Camunda Best Practices

Handling Data in Processes

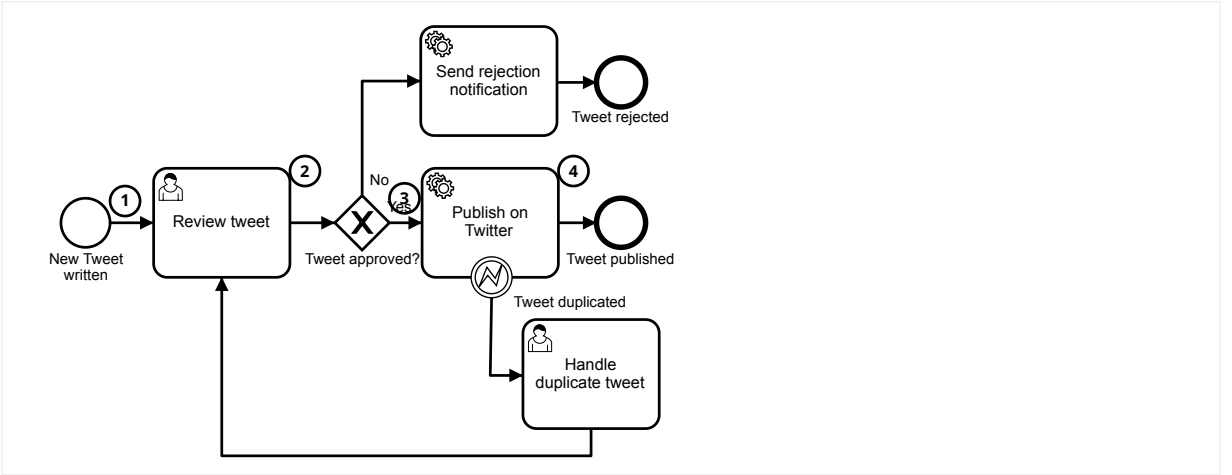
When using Camunda you have access to a dynamic map of "process variables", which lets you associate values/objects to every single process instance, plus one single "business key" per process instance. Make sure to use these mechanisms in a lightweight and meaningful manner storing just the relevant data in the process instance. Consider accessing your process variables in a type safe way centralizing (simple and complex) type conversion and using Constants for process variable names.

Feedback ? best-practices@camunda.com

Understanding Data in Camunda Processes

Dynamic Map of **Process Variables**

When using Camunda you always have access to a **dynamic map of "process variables"** which lets you associate values/objects to every single process instance (and local scopes in case of user tasks or parallel flows). When reading and interpreting a business process diagram, you quickly realize that there is always **data** necessary to work with **from a business perspective** but also to **drive the process** to the correct next steps. To give an example consider the **Tweet Approval Process** - a simple example process we use in various situations.



(1) The process instance starts with a freshly written **tweet** we need to remember.

2. We need to present this **tweet** so that the user can decide whether to **approve** it - or not.
3. The gateway needs to have access to this information: was the tweet **approved**?
4. In order to publish the tweet the service task again needs the **tweet** itself!

Therefore the Tweet Approval Process will need two variables:

Variable Name	Variable Type
tweet	(String)
approved	(Boolean)

You can **dynamically** create such variables by assigning an object of choice to a (string typed) variable name, e.g. by passing a `Map<String, Object>` when completing the "Review tweet" task via the API:

```java
Map<String, Object> variables = new HashMap<String, Object>();
variables.put("approved", true);
taskService.complete(taskId, variables);
```

In Camunda you do **not** declare process variables in the process model. This allows for a lot of flexibility. See recommendations below on how to overcome the disadvantages of this approach.

As the `tweet` variable was already created when starting the process, the Tweet Approval Process will now hold both variables mentioned above:

Variable Name	Variable Type	Variable Value
tweet	(String)	`"Have a look at the Tweet Approval Process!"`
approved	(Boolean)	`true`

Consult the User Guide [1] to learn more about process variables.

Camunda does not treat BPMN **data objects** () as process variables. We recommend to use them occasionally **for documentation**, but also to avoid excessive usage of data objects.

One (String type) **Business Key**

In addition to process variables, you can define **one single "business key" per process instance**. Its data type is always String. It has to be set directly when starting a process instance and **cannot be changed once set**. Note that it does not have to be unique, so you can have multiple process instances of the same definition with the same business key. However, we typically recommend keeping it unique for one process definition.

The main advantages a business key has over process variables are:

- **enhanced performance** - When reading the business key or querying for process instances using the business key, as it is a direct attribute of the process instance entity (hence: the database table).
- **enhanced visibility** - For example the business key is shown very prominently in the web applications shipping with Camunda, like e.g. Camunda Cockpit.

There are two typical use cases to leverage the business key:

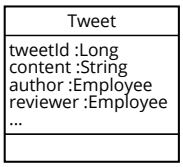
- The process basically represents the **"lifecycle"** of a **domain object** or has a similar strong and unique relationship to a domain object. The business key links it to exactly that object. A more detailed discussion about storing references only is found further below.
- You want to show one (or more) important business reference(s) very prominently in Camunda Cockpit. You could even create a compound business key to do so (e.g. "Customer 0815, Contract 42").

Storing Just the **Relevant Data**

Do not excessively use process variables. As a rule of thumb, store **as less few variables as possible** within Camunda.

Storing **References Only**

If you have leading systems already storing the business relevant data ...



then we suggest you store references only (e.g. ID's) to the objects stored there. So instead of holding the `tweet` and the `approved` variable, the process variables would now for example, look more like the following:

Variable Name	Variable Type	Variable Value
tweetId	(Long)	`8234`

Consider using the **business key** instead of a process variable if the process instance is bound to exactly one domain object as described above.

Use Cases for **Storing Payload**

Store **payload** (actual business data) as process variables, if you ...

☑ ... have data only of interest **within the process** (e.g. for gateway decisions).

In case of the Tweet Approval Process, even if you are using a tweet domain object, it might still be meaningful to hold the "approved" value explicitly as a process variable, because it serves the purpose to guide the gateway decision in the process. It might not be true if you want to keep track in the Tweet domain objects regarding the approval.

Variable Name	Variable Type	Variable Value
tweetId	(Long)	`8234`
approved	(Boolean)	`true`

☑ ... communicate in a **message oriented** style, e.g. retrieving data from one system and hand it over to another system via a process.

When receiving external **messages**, consider storing just those parts of the payload relevant for you, and not the whole response. This not only serves the goal of having a lean process variables map, it also makes you more independent from changes in the service's message interface.

☑ ... want to use the process engine as kind of **cache**, e.g. you cannot query relevant customer data in every step for performance reasons.

☑ ... need to **postpone data changes** in the leading system to a later step in the process, e.g. you only want to insert the Tweet in the Tweet Management Application if it is approved.

☑ ... want to track the **historical development** of the data going through your process.

☑ ... don't have a leading system for this data.

Serializing Complex Data

Simple objects (e.g. String, Boolean or Number) are stored in dedicated columns in the Camunda database. Complex objects have to be serialized in order to be stored in the generic database tables of Camunda. The default serialization option is Java byte code serialization (for historical reasons).

Avoid Java byte code serialization for storing complex data, because you can neither directly read Java byte code in the database nor do any database queries on the encrypted content. Furthermore, upgrades of the Java platform or even your classes may influence serializability.

Camunda offers **automatic serialization** of process variables to **XML or JSON**, implemented in the Camunda Spin [2] library.

We recommend configuring JSON serialization via Spin as default of your process engine [3].

You can **influence the process engine serialization** via JAXB or Jackson configurations, see the Data-format Configuration Example [4].

If you hit any limit you can still do the serialization yourself (e.g. in a Data Accessor, see next section). Then you have it full control of it.

Using **Constants** and Data **Accessors**

Avoid the copy/paste of string representations of your process variable names across your code base. **At least** collect the variable names for a process definition in a **Constants** interface/class:

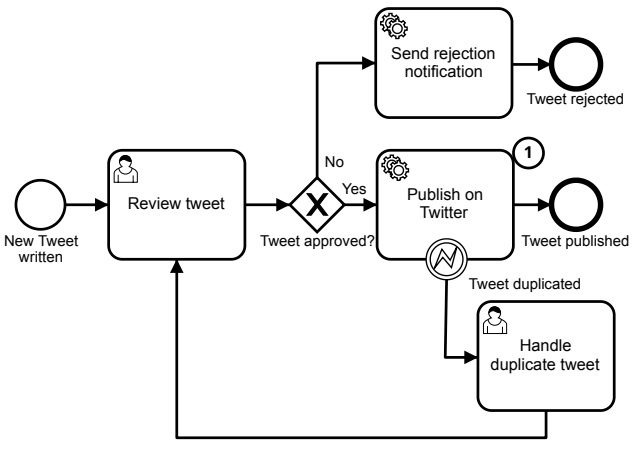
```java
public interface TwitterDemoProcessConstants {
  String VAR_NAME_TWEET = "tweet";
  String VAR_NAME_APPROVED = "approved";
}
```

This way, you have much more security against typos and can easily make use of refactoring mechanisms offered by IDEs.

However, if you also want to avoid spreading the necessary **type conversion (casting)** all over your application and want to have a natural place for **serializing your complex process variables**, we recommend that you use a **Data Accessor** class. It comes in two flavors:

- A **Process Data Accessor**: knows the names and types of all process variables of a certain process definition. It serves as the central point to declare variables for that process.
- A **Process Variable Accessor**: encapsulates the access to exactly one variable. This is useful if you reuse certain variables in different processes.

Consider for example the BPMN "Publish on Twitter" task in the Tweet Approval Process:



(1) We use a TweetPublicationDelegate to connect the "Publish on Tweet" task with a tweet publication service:

```java
@Named
public class TweetPublicationDelegate implements JavaDelegate {

  @Inject
  private TweetPublicationService tweetPublicationService;

  public void execute(DelegateExecution execution) throws Exception {
    String tweet = new TwitterDemoProcessVariables(execution).getTweet(); (1)
    // ...
```

(1) Here we instantiate a reusable class to access the `tweet` variable in a type safe way.

This reusable **Process Data Accessor** class could for example look like the following:

```java
public class TwitterDemoProcessVariables {

  private VariableScope variableScope;

  public TwitterDemoProcessVariables(VariableScope variableScope) { (1)
    this.variableScope = variableScope;
  }

  public String getTweet() { (2)
    return (String) variableScope.getVariable(TwitterDemoProcessConstants.VAR_NAME_TWEE
T);
  }

  public void setTweet(String tweet) {
        variableScope.setVariable(TwitterDemoProcessConstants.VAR_NAME_TWEET, tweet);
  }

  public Boolean isApproved() {
    return (Boolean) variableScope.getVariable(TwitterDemoProcessConstants.VAR_NAME_APPRO
VED);
  }

  public void setApproved(Boolean approved) {
        variableScope.setVariable(TwitterDemoProcessConstants.VAR_NAME_APPROVED, approve
d);
  }

}
```

(1) The accessor can be initialized with an `Execution` object or any other `VariableScope`

(2) The getters and setters take care of necessary casting and centralise serialisation and deserialisation for complex objects

Of course your specific implementation approach might differ, e.g. if using CDI you can leverage the magic around the BusinessProcess class as shown in the next example.

If you store references and often need to query the relevant data, you could add this logic to the Data Accessor as well - even if you need to think about lookup of the service reference. The following example uses CDI:

```java
public class CustomerDemoProcessVariables {

  @Inject
  private BusinessProcess businessProcess; (1)

  public String getCustomerId() {
    return (String) businessProcess.getVariable(
                    CustomerDemoProcessConstants.VAR_NAME_CUSTOMER_ID);
  }

  public void setCustomerId(String customerId) {
    businessProcess.setVariable(
                    CustomerDemoProcessConstants.VAR_NAME_CUSTOMER_ID,
                    customerId);
  }

  @Inject
  private CustomerDao customerDao;

  @Named
  @Produces (2)
  public Customer getCustomer() {
    return customerDao.findCustomer(getCustomerId());
  }

}
```

(1) When using CDI you can leverage the Business Process [5] instead of the Variable Context of the former example. This allows an empty constructor, but makes it harder to be used within test cases. You might want to build a version which can deal with the CDI BusinessProcess **and** the VariableScope within one class.

(2) Because of the @Named @Produces annotations this method is called to load the entity whenever using the bean "customer", e.g. in expressions in your BPMN XML as shown below. Don't worry, CDI makes sure it is handled efficiently when used multiple times within one transaction.

```xml
<sequenceFlow sourceRef="ExclusiveGateway" targetRef="SomeTask">
  <conditionExpression xsi:type="bpmn2:tFormalExpression">#{customer.status=='Approved'}
</bpmn2:conditionExpression>
</sequenceFlow>
```

It is an architectural decision if you want to allow this style or not.

Complex Data as **Entities**

There are some use cases when it is clever to **introduce entities alongside the process** to store complex data in a relational database. You can easily achieve this by using **JPA** (but are not limited to JPA!). You can see this logically as *typed process context" where you create custom tables for your custom process deployment.

You can use for example, Data Accessor classes to access these entities in a convenient manner.

There are a couple of advantages of this approach:

- You can do very **rich queries** on structured process variables. See Enhancing Tasklists with Business Data for an example of doing powerful tasklist queries with additional entities.
- You can apply a custom **data migration strategy** when deploying new versions of your process or services, which require data changes.
- Data can be designed/modeled properly, even graphically by e.g. leveraging UML.

Bonus 🎁 Checking Pre- and Postconditions

Todo: Add content refering to snippet/example

Links

[1] http://docs.camunda.org/manual/latest/user-guide/process-engine/variables/
[2] http://docs.camunda.org/manual/latest/user-guide/data-formats/configuring-spin-integration/
[3] https://docs.camunda.org/manual/7.9/user-guide/data-formats/json/#serializing-process-variables
[4] https://github.com/camunda/camunda-bpm-examples/tree/master/spin/dataformat-configuration
[5] https://docs.camunda.org/manual/7.9/reference/javadoc/?org/camunda/bpm/engine/cdi/BusinessProcess.html

Disclaimer & Copyright

Printed November 1, 2018. Applies to Camunda **7.9**. Any feedback? **best-practices@camunda.com**!