Decision Management Community – February 2021 Challenge

Background

The purpose of this challenge is to compare the performance of different decision engine implementations. The problem has one large decision table and the intent is to build and compare implementations, ideally based on a serverless architecture.

Approach

In this implementation, the solution is based on the Camunda DMN engine deployed as an AWS lambda function. The first step was to convert the supplied rules as a CSV file into a DMN decision table. This is readily achieved as the Camunda Modeller has an optional plugin to import an Excel file and create a DMN decision table. The CSV file was loaded into an Excel spreadsheet, followed by a few text based manipulations to trim whitespace, add quotes in appropriate places and add some basic FEEL date expressions. A snippet of the resulting DMN table is shown below.

Decide Medical Services | **Hit Policy:** First

	When	And	And	And	And	And	And
	PlaceOfService string	ServiceType string	Plan string	GroupSize string	InNetwork string	IsCovered string	ServiceDate FEEL-Date
1	"Inpatient"	"dentalAccidental"	"PL123"	"L"	"Y"	"Y"	[date("2015-01-01")..date("2023-12-31")]
2	"Outpatient"	"dentalAccidental"	"PL123"	"L"	"Y"	"Y"	[date("2015-01-01")..date("2023-12-31")]
3	"Office"	"dentalAccidental"	"PL123"	"L"	"Y"	"Y"	[date("2015-01-01")..date("2023-12-31")]
4	"Inpatient"	"dentalAccidental"	"PL123"	"L"	"N"	"Y"	[date("2015-01-01")..date("2023-12-31")]
5	"Outpatient"	"dentalAccidental"	"PL123"	"L"	"N"	"Y"	[date("2015-01-01")..date("2023-12-31")]

Implementation

The next step was to build the lambda function implementation using the Camunda DMN engine and this DMN table. This implementation approach required two main implementation parts. The first part initializes the DMN engine and loads and parses the decision table as static class initialisers. This code snippet is shown below;

```
//
// Initialise a DMN engine
//
static final DmnEngine dmnEngine =      DmnEngineConfiguration.createDefaultDmnEngineConfiguration().buildEngine();

//
// Load and parse the DRG model
//
static final DmnDecisionRequirementsGraph drg =
dmnEngine.parseDecisionRequirementsGraph(MedicalServicesFunctionHandler.class.getResourceAsStream("/MedicalServices.dmn"));
static final DmnDecision decision = drg.getDecision("DecideMedicalServices");
```

The main lambda function handler needs to un-marshal request parameters, call the decision and marshal the response. This code snippet is shown below (the full method is shown in appendix A);

```
try {
```

```
        //
        // Extract the request variables (json payload) from the body attribute...
        //
        HashMap<String, Object> event = gson.fromJson(reader, HashMap.class);
        HashMap<String, Object> variables = gson.fromJson(event.get("body").toString(), HashMap.class);

        //
        // Load request variables into response
        responseBody = new JSONObject(variables);

        //
        // call the decision engine with the variables...
        //
        long start=System.currentTimeMillis();
        DmnDecisionTableResult results = dmnEngine.evaluateDecisionTable(decision, variables);
        long finish=System.currentTimeMillis();

        //
        // Setup the response - note body attribute must be a JSON string...
        //
        response.put("statusCode", 200);
        Iterator<Entry<String, Object>> it = results.getSingleResult().entrySet().iterator();
        while (it.hasNext()) {
                Entry<String,Object> entry = it.next();
                responseBody.put(entry.getKey(),entry.getValue().toString());
        }
        responseBody.put("duration",finish-start);
        response.put("body", responseBody.toJSONString());

        writer.write(response.toString());
        writer.flush();

        if (writer.checkError()) {
                logger.log("WARNING: Writer encountered an error.");
        }
}
```

In addition to the input and output variables, this solution includes the actual execution time inside the decision engine as an output attribute (duration) in the Json response.

Performance

The solution is deployed as an AWS lambda function. An initial request typically takes around 5 seconds as the AWS lambda implementation needs to be instantiated and deployed. As the function is exercised, runtime JVM optimisations rapidly bring the runtime down. After an initial warmup period, a run of 10,000 samples across the 10 test cases was performed. The round trip including network, marshalling and decision execution for each sample came to;

Average (ms)	Min (ms)	Max (ms)	STD. Deviation
282	139	462	15.28

The table above includes the network and data marshalling overheads. In looking at the duration attribute in the response which reflects the time in the decision engine, the average decision execution time is around 235 ms.

Performance Improvement

Given this is essentially a relatively static lookup table, if performance was critical, an API cache can be used to improve response time.

In this implementation a read through cache is readily enabled via a few clicks in the AWS console. A read through cache uses the inbound request as a key and caches the response for a configurable duration or time to live (TTL). In an inbound request matches a key in the cache, a previously computed result is returned. Otherwise the decision service computation is run, the response added to the cache and the result returned. If a new version of the decision table is deployed, the cache can be flushed as part of the deployment process such that a stale result is never returned. With the cache enabled, the response time for a cached result drops to an average of 34ms inclusive of network overheads as shown in the table below.

Average (ms)	Min (ms)	Max (ms)	STD. Deviation
34	30	199	6.73

Build

This solution is readily built and deployed using a standard Java IDE by generating an AWS lambda project, using the code in Appendix A as the handler function , the DMN model, and deploying the resulting build using an AWS console to create a lambda function.

Run AWS Lambda from POSTMAN

You may execute this decision service deployed as AWS Lambda from POSTMAN using the following POST endpoint URL. The cache is enabled. The response includes a duration attribute which reflects the time spent computing the decision result. If the response time is less than the duration, the result will have been retrieved from cache. One way to force a cache miss is to generate a request using values not present in the decision table as this request key will typically not be in the cache and this request would exercise the decision service (the response will have “ERR” in the output attributes indicating no rule match was found). A cached result will eventually be evicted from the cache as the TTL in the cache is configured to be about an hour.

https://9jia1nrk1h.execute-api.ap-southeast-2.amazonaws.com/default/MedicalServices

and the following JSON input body


```
{
  "placeOfService":"Inpatient",
  "type":"adultImmunizations",
  "plan":"PL123",
  "groupSize":"L",
  "inNetwork":"Y",
  "isCovered":"Y",
  "dateOfService":"2019-03-25",
  "coveredInFull":null,
  "copay":null,
  "coInsurance":null
}
```


Here is an example of the produced result:


```
{
  "duration":234,
  "inNetwork":"Y",
```

```
  "groupSize":"L",
  "dateOfService":"2019-03-25",
  "copay":"Copay Minimal",
  "coInsurance":"N",
  "placeOfService":"Inpatient",
  "type":"adultImmunizations",
  "coveredInFull":"N",
  "plan":"PL123",
  "isCovered":"Y"
}
```

Appendix A – Lambda Function Handler

```
	@SuppressWarnings("unchecked")
	@Override
	public void handleRequest(InputStream inputStream, OutputStream outputStream, Context context) throws
IOException {

		LambdaLogger logger = context.getLogger();

		//
		// Create readers and writers for the input and output streams
		//
		BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream,
Charset.forName("US-ASCII")));
		PrintWriter writer = new PrintWriter(
				new BufferedWriter(new OutputStreamWriter(outputStream, Charset.forName("US-
ASCII"))));

		//
		// Response object initialisation
		//
		JSONObject response = new JSONObject();
		JSONObject responseBody = null;

		try {
			//
			// Extract the request variables (json payload) from the body attribute...
			//
			HashMap<String, Object> event = gson.fromJson(reader, HashMap.class);
			HashMap<String, Object> variables = gson.fromJson(event.get("body").toString(),
HashMap.class);

			//
			// Load request variables into response
			responseBody = new JSONObject(variables);

			//
			// call the decision engine with the variables...
			//
			long start=System.currentTimeMillis();
			DmnDecisionTableResult results = dmnEngine.evaluateDecisionTable(decision, variables);
			long finish=System.currentTimeMillis();

			//
			// Setup the response - note body attribute must be a JSON string...
			//
			response.put("statusCode", 200);
			Iterator<Entry<String, Object>> it = results.getSingleResult().entrySet().iterator();
			while (it.hasNext()) {
				Entry<String,Object> entry = it.next();
				responseBody.put(entry.getKey(),entry.getValue().toString());
			}
			responseBody.put("duration",finish-start);
			response.put("body", responseBody.toJSONString());

			writer.write(response.toString());
			writer.flush();

			if (writer.checkError()) {
				logger.log("WARNING: Writer encountered an error.");
			}
		} catch (Exception exception) {
			logger.log(exception.toString());
			response.put("statusCode", 400);
			responseBody.put("exception", exception.getMessage());
			response.put("body", responseBody.toJSONString());

			writer.write(response.toString());
			writer.flush();

		} finally {
			reader.close();
			writer.close();
		}
	}
```