Fabian Bouché, January 1st 2019

Springboot - Activiti Proof of Concept

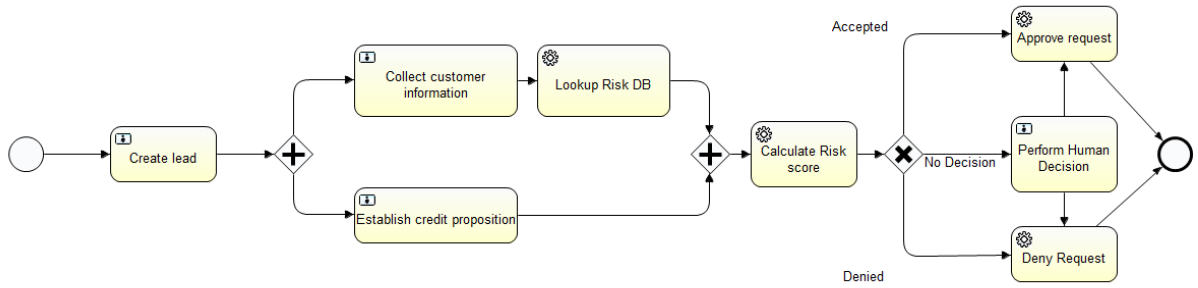
Introduction

Activity is a BPM engine able to orchestrate human and automated tasks inside of a workflow.

As such, it can be used to orchestrate processes which have some of the following characteristics:

- Termination of the process may require a human validation
- The process requires actors to provide information / documents

We can use the credit application process as an example. Below a simplified example of what it looks like:



- This process requires the execution of some human tasks (👤) in parallel.
- They can be performed in whatever order but they all need to be completed to perform the automated task (⚙) “Calculate Risk score”.
- However, the “Lookup risk DB” task can be triggered as soon as “Collect customer information” has been completed *(this task may take some time and block the process if postponed upon total completion of the credit application).*

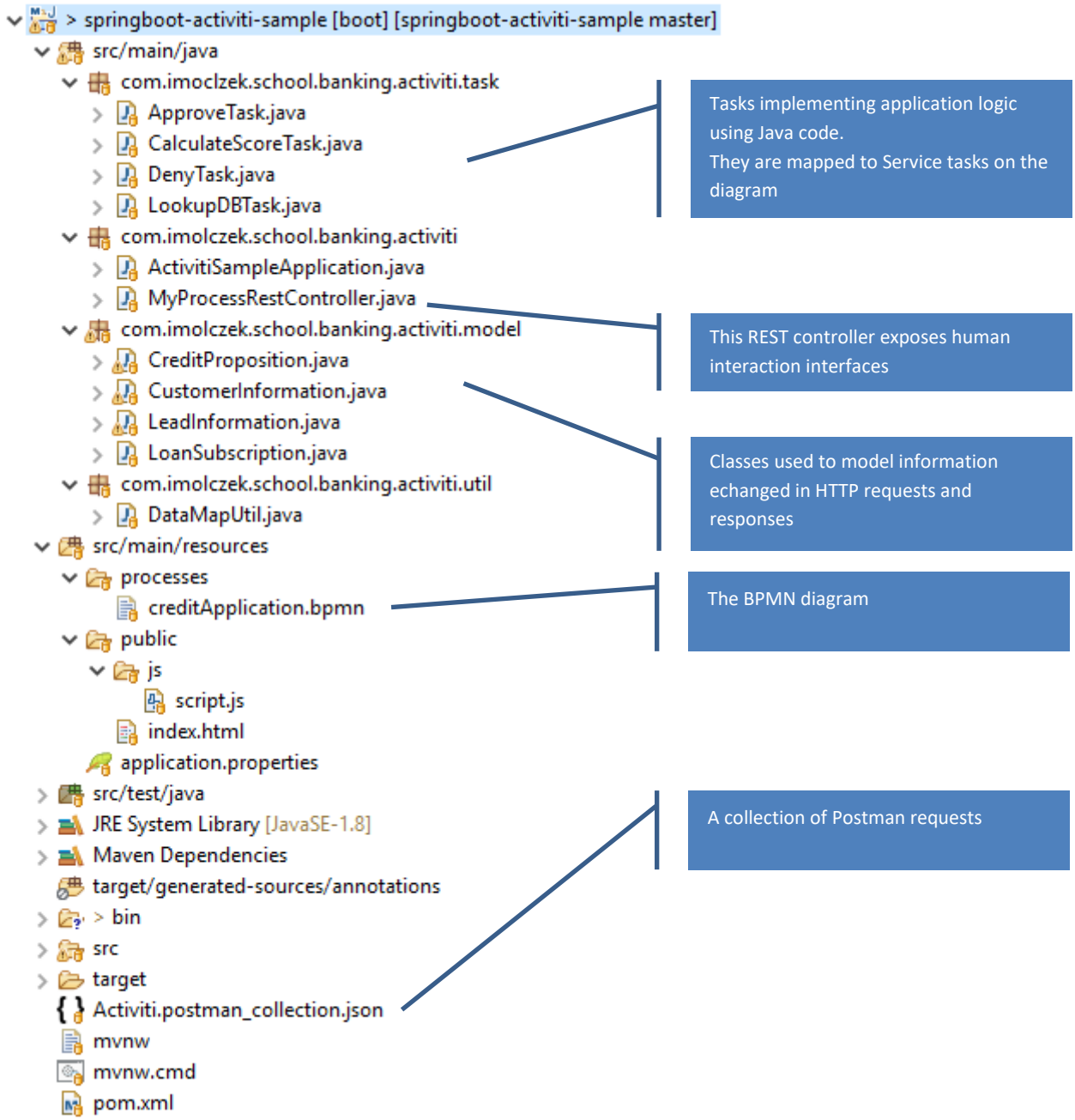
Setup of the environment

The following components have been setup for the POC:

- Eclipse IDE / Spring Tool Suite
- Activiti Designer plugin (installed using the Eclipse “Install new software” feature)
    - This plugin adds the following capabilities to design BPMN diagrams in Eclipse

Development of a Custom process

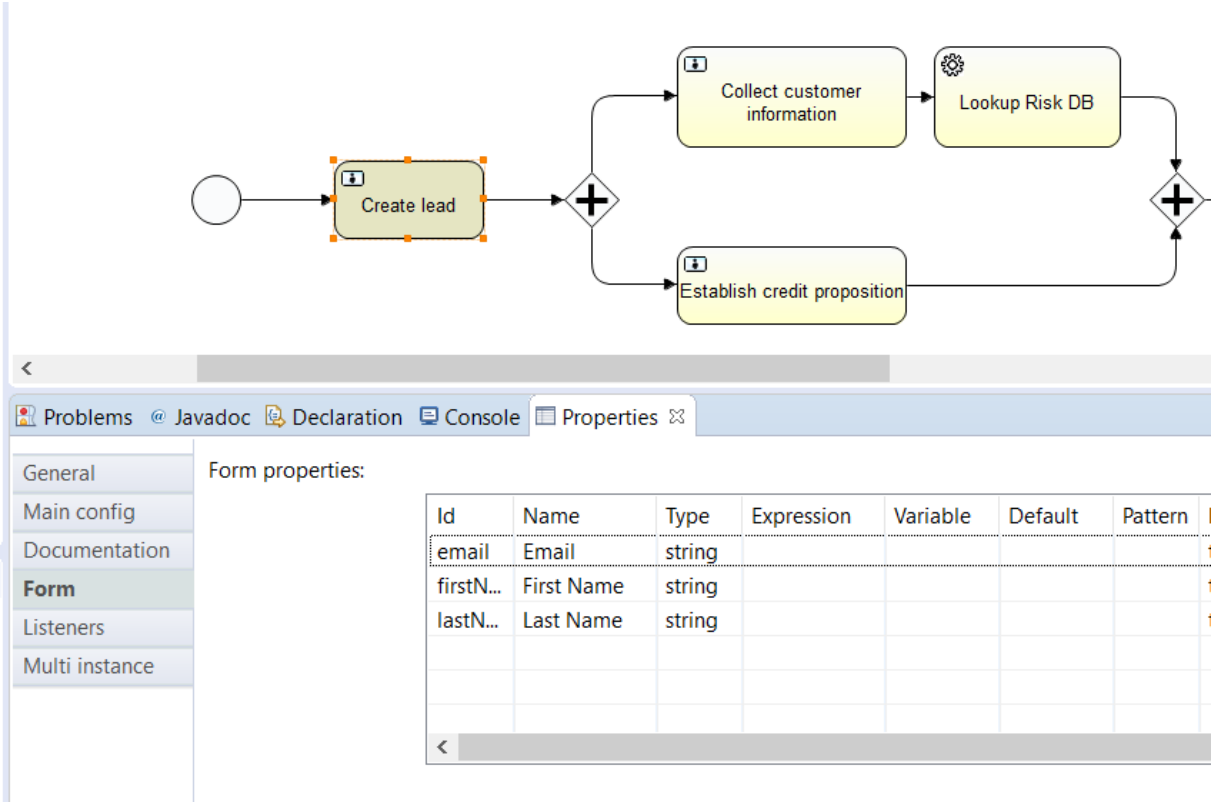
In order to test Activiti, I've modelled a dummy Credit Application process (see illustration in "Introduction"). Below is the project structure:



To help you setup the project, fork the following repository.

https://github.com/raoul-imolczek/springboot-activiti-empty

The BPMN editor allows the process author to define the information which has to be collected for each human task:



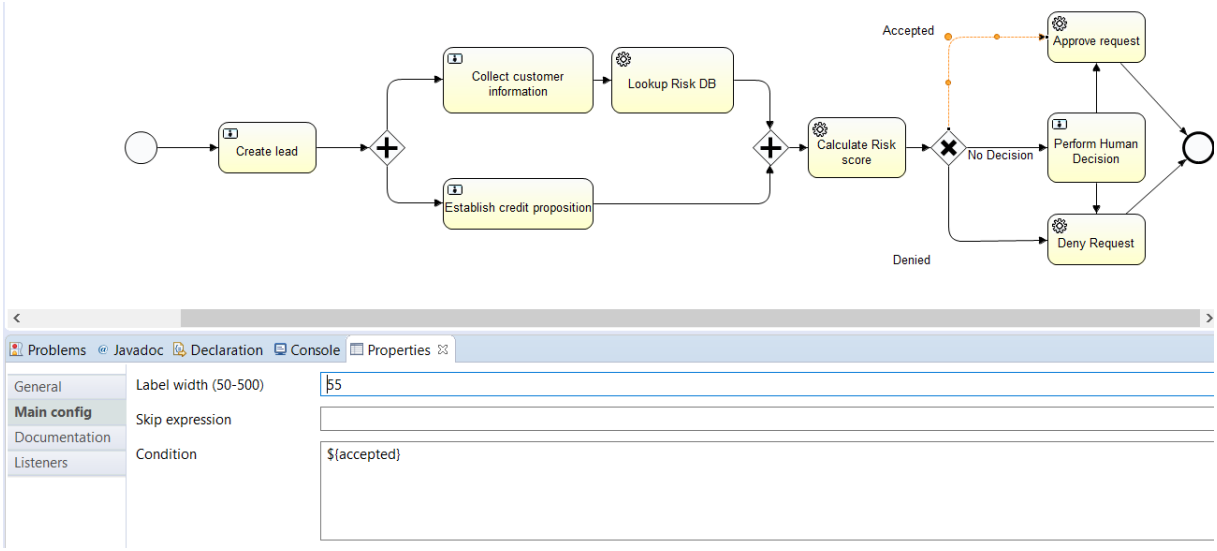
Below an example of an automated task:

creditApplication | ExampleTask.java

```java
package com.imoclzek.school.banking.activiti.task;

import org.activiti.engine.delegate.DelegateExecution;

public class ExampleTask implements JavaDelegate {

    private Logger LOG = Logger.getLogger(ExampleTask.class);

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

        // Read some variable
        String readVariable = (String) execution.getVariable("readVariable");
        LOG.info("Read value " + readVariable);

        // Set some variable value
        execution.setVariable("writeVariable", "value");

    }

}
```

Process variables can be retrieved and set using DelegateExecution.getVariable and DelegateExecution.setVariable.

Those same variables can be used inside of the BPM definition. Here, the status variable is used to decide which branch to follow after the credit request has been recorded:



Eventually, in order to allow users to interact with the process, let's write a controller with methods allowing to start a process, submit a form, complete a task.

```
@Autowired
private RuntimeService runtimeService;

@Autowired
private TaskService taskService;

@Autowired
private FormService formService;

@PostMapping("/start-process")
public String startProcess(@ModelAttribute FormInformation formInformation) {

    ProcessInstance instance = runtimeService.startProcessInstanceByKey("myProcess");

    Task task = taskService.createTaskQuery()
              .processInstanceId(instance.getId())
              .taskDefinitionKey("firstTask")
              .singleResult();

    Map<String, String> formInformationDataMap = formInformation.getFormInformationDataMap();

    formService.submitTaskFormData(task.getId(), formInformationDataMap);

    return "Process started with ID: "
            + instance.getId()
            + "\nNumber of currently running process instances: "
            + runtimeService.createProcessInstanceQuery().count();
}
```

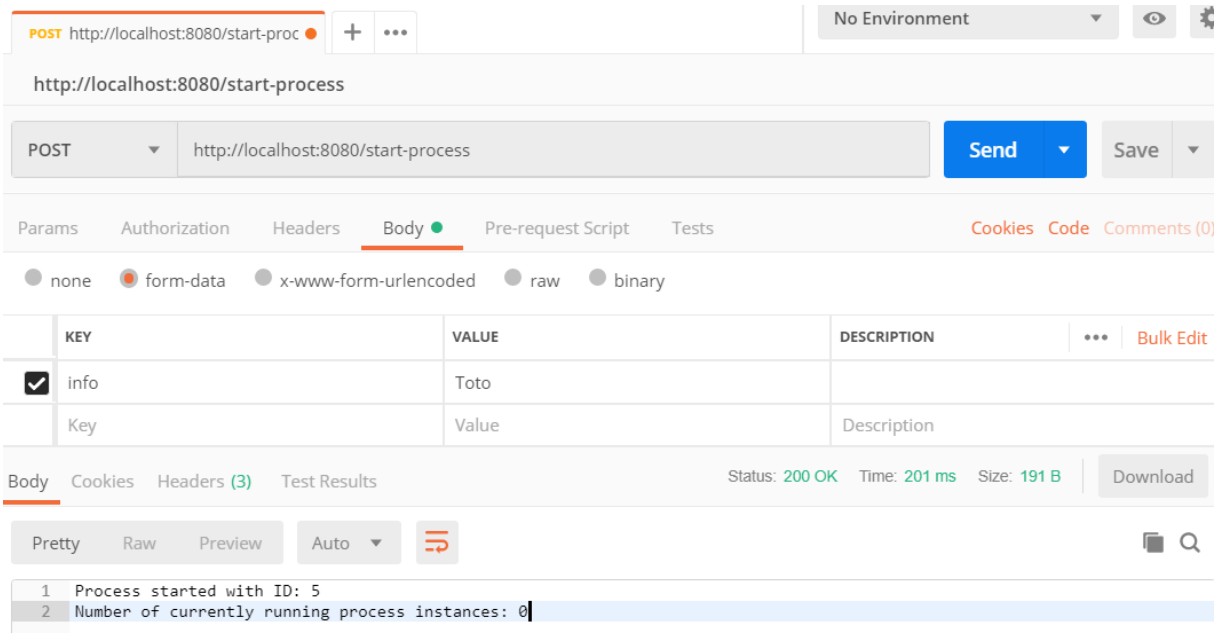
Create @GetMapping or @PostMapping methods depending on the desired type of request. A POST HTTP request allows for the submission of form data.

Useful methods:

- RuntimeService.startProcessInstanceByKey:
    - Use this method to start a process
    - The key is the ID of the process you have configured in the diagram's General properties
    - Make sure to get the process ID as you'll need it to interact with the generated process instance
- HistoryService.createHistoricProcessInstanceQuery:
    - Query the history of terminated process instances
    - For example, it's possible to determine if a given process instance has gone through a particular activity (eg. Approved, denied…).
- TaskService.createTaskQuery:
    - Use this method to get the task ID of a running process instance's task
- TaskService.complete:
    - Use this method to complete a task
- FormService.submitTaskFormData:
    - Alternatively, use this method if you with to complete a task while submitting a form

Using Postman to interact with the API

Create a test case for each of the methods mapped inside of your controller.



Develop a Vue.js client

Use the same methodology from the previous assignment.

https://github.com/raoul-imolczek/springboot-credit-calculator

Hints:

Vue.js allows to add attributes to HTML tags to trigger conditional display.

```
<div v-if="id == ''">
</div>

<div v-else-if="!(creditPropositionSubmitted && customerInformationSubmitted)">
</div>

<div v-else>
</div>
```

Vue.js model variables can be displayed using mustaches:

```
<td>{{ income }}</td>
```

They can also be binded with input fields:

```
<input v-model="firstName" type="text" class="form-control" id="firstName" aria-
describedby="firstNameHelp" placeholder="Enter your first name">
```

The vue.js app will implement an HTTP client to interact with the Springboot app's REST controller.

This is a sample POST request:

```
submitCreditProposition: function(event) {
	$.post('http://localhost:8080/submit-credit-proposition/' + this.id, {
		loanAmount: this.loanAmount,
		loanDuration: this.loanDuration
	}, function(data, status) {
		this.creditPropositionSubmitted = true;
		this.updateStatus();
	}.bind(this));
}
```

And this is a sample GET request :

```
updateStatus: function(event) {
	$.get('http://localhost:8080/status/' + this.id, function(data, status) {
		this.status = data;
	}.bind(this));
}
```

The difference is that a POST request includes the submitted form data. In the example, the form data is gathered from the Vue.js app's model (this.variableName).

The data received in the HTTP response is then saved in the Vue.js app's model in return.

Build a GUI with the required forms to interact with the process **as a customer**.

Forms allowing a credit agent to decide the approval of the credit would be available through a distinct GUI. We'll leave them to be done through POSTMAN.

Add anything to the project

Now that you're done, change anything to the project (eg. Add a task to the process) and implement the changes all the way through the application.

Reference

- Activiti user guide: http://www.activiti.org/userguide/
- Activiti in action, by Tijs Rademakers