Migration-ready solutions

To implement Camunda 7 process solutions that can be easily migrated, follow these rules and development practices.

Overview

These practices might also inform a refactoring step to prepare your existing Camunda 7 solution for migration:

- Implement what we call Clean Delegates - concentrate on reading and writing process variables, plus business logic delegation. Data transformations will be mostly done as part of your delegate (and especially not as listeners, as mentioned below). Separate your actual business logic from the delegates and all Camunda APIs. Avoid accessing the BPMN model and invoking Camunda APIs within your delegates.
- Use primitive variable types or JSON payloads only (no XML or serialized Java objects).
- Use simple expressions or plug-in FEEL. FEEL is the only supported expression language in Camunda 8. JSONPath is also relatively easy to translate to FEEL. Avoid using special variables in expressions, for example `execution` or `task`.
- Use your own user interface for task forms or Camunda Forms; the other form mechanisms are not supported out of the box in Camunda 8.
- Don't rely on an ACID transaction manager spanning multiple steps or resources.
- Don't expose Camunda APIs (REST or Java) to other services or frontend applications.
- Don't call Spring beans in expressions (for example to leverage Java code to do data transformations).
- Avoid using any implementation classes from Camunda; generally, those with `\*.impl.\*` in their package name.
- Avoid using process engine plugins.
- Avoid using Cockpit plugins.

Clean delegates

Given that Java delegates and the workflow engine are embedded as a library, projects can do dirty hacks in their code. Casting to implementation classes? No problem. Using a ThreadLocal or trusting a specific transaction manager

implementation? Yeah, possible. Calling complex Spring beans hidden behind a simple Java Unified Expression Language (JUEL) expression? Well, you guessed it — doable!

Those hacks are the real showstoppers for migration, as they cannot be migrated to Camunda 8. In fact, Camunda 8 increased isolation intentionally.

Concentrate on what a Java delegate is intended to do:

1. Read variables from the process and potentially manipulate or transform that data to be used by your business logic.
2. Delegate to business logic — this is where Java delegates got their name from. In a perfect world, you would simply issue a call to your business code in another Spring bean or remote service.
3. Transform the results of that business logic into variables you write into the process.

The following is an example of a good Java delegate:

```
@Component
public class CreateCustomerInCrmJavaDelegate implements JavaDelegate {

  @Autowired
  private CrmFacade crmFacade;

  public void execute(DelegateExecution execution) throws Exception {
    // Data Input Mapping
    CustomerData customerData = (CustomerData)
execution.getVariable("customerData");

    // Delegate to business logic
    String customerId = crmFacade.createCustomer(customerData);

    // Data Output Mapping
    execution.setVariable("customerId", customerId);
  }
}
```

Never cast to Camunda implementation classes, use any ThreadLocal object, or influence the transaction manager in any way. Java delegates should always be stateless and not store any data in their fields.

Such a delegate can be easily migrated according to our code conversion patterns, for example using OpenRewrite recipes.

No transaction managers

You should not trust ACID transaction managers to glue together the workflow engine with your business code.

- Instead, embrace eventual consistency and make every service task its own transactional step. If you are familiar with Camunda 7 lingo, this means that all BPMN elements will be `async=true`.
- A process solution that relies on five service tasks to be executed within one ACID transaction, probably rolling back in case of an error, will make migration challenging.

Don’t expose Camunda API

You should apply the information hiding principle and not expose too much of the Camunda API to other parts of your application.

In the following example, you should not hand over an execution context to your `CrmFacade`:

```
// DO NOT DO THIS!
crmFacade.createCustomer(execution);
```

The same holds true when a new order is placed, and your order fulfillment process should be started. Instead of the frontend calling the Camunda API to start a process instance, provide your own endpoint to translate between the inbound REST call and Camunda.

For example:

```
@RestController
public class OrderFulfillmentRestController {

 @Autowired
 private ProcessEngine camunda;

 @RequestMapping(path = "/order", method = POST)
 public ResponseEntity<StatusDto> placeOrder(@RequestBody OrderDto
orderPayload) throws Exception {
  // TODO: Somehow extract data from orderPayload
  OrderData orderData = OrderData.from(orderPayload);

  ProcessInstance pi = camunda.getRuntimeService()
```

```
      .startProcessInstanceByKey("orderFulfillment", Variables.putValue("order",
orderData));

  response.setStatus(HttpServletResponse.SC_ACCEPTED);
  return ResponseEntity.accepted().body(StatusDto.of("pending"));
 }
}
```

Use primitive variable types or JSON

Camunda 7 provides flexible ways to add data to your process. For example, you could add Java objects that would be serialized as byte code. Java byte code is brittle and also tied to the Java runtime environment.

Another possibility is transforming those objects on the fly to JSON or XML using Camunda Spin. It turned out this was black magic and led to regular problems, which is why Camunda 8 does not offer this any more. Instead, you should perform any transformation within your code before communicating with the Camunda API. Camunda 8 only takes JSON as a payload, which automatically includes primitive values.

In the following Java delegate example, you can see Spin and Jackson were used in the delegate for JSON to Java mapping:

```
@Component
public class CreateCustomerInCrmJavaDelegate implements JavaDelegate {

  @Autowired
  private ObjectMapper objectMapper;
  //...

  public void execute(DelegateExecution execution) throws Exception {
    // Data Input Mapping
    JsonNode customerDataJson = ((JacksonJsonNode)
execution.getVariable("customerData")).unwrap();
    CustomerData customerData = objectMapper.treeToValue(customerDataJson,
CustomerData.class);
    // ...
  }
}
```

This way, you have full control over what is happening, and such code is also easily migratable. The overall complexity is even lower, as Jackson is quite well known to

Java people — a kind of de-facto standard with a lot of best practices and recipes available.

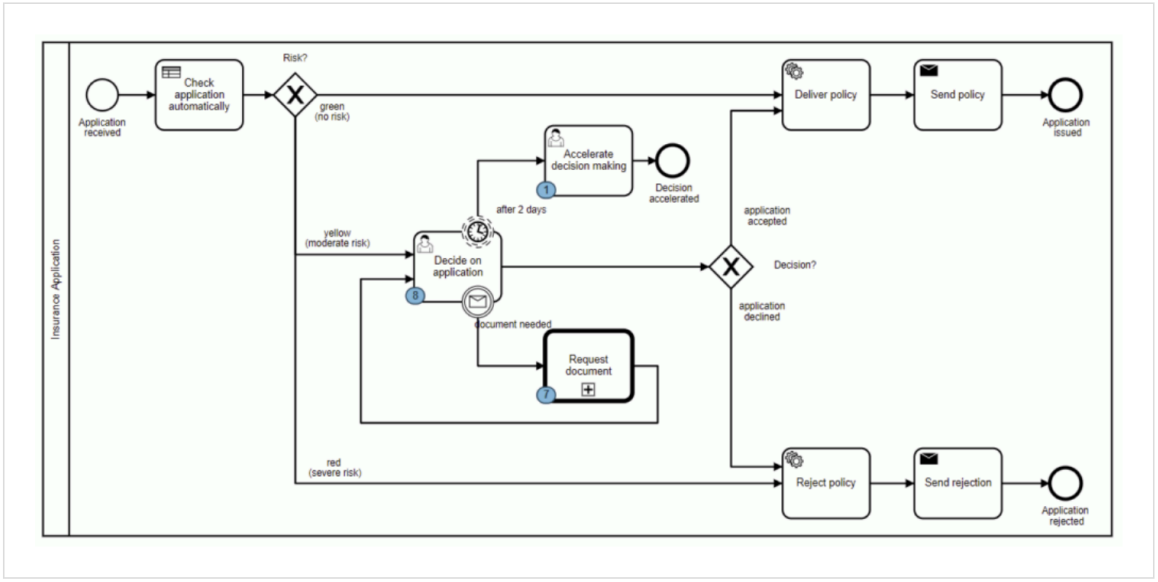
Simple expressions and FEEL

Camunda 8 uses FEEL as its expression language. There are big advantages to this decision. Not only are the expression languages between BPMN and DMN harmonized, but also the language is really powerful for typical expressions. One of my favorite examples is the following onboarding demo we regularly show. A decision table will hand back a list of possible risks, whereas every risk has a severity indicator (yellow, red) and a description.

Risk Assessment

C	Input			Output		Annotation
	Age = 37	Car manufacturer = Porsche	Car type = 911	Risk	Risk assesment	
	integer	string	string	string	string	
1	<= 21	-	-	"Beginner"	"yellow"	
2	<= 25	"Porsche"	-	"Young and too fast"	"red"	No sorry - that's too risky!
3	<= 30	"BMW"	-	"Young and fast"	"yellow"	
4		"Porsche"	"911"	"Fast and furious" = Fast and furious	"yellow" = yellow	
5		"BMW"	"X3"	"High value vehicle"	"yellow"	

The result of this decision will be used in the process to make a routing decision:



To unwrap the DMN result in Camunda 7, you could write some Java code and attach that to a listener when leaving the DMN task (this is already an anti-pattern for migration as you will read next). The code is not very readable:

```
@Component
public class MapDmnResult implements ExecutionListener {

@Override
public void notify(DelegateExecution execution) throws Exception {
  List<String> risks = new ArrayList<String>();
  Set<String> riskLevels = new HashSet<String>();

  Object oDMNresult = execution.getVariable("riskDMNresult");
  for (Object oResult : (List<?>) oDMNresult) {
    Map<?, ?> result = (Map<?, ?>) oResult;
    risks.add(result.containsKey("risk") ? (String) result.get("risk") : "");
    if (result.get("riskLevel") != null) {
      riskLevels.add(((String) result.get("riskLevel")).toLowerCase());
    }
  }

  String accumulatedRiskLevel = "green";
  if (riskLevels.contains("rot") || riskLevels.contains("red")) {
    accumulatedRiskLevel = "red";
  } else if (riskLevels.contains("gelb") || riskLevels.contains("yellow")) {
    accumulatedRiskLevel = "yellow";
  }

  execution.setVariable("risks",
Variables.objectValue(risks).serializationDataFormat(SerializationDataFormats.JSO
N).create());
  execution.setVariable("riskLevel", accumulatedRiskLevel);
}
}
```

With FEEL, you can evaluate that data structure directly and have an expression on the "red" path:

```
= some risk in riskLevels satisfies risk = "red"
```

Additionally, you can even hook in FEEL as the scripting language in Camunda 7 (as explained in Scripting with DMN inside BPMN or User Task Assignment based on a DMN Decision Table).

However, more commonly you will keep using JUEL in Camunda 7. If you write simple expressions, they can be migrated automatically, as you can see in the test case of the migration community extension. You should avoid more complex expressions if possible.

Very often, a good workaround to achieve this is to adjust the output mapping of your Java delegate to prepare data in a form that allows for easy expressions.

Avoid hooking in Java code during an expression evaluation. The above listener to process the DMN result was one example of this, but a more diabolic example could be the following expression in Camunda 7:

```
// DON'T DO THIS:
#{ dmnResultChecker.check( riskDMNresult ) }
```

Now, the `dmnResultChecker` is a Spring bean that can contain arbitrary Java logic, possibly even querying some remote service to query whether we currently accept yellow risks or not. Such code cannot be executed within Camunda 8 FEEL expressions, and the logic needs to be moved elsewhere.

Camunda Forms

Finally, while Camunda 7 supports different types of task forms, Camunda 8 only supports Camunda Forms (and will actually be extended over time). If you rely on other form types, you either need to make Camunda Forms out of them or use a bespoke tasklist where you still support those forms.