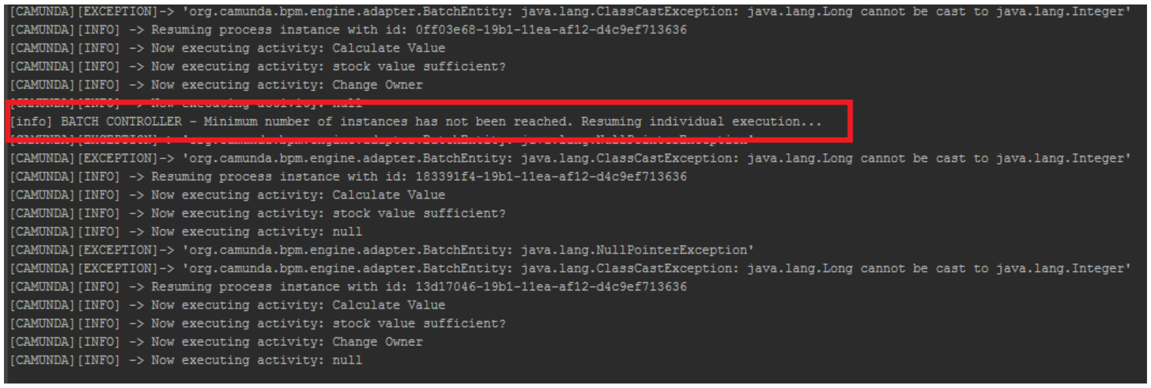
Test Case 4

For the next test case, let us start an instance of the process in Figure 20. Initially, there is no batch model for this process. Once started, the instance waits for input by a human agent as shown in Figure 36. While the instance is in this waiting state, we can create



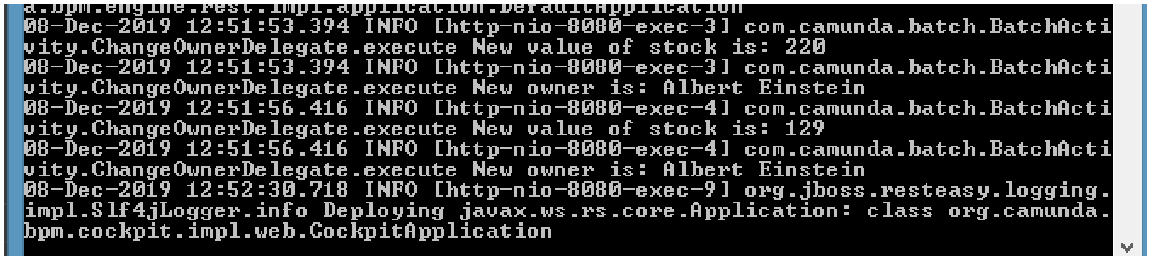
Id	Process	Activity	
13d17046-19b1-11ea-af12-d4c9ef713636	Stock Evaluation	Calculate Value	Resume
183391f4-19b1-11ea-af12-d4c9ef713636	Stock Evaluation	Calculate Value	Resume

Figure 31: Two instances now remain in the batch



```
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer'
[CAMUNDA][INFO] -> Resuming process instance with id: 0ff03e68-19b1-11ea-af12-d4c9ef713636
[CAMUNDA][INFO] -> Now executing activity: Calculate Value
[CAMUNDA][INFO] -> Now executing activity: stock value sufficient?
[CAMUNDA][INFO] -> Now executing activity: Change Owner
[CAMUNDA][INFO] -> Now executing activity: null
[info] BATCH CONTROLLER - Minimum number of instances has not been reached. Resuming individual execution...
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.NullPointerException'
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer'
[CAMUNDA][INFO] -> Resuming process instance with id: 183391f4-19b1-11ea-af12-d4c9ef713636
[CAMUNDA][INFO] -> Now executing activity: Calculate Value
[CAMUNDA][INFO] -> Now executing activity: stock value sufficient?
[CAMUNDA][INFO] -> Now executing activity: null
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.NullPointerException'
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer'
[CAMUNDA][INFO] -> Resuming process instance with id: 13d17046-19b1-11ea-af12-d4c9ef713636
[CAMUNDA][INFO] -> Now executing activity: Calculate Value
[CAMUNDA][INFO] -> Now executing activity: stock value sufficient?
[CAMUNDA][INFO] -> Now executing activity: Change Owner
[CAMUNDA][INFO] -> Now executing activity: null
```

Figure 32: Resuming one instance the batch is canceled after countdown



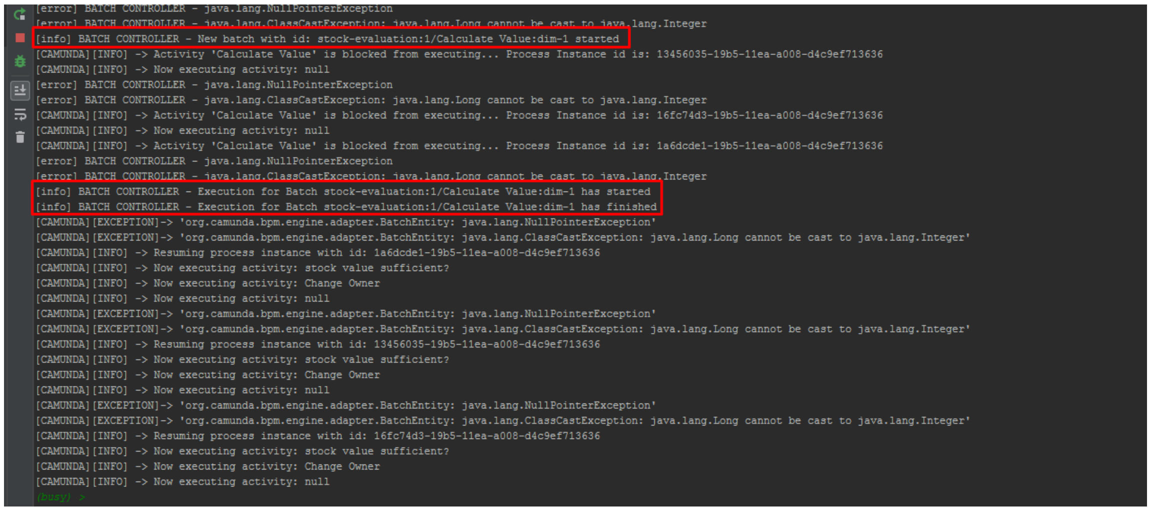
```
a.bpm.engine.rest.impl.application.DefaultApplication
08-Dec-2019 12:51:53.394 INFO [http-nio-8080-exec-3] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New value of stock is: 220
08-Dec-2019 12:51:53.394 INFO [http-nio-8080-exec-3] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New owner is: Albert Einstein
08-Dec-2019 12:51:56.416 INFO [http-nio-8080-exec-4] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New value of stock is: 129
08-Dec-2019 12:51:56.416 INFO [http-nio-8080-exec-4] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New owner is: Albert Einstein
08-Dec-2019 12:52:30.718 INFO [http-nio-8080-exec-9] org.jboss.resteasy.logging.
impl.Slf4jLogger.info Deploying javax.ws.rs.core.Application: class org.camunda.
bpm.cockpit.impl.web.CockpitApplication
```

Figure 33: Result of execution on Tomcat

a batch model as shown in Figure 37, similarly to the previous cases. Once the model is created, the instance may continue the operation by pushing the Complete button in Figure 36. The instance now blocks, as expected, at the Web Service activity that was specified in the batch model, shown in Figure 38.

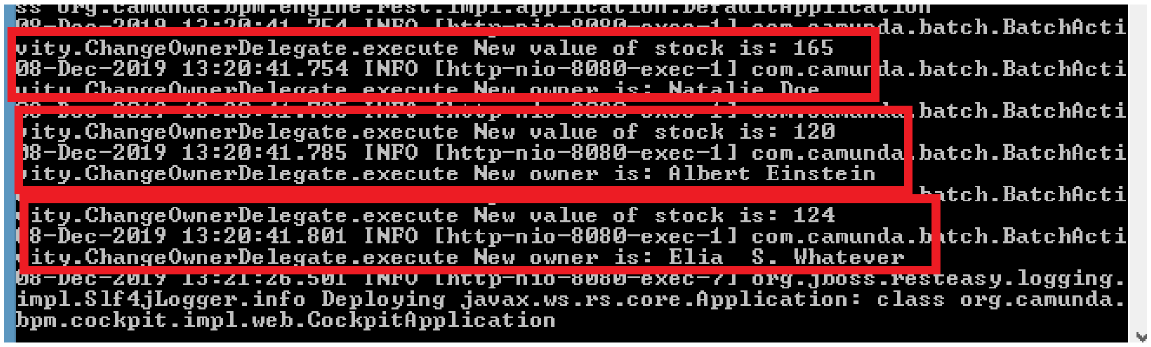
Test Case 5

For the last case, we shall create a batch model in Figure 39. Then we will start 2 instances of the Stock Evaluation process. The cluster that has been created will wait for 10 hours before it is executed. In the meantime, the model will be modified from a maximum number of instances 4 to 3. Then after starting a third instance, we can see in Figure **??** that the batch has started execution immediately, as expected.



```
[error] BATCH CONTROLLER - java.lang.NullPointerException
[error] BATCH CONTROLLER - java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer
[info] BATCH CONTROLLER - New batch with id: stock-evaluation:1/Calculate Value:dim-1 started
[CAMUNDA][INFO] -> Activity 'Calculate Value' is blocked from executing... Process Instance id is: 13456035-19b5-11ea-a008-d4c9ef713636
[CAMUNDA][INFO] -> Now executing activity: null
[error] BATCH CONTROLLER - java.lang.NullPointerException
[error] BATCH CONTROLLER - java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer
[CAMUNDA][INFO] -> Activity 'Calculate Value' is blocked from executing... Process Instance id is: 16fc74d3-19b5-11ea-a008-d4c9ef713636
[CAMUNDA][INFO] -> Now executing activity: null
[CAMUNDA][INFO] -> Activity 'Calculate Value' is blocked from executing... Process Instance id is: 1a6dcde1-19b5-11ea-a008-d4c9ef713636
[error] BATCH CONTROLLER - java.lang.NullPointerException
[error] BATCH CONTROLLER - java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer
[info] BATCH CONTROLLER - Execution for Batch stock-evaluation:1/Calculate Value:dim-1 has started
[info] BATCH CONTROLLER - Execution for Batch stock-evaluation:1/Calculate Value:dim-1 has finished
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.NullPointerException'
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer'
[CAMUNDA][INFO] -> Resuming process instance with id: 1a6dcde1-19b5-11ea-a008-d4c9ef713636
[CAMUNDA][INFO] -> Now executing activity: stock value sufficient?
[CAMUNDA][INFO] -> Now executing activity: Change Owner
[CAMUNDA][INFO] -> Now executing activity: null
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.NullPointerException'
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer'
[CAMUNDA][INFO] -> Resuming process instance with id: 13456035-19b5-11ea-a008-d4c9ef713636
[CAMUNDA][INFO] -> Now executing activity: stock value sufficient?
[CAMUNDA][INFO] -> Now executing activity: Change Owner
[CAMUNDA][INFO] -> Now executing activity: null
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.NullPointerException'
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer'
[CAMUNDA][INFO] -> Resuming process instance with id: 16fc74d3-19b5-11ea-a008-d4c9ef713636
[CAMUNDA][INFO] -> Now executing activity: stock value sufficient?
[CAMUNDA][INFO] -> Now executing activity: Change Owner
[CAMUNDA][INFO] -> Now executing activity: null
(busy) >
```

Figure 34: The batch is executed



```
ss org.camunda.bpm.engine.rest.impl.application.DefaultApplication
08-Dec-2019 13:20:41.754 INFO [http-nio-8080-exec-1] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New value of stock is: 165
08-Dec-2019 13:20:41.754 INFO [http-nio-8080-exec-1] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New owner is: Natalie Doe
08-Dec-2019 13:20:41.785 INFO [http-nio-8080-exec-1] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New value of stock is: 120
08-Dec-2019 13:20:41.785 INFO [http-nio-8080-exec-1] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New owner is: Albert Einstein
08-Dec-2019 13:20:41.801 INFO [http-nio-8080-exec-1] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New value of stock is: 124
08-Dec-2019 13:20:41.801 INFO [http-nio-8080-exec-1] com.camunda.batch.BatchActi
vity.ChangeOwnerDelegate.execute New owner is: Elia  S. Whatever
08-Dec-2019 13:21:26.501 INFO [http-nio-8080-exec-7] org.jboss.resteasy.logging.
impl.Slf4jLogger.info Deploying javax.ws.rs.core.Application: class org.camunda.
bpm.cockpit.impl.web.CockpitApplication
```

Figure 35: Result of executed batch on Tomcat

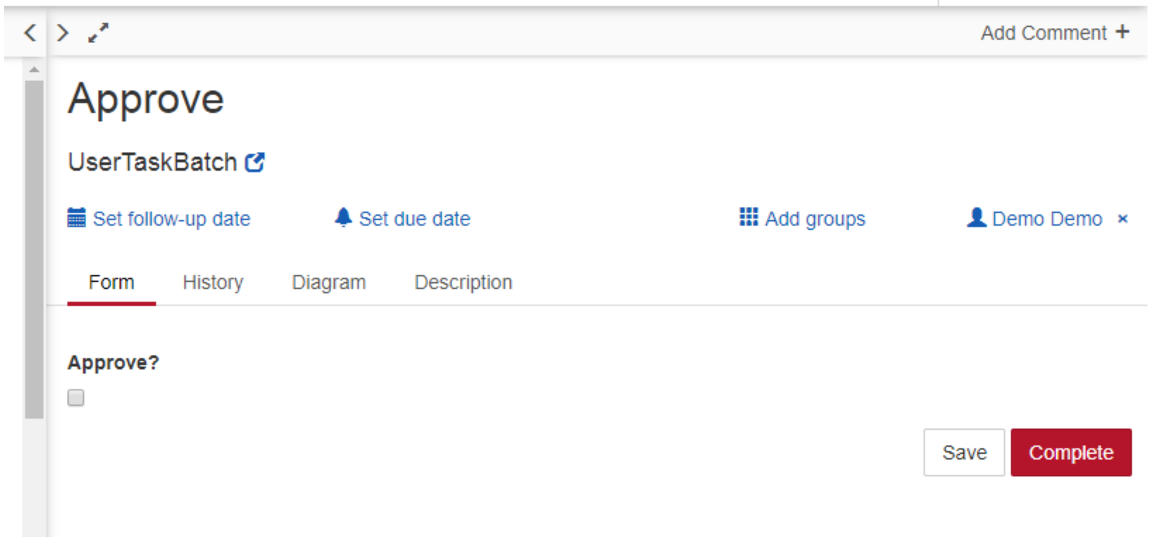
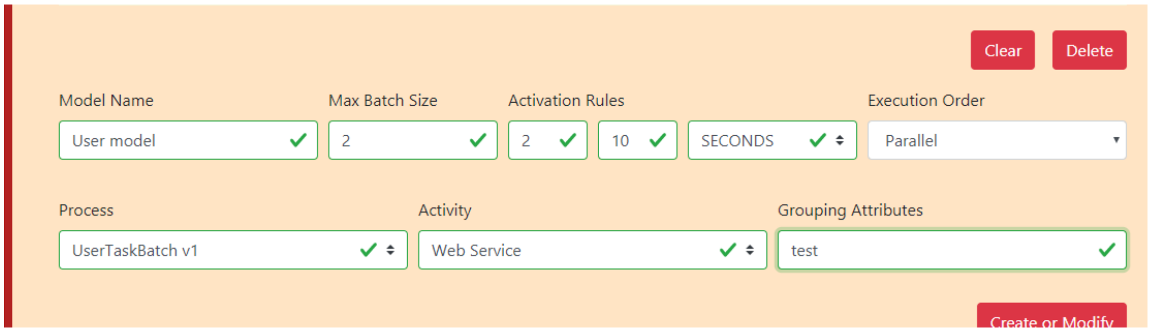


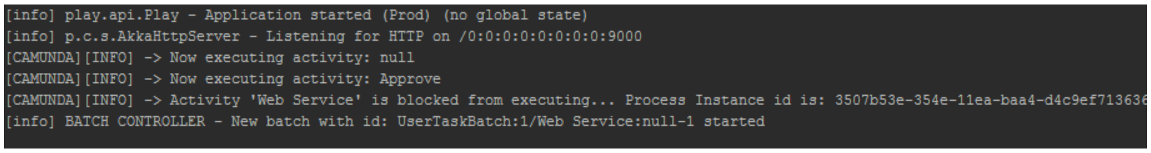
Figure 36: User task input



Model Name	Max Batch Size	Activation Rules			Execution Order
User model	2	2	10	SECONDS	Parallel

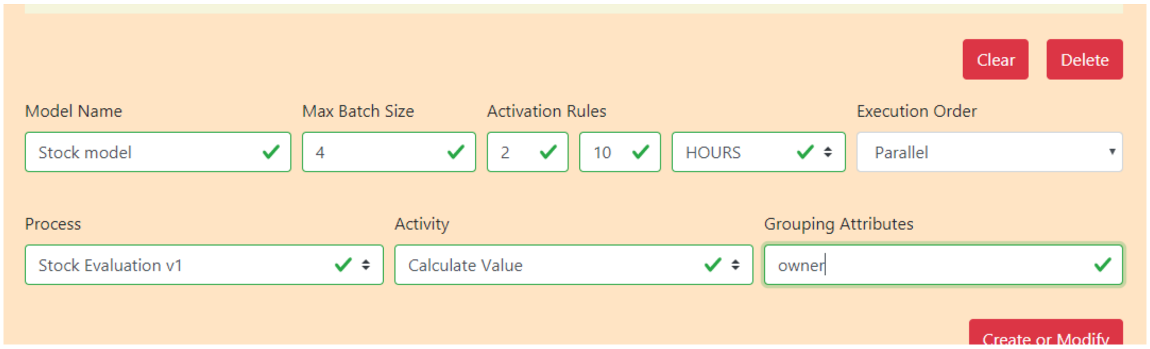
Process	Activity	Grouping Attributes
UserTaskBatch v1	Web Service	test

Figure 37: Batch model for UserTaskBatch process



```
[info] play.api.Play - Application started (Prod) (no global state)
[info] p.c.s.AkkaHttpServer - Listening for HTTP on /0:0:0:0:0:0:0:0:9000
[CAMUNDA][INFO] -> Now executing activity: null
[CAMUNDA][INFO] -> Now executing activity: Approve
[CAMUNDA][INFO] -> Activity 'Web Service' is blocked from executing... Process Instance id is: 3507b53e-354e-11ea-baa4-d4c9ef713636
[info] BATCH CONTROLLER - New batch with id: UserTaskBatch:1/Web Service:null-1 started
```

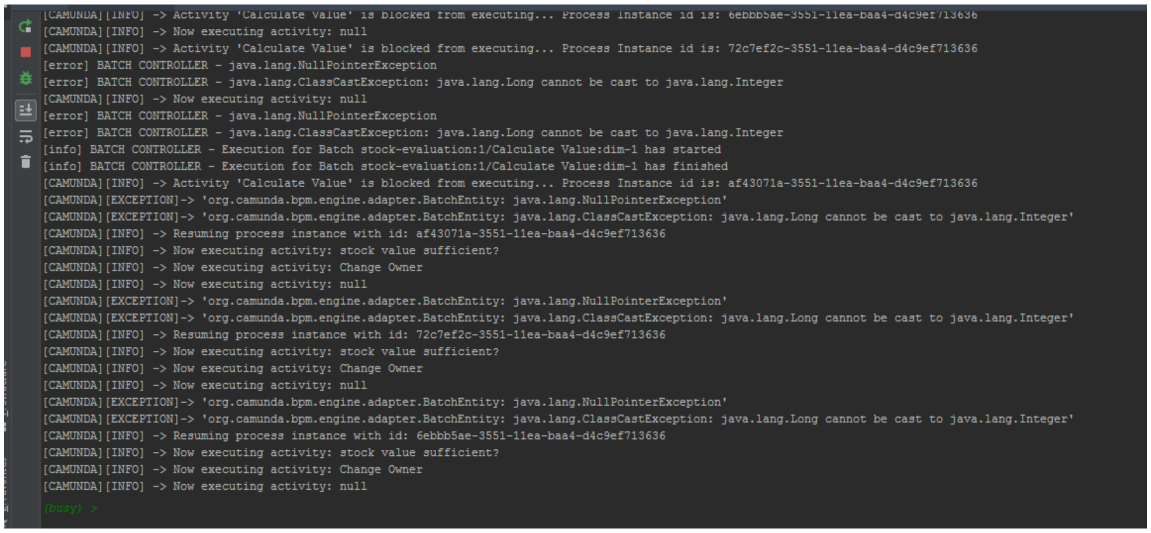
Figure 38: Instance is blocked



Model Name	Max Batch Size	Activation Rules			Execution Order
Stock model	4	2	10	HOURS	Parallel

Process	Activity	Grouping Attributes
Stock Evaluation v1	Calculate Value	owner

Figure 39: Batch model for case 5



```
[CAMUNDA][INFO] -> Activity 'Calculate Value' is blocked from executing... Process Instance id is: 6ebbb5ae-3551-11ea-baa4-d4c9ef713636
[CAMUNDA][INFO] -> Now executing activity: null
[CAMUNDA][INFO] -> Activity 'Calculate Value' is blocked from executing... Process Instance id is: 72c7ef2c-3551-11ea-baa4-d4c9ef713636
[error] BATCH CONTROLLER - java.lang.NullPointerException
[error] BATCH CONTROLLER - java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer
[CAMUNDA][INFO] -> Now executing activity: null
[error] BATCH CONTROLLER - java.lang.NullPointerException
[error] BATCH CONTROLLER - java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer
[info] BATCH CONTROLLER - Execution for Batch stock-evaluation:1/Calculate Value:dim-1 has started
[info] BATCH CONTROLLER - Execution for Batch stock-evaluation:1/Calculate Value:dim-1 has finished
[CAMUNDA][INFO] -> Activity 'Calculate Value' is blocked from executing... Process Instance id is: af43071a-3551-11ea-baa4-d4c9ef713636
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.NullPointerException'
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer'
[CAMUNDA][INFO] -> Resuming process instance with id: af43071a-3551-11ea-baa4-d4c9ef713636
[CAMUNDA][INFO] -> Now executing activity: stock value sufficient?
[CAMUNDA][INFO] -> Now executing activity: Change Owner
[CAMUNDA][INFO] -> Now executing activity: null
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.NullPointerException'
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer'
[CAMUNDA][INFO] -> Resuming process instance with id: 72c7ef2c-3551-11ea-baa4-d4c9ef713636
[CAMUNDA][INFO] -> Now executing activity: stock value sufficient?
[CAMUNDA][INFO] -> Now executing activity: Change Owner
[CAMUNDA][INFO] -> Now executing activity: null
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.NullPointerException'
[CAMUNDA][EXCEPTION]-> 'org.camunda.bpm.engine.adapter.BatchEntity: java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.Integer'
[CAMUNDA][INFO] -> Resuming process instance with id: 6ebbb5ae-3551-11ea-baa4-d4c9ef713636
[CAMUNDA][INFO] -> Now executing activity: stock value sufficient?
[CAMUNDA][INFO] -> Now executing activity: Change Owner
[CAMUNDA][INFO] -> Now executing activity: null
(busy) >
```

Figure 40: Batch executed immediately after after reducing the maximum number of instances

4 Discussion

In the beginning of Section III, the four requirements that were defined, essentially dictate the measure of success of this project. It was also stated early in this work, that a flexible approach for batch processing has to adhere to two sources: the work of Karastoyanova et al. [2] describing the most flexible strategy, and the nature of batch processing itself. The architecture and implementation presented in sections II and III were done having these requirements in mind. Table 3 shows whether the requirements were fulfilled and also which functionality was directly responsible.

Due to the architecture itself, the batch model and the process are completely independent from one another, which satisfies the Evolution requirement. Updating either of them does not affect the other in any way, since the updating the model only requires the Batch Management Tool, the Batch Controller and MySQL database. Updating a process only requires the Camunda Modeller and the Process Engine for deployment.

The Looseness property is also satisfied by the fact that because of the architecture, the batch model and the process are independent. It was shown early in the Results section that a batch model can be specified after a process has been instantiated. In the subsequent demonstration, the next instance that was started was blocked in the activity specified in the batch model. Although it was not explicitly exhibited, the same instance would be blocked even if the model was created (or update) after the instance had been started. This is because, the ActivityImpl object described in Section III is common for all instances and as a result, setting its flag to true would affect all future instances of that process. Coincidentally, this also satisfies the Adaptation of Batch Activity requirement since the batch model can be updated during runtime by specifying am activity to batch over. The flag of the activity is then set to true which now makes it a batch activity.

The process interruption was showcased in the Implementation section with the introduction of the blocking mechanism. While the mechanism succeeds in its basic function and also does not negatively affect the overall execution of the process, it may introduce bugs when taking into consideration the general workflow of Camunda. One such example is the fact that when a process resumes after being interrupted in activity A, the Cockpit application is not updated and continues to show the instance at activity A. This potentially means that traces of the instance have been left behind after it has finished and could interfere with other operations of Camunda. Furthermore, the blocking mechanism has only been implemented on simple process, meaning, a process that does not have concurrent activities (activities that are executed concurrently in different threads). This would require a different approach, in order to ensure that no zombie threads are left behind and that overall process execution is not distorted because of parent-child thread mismanagement.

The process aggregation is batches and execution seems to be working correctly, with process of the same type rounded up in batches based on the grouping attribute specified in the batch model. Some properties of the model can be updated in real-time and affect the batches referring to that model. This exempts the process and activity properties, and also the timer, since once the countdown has started it will not be possible to change the delay of the Timer object. Execution is parallel and sequential is also working, with minimal thread management that waits for all instances to finish execution before aggregating them again in a list for Camunda. As sufficiently shown in the Result section, the processes can continue their execution from the next activity in line with new variable values specified from the Batch Controller.

For increased flexibility reasons, if the batch is canceled the instances it contains must be sent back to Camunda and resume execution at the same activity they were supposed to batch over. This is indeed the case as it has been shown in the Results in Figures 32 and 33. As an additional flexible aspect, the Batch Management Tool provides a list of the blocked process with the possibility of resuming one more of them. This may obviously affect the operation of a batch as shown in the above images.

Requirement	**Fulfilled**	**Implementation**
Evolution	✓	Update model methods in Batch Controller
Looseness	✓	Process Engine modification
Process Adaptation	✓	Architecture
Adaptation of Batch Activity	✓	Update model methods in Batch Controller
Process interruption	✓	Process Engine modification
Batch Execution	✓	Batch Controller implementation
Canceled batch and process instances execution	✓	Batch Controller implementation

Table 3: Requirements

In general, the implementation of batch processing has been applied to a specific type of activity, the Service Task type which in the Camunda system is supposed to simulate a web service call. In real-life operations, batching is applied to few tasks the most important of which is the user task and the web service interaction. Considering the first is implemented in the work of Dr. Luise Pufahl [1], this work undertook the second option, with very promising results. Due to time constraints this flexible approach could not be applied to the user task, which can be the subject of a future work. Moreover, batch processing cannot be applied to a group of activities, as of yet. This means that a batch model can only be assigned to a single activity. This may count as a reduction in flexibility, as the work of Karastoyanova et. al clearly states that a batch should also apply to a group of activities, effectively forming a batch region. Unfortunately the concept of a batch region is not applicable to this project.

Additionally, given the current implementation of the Batch Controller, the Java script that is supposed to be executed and is evaluated during runtime, must be hardcoded in classpath of the project. Again due to time constraints, a Class loader could not be implemented in order to load classes in real-time as happens in Camunda. However this is only relevant when developing the software for use in actual production and does not affect elements of batch processing.

In a brief mention earlier in the section, is was reported that the blocking mechanism in Camunda leaves traces of finished instances. Given the current implementation of certain operations in the system (Process Engine), certain aspects of the mechanism may not yield the most efficient solution, in terms of memory management.

5 Conclusion

The aim of this project was to provide a concrete implementation of the most flexible approach in batch processing as described in Karastoyanova et. al [2]. An architecture was provided to support the implementation, with three major components, the BPM Engine, the Batch Controller and the User Interface, each having been implemented from scratch or modified. For determining the success of the project, six requirements were identified, stemming from the most flexible approach and the batch processing theory. The results after running several test cases have showed that all requirements have been satisfied. In conclusion, flexible batch processing on Camunda BPM is definitely possible and with a high degree of efficiency.

I Future Work

The promising results of the project also provide an opportunity for further applications in batch processing, but also additional modifications that could enhance the efficiency and usability in the context of the current implementation. The following list contains suggestions on how to proceed with future work on the project.

- **Enable batch processing in a region.** This project has achieved batching on a single activity at a time but existing research suggests that it can also be applied to a group of activities.
- **Enable batch processing for various activity types.** This work applies batching explicitly on activities of type "Script" which is executes Java code. However, it should also apply to activities that are undertaken by human agents (User Tasks). This has been sucessfully implemented by dr. Luise Pufahl in [1]. A combination of both projects would be very interesting and challenging.
- **Update Cockpit application to propagate changes occurring after batching.** The Camunda Cockpit app provides information on the state of every process instance. Unfortunately, after the modification, that information is no longer displayed correctly.
- **Implementation of multiple attribute grouping.** Grouping the instances in batches is currently realized using only one attribute.
- **Improvement of the blocking/restart mechanism.** The blocking/restart mechanism of the engine modification probably can be implemented more efficiently to avoid memory overhead.
- **Enable batch processing on concurrent activities.** This implementation of batch processing does not support activities running concurrently on different threads on the engine. This area might prove to be challenging given the different handling it requires.

Acknowledgements

I would like to give thanks to pr. dr. Dimka Karastoyanova from the University of Groningen for giving me the opportunity to work on such a challenging and thought provoking project, as well as for providing the necessary theoretical background.

I would also like to thank dr. Luise Pufahl from the University of Potsdam for her important inputs and suggestions during this period.

Lastly, I would like to thank my friend Angeliki, for giving me support.

References

[1] Luise Pufahl. *Modeling and enacting batch activities in business processes*. PhD thesis, University of Potsdam, 2018.

[2] Luise Pufahl and Dimka Karastoyanova. Enhancing business process flexibility by flexible batch processing. In *OTM Conferences*, 2018.

[3] Camunda docs user guide. `https://docs.camunda.org/stable/guides/user-guide/`. [Online; accessed 23-August-2019].

[4] M. Dumas, M. La Rosa, J. Mendling, and H. A. Reijers. *Fundamentals of Business Process Management, Volume 1*. Springer, 2013.

[5] N. Slack, S. Chambers, and R. Johnston. *Operations and Process Management: principles and practice for strategic impact.* Pearson Education, 2009.

[6] Luise Pufahl and Mathias Weske. Batch activity: enhancing business process modeling and enactment with batch processing. *Computing*, 04 2019.

[7] J. Liu and J. Hu. *Dynamic batch processing in workflows: Model and Implementation*. Pearson EducationFuture Generation Computer Systems, 2007.

[8] Enabling batch processing in bpmn processes. `https://bpt.hpi.uni-potsdam.de/Public/BatchProcessing`. [Online; accessed 20-July-2019].

[9] Business process model and notation. `https://en.wikipedia.org/wiki/Business_Process_Model_and_Notation`. [Online; accessed 20-July-2019].