Error Handling

Error Handling Strategies

There are a couple of basic strategies to handle errors and exceptions within processes. The decision which strategy to use depends on:

- Technical vs. Business Errors: Does the error have some business meaning and causes an alternative process flow (like “item not on stock”) or is it a technical malfunction (like “network currently down”)?
- Explicit error handling or generic approach: For some situations you want to explicitly model what should happen in case of an error (typically for business errors). For a lot of situations you don’t want to do that but have some generic mechanism which applies for errors, simplifying your process models (typical for technical errors, imagine you would have to model network outage on every task were it might possibly occur? You wouldn’t be able to recognize your business process any more).

In the context of the process engine, errors are normally raised as Java exceptions which you have to handle. Let’s have a look at how to handle them.

Transaction Rollbacks

The standard handling strategy is that exceptions are thrown to the client, meaning that the current transaction is rolled back. This means that the process state is rolled back to the last wait state. This behavior is described in detail in the Transactions in Processes section of the User Guide. Error handling is delegated to the client by the engine.

Let’s show this in a concrete example: the user gets an error dialog on the frontend stating that the stock management software is currently not reachable due to network errors. To perform a retry, the user might have to click the same button again. Even if this is often not desired it is still a simple strategy applicable in a lot of situations.

Async and Failed Jobs

If you don’t want the exception being shown to the user, one option is to make service calls, which might cause an error, async (as described in Transactions in Processes). In that case the exception is stored in the process engine database and the Job in the background is marked as failed (to be more precise, the exception is stored and some retry counter is decremented).

In the example above this means that the user will not see an error but an “everything successful” dialog. The exception is stored on the job. Now either a clever retry strategy will automatically re-trigger the job later on (when the network is available again) or an operator needs to have a look at the error and trigger an additional retry. This is shown later in more detail.

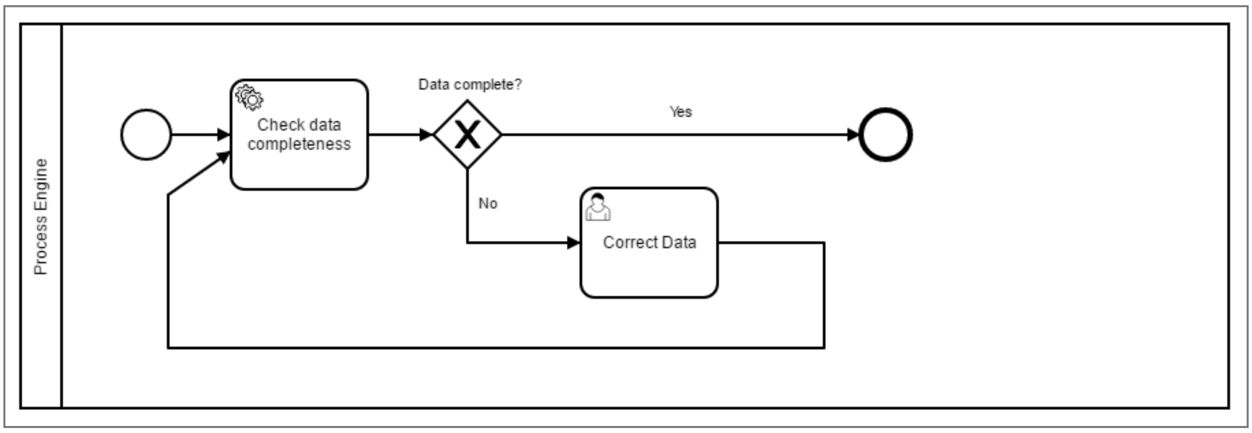
This strategy is pretty powerful and applied often in real-life projects, however, it still hides the error in the BPMN diagram, so for business errors which you want to be visible in the process diagram, it would be better to use Error Events.

Catch Exception and use Data Based XOR-Gateway

If you call Java Code which can throw an exception, you can catch the exception within the Java Delegate, CDI Bean or whatsoever. Maybe it is already sufficient to log some information and go on, meaning that you ignore the error. More often you write the result into a process variable and model an XOR-Gateway later in the process flow to take a different path if that error occurs.

In that case you model the error handling explicitly in the process model but you make it look like a normal result and not like an error. From a business perspective it is not an error but a result, so the decision should not be made lightly. A rule of thumb is that results can be handled this way, exceptional errors should not. However, the BPMN perspective does not always have to match the technical implementation.

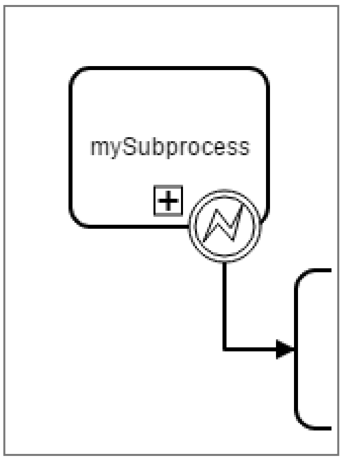
Example:



We trigger a “check data completeness” task. The Java Service might throw a “DataIncompleteException”. However, if we check for completeness, incomplete data is not an exception, but an expected result, so we prefer to use an XOR-Gateway in the process flow which evaluates a process variable, e.g., “#{dataComplete==false}”.

BPMN 2.0 Error Event

The BPMN 2.0 error event gives you the possibility to explicitly model errors, tackling the use case of business errors. The most prominent example is the “intermediate catching error event”, which can be attached to the boundary of an activity. Defining a boundary error event makes most sense on an embedded subprocess, a call activity or a Service Task. An error will cause the alternative process flow to be triggered:



See the Error Events section of the BPMN 2.0 Implementation Reference and the Throwing Errors from Delegation Code section of the User Guide for more information.

BPMN 2.0 Compensation and Business Transactions

BPMN 2.0 transactions and compensations allow you to model business transaction boundaries (however, not in a technical ACID manner) and make sure already executed actions are compensated during a rollback. Compensation means to make the effect of the action invisible, e.g. book in goods if you have previously booked out the goods. See the BPMN Compensation event and the BPMN Transaction Subprocess sections of the BPMN 2.0 Implementation Reference for details.

Monitoring and Recovery Strategies

In case the error occurred, different recovery strategies can be applied.

Let the User Retry

As mentioned above, the simplest error handling strategy is to throw the exception to the client, meaning that the user has to retry the action himself. How he does that is up to the user, normally reloading the page or clicking again.

Retry Failed Jobs

If you use Jobs (`async`), you can leverage Cockpit as monitoring tool to handle failed jobs, in this case no end user sees the exception. Then you normally see failures in cockpit when the retries are depleted (see the Failed Jobs section of the Web Applications for more information).

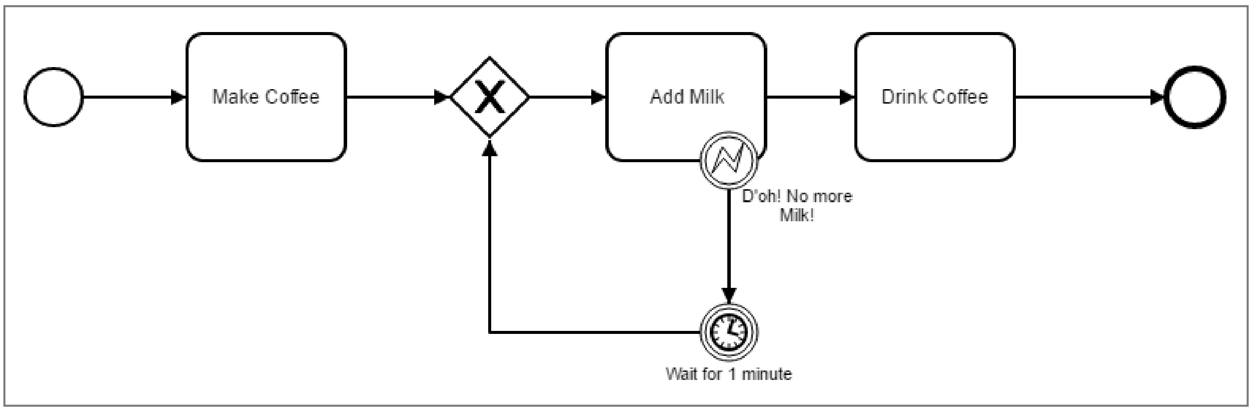
See the Failed Jobs in Cockpit section of the Web Applications for more details.

If you don’t want to use Cockpit, you can also find the failed jobs via the API yourself:

```
List<Job> failedJobs = processEngine.getManagementService().createJobQuery().withExcep
for (Job failedJob : failedJobs) {
  processEngine.getManagementService().setJobRetries(failedJob.getId(), 1);
}
```

Explicit Modeling

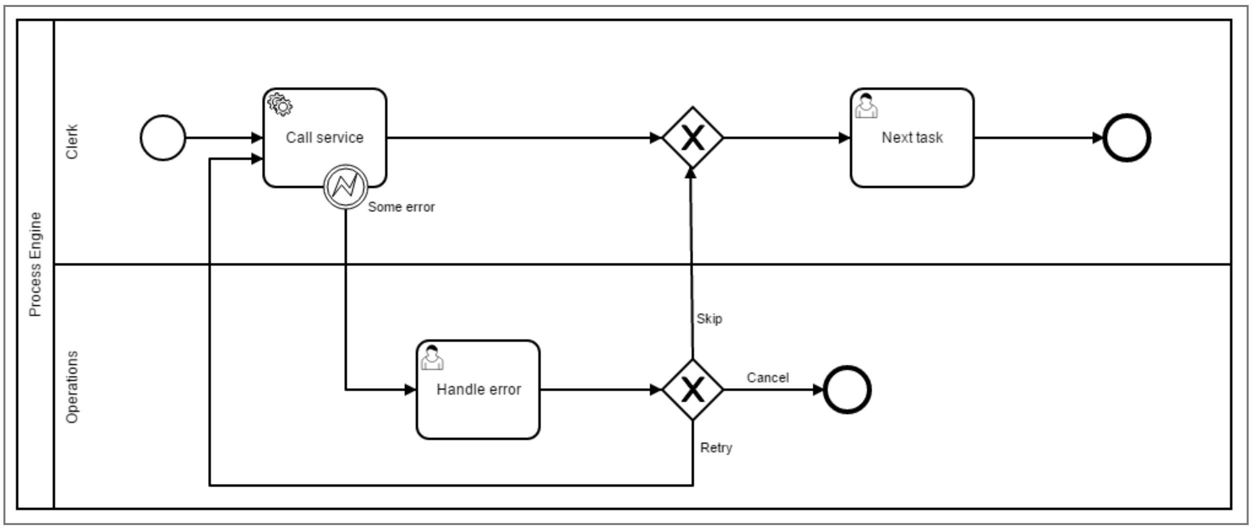
Of course you can always explicitly model a retry mechanism as pointed out in Where is the retry in BPMN 2.0:



We would recommend to limit it to cases where you want to see it in the process diagram for a good reason. We prefer asynchronous continuation, as it doesn't bloat your process diagram and basically can do the same thing with even less runtime overhead, as "walking" through the modeled loop involves additional action, e.g., writing an audit log.

User Tasks for Operations

We often see something like this in projects:



Actually this is a valid approach in which you assign errors to an operator as User Tasks and model what options he has to solve the problem. However, this is a strange mixture: We want to handle a technical error we but add it to our business process model. Where do we stop? Do we have to model it on every Service Task now?

Having a failed jobs list instead of using the "normal" task list feels like a more natural approach for this situation, which is why we normally recommend the other possibility and do not consider this to be best practice.

Exception codes

Sometimes an API call doesn't succeed because a problem occurs. The Java programming model uses exceptions to handle these situations. Exceptions that occur on the process engine's application level are of the type `ProcessEngineException` .

Here are two examples of everyday situations in which the engine throws a `ProcessEngineException`:

1. You cannot start a process instance since the variable's value is too long.

2. Two users in parallel complete the same task.

You can read the exception message to understand the reason for a `ProcessEngineException`. However, sometimes the message of the top-level exception is too generic. In these situations, the cause might contain a more insightful exception message. Traversing through exception causes might be tedious. Also, causes are unavailable when an error occurs on the REST API level.

While reading the error message might help users to understand the root cause of the problem, evaluating exception messages in an automated way is not a good idea since:

- The message might change with newer versions.
- Relying on fragments of the message can be error-prone.

This is why we introduced static exception codes your business logic can rely on to determine specific problems and react accordingly.

You can access error codes via Java as well as REST API.

Built-in codes

We identified common situations in which the engine throws an exception and assigned a built-in error code to the exception. You can look up the built-in codes in the Categories, ranges, and codes section.

Custom codes

Sometimes you may want to assign codes to specific errors Camunda hasn’t covered so far. You can either define custom codes from delegation code or by registering your custom `ExceptionCodeProvider`.

Delegation code

Learn more on how to assign a custom error code to an exception in the documentation about Delegation Code.

Configuration

You can configure the exception error codes feature in your process engine configuration:

- To disable the exception codes feature entirely, set the flag `disableExceptionCode` in your process engine configuration to `true`.
- To disable the built-in exception code provider, set the flag `disableBuiltinExceptionCodeProvider` in your process engine configuration to `true`. Disabling the built-in exception code provider allows overriding the reserved code range with your custom exception codes.

Register a Custom Code Provider

With the help of a `ProcessEnginePlugin` you can register a custom `ExceptionCodeProvider` :

```
engineConfig.setCustomExceptionCodeProvider(new ExceptionCodeProvider() {

  @Override
  public Integer provideCode(ProcessEngineException processEngineException) {
    // Put your business logic here to determine the
    // error code in case a process engine exception was thrown.
    return 22_222;
  }

  @Override
  public Integer provideCode(SQLException sqlException) {
    // Put your business logic here to determine the
    // error code in case a sql exception was thrown.
    return 33_333;
  }

});
```

Heads-up!

If your custom error code violates the reserved code range, it will be overridden with `0` unless you disable the built-in code provider.

Categories, ranges, and codes

In the table below, you will find an overview of all categories, ranges, and codes:

Category	Range	Code	Description	Safe to retry
Fallback		0	All errors with no code assigned.	
Engine	[1, 9999]	1	`OptimisticLockingException`	X
Persistence	[10000, 19999]	10,000	Deadlock situation occurred.	X
		10,001	A foreign key constraint was violated.	
		10,002	The column size is too small.	
Custom	[20000, 39999]	*E.g., 22,222*	*E.g., custom `JavaDelegate` validation error.*	

Reserved code range

The codes <= 19,999 and >= 40,000 are reserved for built-in codes. If you disable the built-in code provider, you can also use the reserved code range for your custom codes.